S12CPUV2

Reference Manual

HCS12 Microcontrollers

S12CPUV2
Rev. 4.0
03/2006

freescale.com

freescale™
semiconductor

S12CPUV2

Reference Manual

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document.

Revision History

Revision Number	Date	Summary of Changes
3.0	April, 2002	Incorporated information covering HCS12 Family of 16-bit MCUs throughout the book.
4.0	March, 2006	Reformatted to Freescale publication standards. Corrected mistake in ANDCC/TAP descriptions (Instruction Glossary). Corrected mistake in MEM description (Instruction Glossary).

List of Sections

Table of Contents 7

Section 1. Introduction 15

Section 2. Overview 21

Section 3. Addressing Modes 29

Section 4. Instruction Queue 47

Section 5. Instruction Set Overview 55

Section 6. Instruction Glossary 87

Section 7. Exception Processing. 311

Section 8. Instruction Queue 323

Section 9. Fuzzy Logic Support. 337

Appendix A. Instruction Reference 375

Appendix B. M68HC11 to CPU12 Upgrade Path. 403

Appendix C. High-Level Language Support 425

Index 433

Table of Contents

Section 1. Introduction

1.1 Introduction 15
1.2 Features 15
1.3 Symbols and Notation. 16
1.3.1 Abbreviations for System Resources 16
1.3.2 Memory and Addressing. 17
1.3.3 Operators 18
1.3.4 Definitions. 19

Section 2. Overview

2.1 Introduction 21
2.2 Programming Model 21
2.2.1 Accumulators 22
2.2.2 Index Registers. 22
2.2.3 Stack Pointer 22
2.2.4 Program Counter 23
2.2.5 Condition Code Register 23
2.2.5.1 S Control Bit 24
2.2.5.2 X Mask Bit 25
2.2.5.3 H Status Bit 25
2.2.5.4 I Mask Bit 25
2.2.5.5 N Status Bit 26
2.2.5.6 Z Status Bit. 26
2.2.5.7 V Status Bit. 26
2.2.5.8 C Status Bit 26
2.3 Data Types 27
2.4 Memory Organization 27
2.5 Instruction Queue 28

Section 3. Addressing Modes

3.1 Introduction 29

3.2 Mode Summary. 29

3.3 Effective Address 29

3.4 Inherent Addressing Mode 31

3.5 Immediate Addressing Mode 31

3.6 Direct Addressing Mode 32

3.7 Extended Addressing Mode 33

3.8 Relative Addressing Mode 33

3.9 Indexed Addressing Modes 34
3.9.1 5-Bit Constant Offset Indexed Addressing 37
3.9.2 9-Bit Constant Offset Indexed Addressing 37
3.9.3 16-Bit Constant Offset Indexed Addressing 38
3.9.4 16-Bit Constant Indirect Indexed Addressing .38
3.9.5 Auto Pre/Post Decrement/Increment Indexed Addressing 39
3.9.6 Accumulator Offset Indexed Addressing 40
3.9.7 Accumulator D Indirect Indexed Addressing .41

3.10 Instructions Using Multiple Modes 41
3.10.1 Move Instructions 41
3.10.2 Bit Manipulation Instructions. 43

3.11 Addressing More than 64 Kbytes 44

Section 4. Instruction Queue

4.1 Introduction 47

4.2 Queue Description 47
4.2.1 Original M68HC12 Queue Implementation. 48
4.2.2 HCS12 Queue Implementation. 48

4.3 Data Movement in the Queue 48
4.3.1 No Movement 49
4.3.2 Latch Data from Bus (Applies Only to the M68HC12 Queue) 49
4.3.3 Advance and Load from Data Bus 49
4.3.4 Advance and Load from Buffer (Applies Only to M68HC12 Queue) . . .49

4.4 Changes in Execution Flow 49
4.4.1 Exceptions 50
4.4.2 Subroutines 50
4.4.3 Branches 51

4.4.3.1 Short Branches. 52
4.4.3.2 Long Branches 52
4.4.3.3 Bit Condition Branches. 53
4.4.3.4 Loop Primitives. 53
4.4.4 Jumps 53

Section 5. Instruction Set Overview

5.1 Introduction 55
5.2 Instruction Set Description 55
5.3 Load and Store Instructions 56
5.4 Transfer and Exchange Instructions 57
5.5 Move Instructions 58
5.6 Addition and Subtraction Instructions 59
5.7 Binary-Coded Decimal Instructions. 60
5.8 Decrement and Increment Instructions 61
5.9 Compare and Test Instructions. 62
5.10 Boolean Logic Instructions 63
5.11 Clear, Complement, and Negate Instructions 63
5.12 Multiplication and Division Instructions 64
5.13 Bit Test and Manipulation Instructions 65
5.14 Shift and Rotate Instructions. 66
5.15 Fuzzy Logic Instructions 67
5.15.1 Fuzzy Logic Membership Instruction 67
5.15.2 Fuzzy Logic Rule Evaluation Instructions. 67
5.15.3 Fuzzy Logic Weighted Average Instruction 68
5.16 Maximum and Minimum Instructions 70
5.17 Multiply and Accumulate Instruction 71
5.18 Table Interpolation Instructions. 72
5.19 Branch Instructions 73
5.19.1 Short Branch Instructions 74
5.19.2 Long Branch Instructions 75
5.19.3 Bit Condition Branch Instructions 76
5.20 Loop Primitive Instructions 77
5.21 Jump and Subroutine Instructions 78

5.22 Interrupt Instructions 79
5.23 Index Manipulation Instructions 81
5.24 Stacking Instructions. 82
5.25 Pointer and Index Calculation Instructions 83
5.26 Condition Code Instructions 84
5.27 Stop and Wait Instructions 85
5.28 Background Mode and Null Operations 86

Section 6. Instruction Glossary

6.1 Introduction 87
6.2 Glossary Information. 88
6.3 Condition Code Changes 89
6.4 Object Code Notation 90
6.5 Source Forms 91
6.6 Cycle-by-Cycle Execution. 94
6.7 Glossary 99

Section 7. Exception Processing

7.1 Introduction 311
7.2 Types of Exceptions 311
7.3 Exception Priority 312
7.4 Resets. 313
7.4.1 Power-On Reset 314
7.4.2 External Reset 314
7.4.3 COP Reset 314
7.4.4 Clock Monitor Reset 314
7.5 Interrupts. 315
7.5.1 Non-Maskable Interrupt Request ($\overline{\text{XIRQ}}$) 315
7.5.2 Maskable Interrupts 315
7.5.3 Interrupt Recognition 316
7.5.4 External Interrupts 317
7.5.5 Return-from-Interrupt Instruction (RTI) 317
7.6 Unimplemented Opcode Trap. 317
7.7 Software Interrupt Instruction (SWI) 318

7.8 Exception Processing Flow 318
7.8.1 Vector Fetch 318
7.8.2 Reset Exception Processing 320
7.8.3 Interrupt and Unimplemented Opcode Trap Exception Processing . . 320

Section 8. Instruction Queue

8.1 Introduction 323

8.2 External Reconstruction of the Queue 323

8.3 Instruction Queue Status Signals 324
8.3.1 HCS12 Timing Detail 324
8.3.2 M68HC12 Timing Detail 325
8.3.3 Null (Code 0:0) 327
8.3.4 LAT — Latch Data from Bus (Code 0:1) 327
8.3.5 ALD — Advance and Load from Data Bus (Code 1:0) 327
8.3.6 ALL — Advance and Load from Latch (Code 1:1) 327
8.3.7 INT — Interrupt Sequence Start (Code 0:1) 327
8.3.8 SEV — Start Instruction on Even Address (Code 1:0) 328
8.3.9 SOD — Start Instruction on Odd Address (Code 1:1) 328

8.4 Queue Reconstruction (for HCS12) 328
8.4.1 Queue Reconstruction Registers (for HCS12) 329
8.4.1.1 fetch_add Register 329
8.4.1.2 st1_add, st1_dat Registers 329
8.4.1.3 st2_add, st2_dat Registers 329
8.4.1.4 st3_add, st3_dat Registers 329
8.4.2 Reconstruction Algorithm (for HCS12) 330

8.5 Queue Reconstruction (for M68HC12) 331
8.5.1 Queue Reconstruction Registers (for M68HC12) 331
8.5.1.1 in_add, in_dat Registers 332
8.5.1.2 fetch_add, fetch_dat Registers 332
8.5.1.3 st1_add, st1_dat Registers 332
8.5.1.4 st2_add, st2_dat Registers 332
8.5.2 Reconstruction Algorithm (for M68HC12) 332
8.5.2.1 LAT Decoding 333
8.5.2.2 ALD Decoding 333
8.5.2.3 ALL Decoding 333

8.6 Instruction Tagging 335

Section 9. Fuzzy Logic Support

9.1 Introduction 337
9.2 Fuzzy Logic Basics 338
9.2.1 Fuzzification (MEM) 340
9.2.2 Rule Evaluation (REV and REVW) 342
9.2.3 Defuzzification (WAV) 344
9.3 Example Inference Kernel 345
9.4 MEM Instruction Details 347
9.4.1 Membership Function Definitions 347
9.4.2 Abnormal Membership Function Definitions 349
9.4.2.1 Abnormal Membership Function Case 1 .351
9.4.2.2 Abnormal Membership Function Case 2 .352
9.4.2.3 Abnormal Membership Function Case 3 .352
9.5 REV and REVW Instruction Details 353
9.5.1 Unweighted Rule Evaluation (REV) 353
9.5.1.1 Set Up Prior to Executing REV. 353
9.5.1.2 Interrupt Details 355
9.5.1.3 Cycle-by-Cycle Details for REV 355
9.5.2 Weighted Rule Evaluation (REVW) 359
9.5.2.1 Set Up Prior to Executing REVW 359
9.5.2.2 Interrupt Details 361
9.5.2.3 Cycle-by-Cycle Details for REVW 361
9.6 WAV Instruction Details 364
9.6.1 Set Up Prior to Executing WAV 364
9.6.2 WAV Interrupt Details 365
9.6.3 Cycle-by-Cycle Details for WAV and wavr 365
9.7 Custom Fuzzy Logic Programming 369
9.7.1 Fuzzification Variations. 369
9.7.2 Rule Evaluation Variations 372
9.7.3 Defuzzification Variations 373

Appendix A. Instruction Reference

A.1 Introduction 375
A.2 Stack and Memory Layout 376
A.3 Interrupt Vector Locations. 376
A.4 Notation Used in Instruction Set Summary 377
A.5 Hexadecimal to Decimal Conversion 402

A.6 Decimal to Hexadecimal Conversion 402

Appendix B. M68HC11 to CPU12 Upgrade Path

B.1 Introduction 403
B.2 CPU12 Design Goals 403
B.3 Source Code Compatibility 404
B.4 Programmer's Model and Stacking. 407
B.5 True 16-Bit Architecture 407
B.5.1 Bus Structures 407
B.5.2 Instruction Queue 408
B.5.3 Stack Function 409
B.6 Improved Indexing 410
B.6.1 Constant Offset Indexing 412
B.6.2 Auto-Increment Indexing 413
B.6.3 Accumulator Offset Indexing 414
B.6.4 Indirect Indexing 414
B.7 Improved Performance 415
B.7.1 Reduced Cycle Counts. 415
B.7.2 Fast Math 415
B.7.3 Code Size Reduction 416
B.8 Additional Functions 417
B.8.1 Memory-to-Memory Moves. 420
B.8.2 Universal Transfer and Exchange 420
B.8.3 Loop Construct 420
B.8.4 Long Branches 421
B.8.5 Minimum and Maximum Instructions 421
B.8.6 Fuzzy Logic Support. 422
B.8.7 Table Lookup and Interpolation 422
B.8.8 Extended Bit Manipulation 423
B.8.9 Push and Pull D and CCR 423
B.8.10 Compare SP 423
B.8.11 Support for Memory Expansion 424

Appendix C. High-Level Language Support

C.1 Introduction 425
C.2 Data Types 425

C.3 Parameters and Variables 426
C.3.1 Register Pushes and Pulls 426
C.3.2 Allocating and Deallocating Stack Space 427
C.3.3 Frame Pointer 427
C.4 Increment and Decrement Operators 428
C.5 Higher Math Functions 428
C.6 Conditional If Constructs 429
C.7 Case and Switch Statements 429
C.8 Pointers 429
C.9 Function Calls 430
C.10 Instruction Set Orthogonality 431

Section 1. Introduction

1.1 Introduction

This manual describes the features and operation of the core (central processing unit, or CPU, and development support functions) used in all HCS12 microcontrollers. For reference, information is provided for the M68HC12.

1.2 Features

The CPU12 is a high-speed, 16-bit processing unit that has a programming model identical to that of the industry standard M68HC11 central processor unit (CPU). The CPU12 instruction set is a proper superset of the M68HC11 instruction set, and M68HC11 source code is accepted by CPU12 assemblers with no changes.

- Full 16-bit data paths supports efficient arithmetic operation and high-speed math execution
- Supports instructions with odd byte counts, including many single-byte instructions. This allows much more efficient use of ROM space.
- An instruction queue buffers program information so the CPU has immediate access to at least three bytes of machine code at the start of every instruction.
- Extensive set of indexed addressing capabilities, including:
  - Using the stack pointer as an indexing register in all indexed operations
  - Using the program counter as an indexing register in all but auto increment/decrement mode
  - Accumulator offsets using A, B, or D accumulators
  - Automatic index predecrement, preincrement, postdecrement, and postincrement (by –8 to +8)

1.3 Symbols and Notation

The symbols and notation shown here are used throughout the manual. More specialized notation that applies only to the instruction glossary or instruction set summary are described at the beginning of those sections.

1.3.1 Abbreviations for System Resources

A — Accumulator A
B — Accumulator B
D — Double accumulator D (A : B)
X — Index register X
Y — Index register Y
SP — Stack pointer
PC — Program counter
CCR — Condition code register

- S — STOP instruction control bit
- X — Non-maskable interrupt control bit
- H — Half-carry status bit
- I — Maskable interrupt control bit
- N — Negative status bit
- Z — Zero status bit
- V — Two’s complement overflow status bit
- C — Carry/Borrow status bit

1.3.2 Memory and Addressing

M — 8-bit memory location pointed to by the effective address of the instruction

M : M+1 — 16-bit memory location. Consists of the contents of the location pointed to by the effective address concatenated with the contents of the location at the next higher memory address. The most significant byte is at location M.

M~M+3
$M_{(Y)}$~$M_{(Y+3)}$ — 32-bit memory location. Consists of the contents of the effective address of the instruction concatenated with the contents of the next three higher memory locations. The most significant byte is at location M or $M_{(Y)}$.

$M_{(X)}$ — Memory locations pointed to by index register X

$M_{(SP)}$ — Memory locations pointed to by the stack pointer

$M_{(Y+3)}$ — Memory locations pointed to by index register Y plus 3

PPAGE — Program overlay page (bank) number for extended memory (>64 Kbytes).

Page — Program overlay page

X_H — High-order byte

X_L — Low-order byte

( ) — Content of register or memory location

$ — Hexadecimal value

% — Binary value

1.3.3 Operators

+ — Addition

– — Subtraction

• — Logical AND

+ — Logical OR (inclusive)

⊕ — Logical exclusive OR

× — Multiplication

÷ — Division

$\overline{\text{M}}$ — Negation. One’s complement (invert each bit of M)

: — Concatenate

Example: A : B means the 16-bit value formed by concatenating 8-bit accumulator A with 8-bit accumulator B.
A is in the high-order position.

⇒ — Transfer

Example: (A) ⇒ M means the content of accumulator A is transferred to memory location M.

⇔ — Exchange

Example: D ⇔ X means exchange the contents of D with those of X.

1.3.4 Definitions

Logic level 1 is the voltage that corresponds to the true (1) state.

Logic level 0 is the voltage that corresponds to the false (0) state.

Set refers specifically to establishing logic level 1 on a bit or bits.

Cleared refers specifically to establishing logic level 0 on a bit or bits.

Asserted means that a signal is in active logic state. An active low signal changes from logic level 1 to logic level 0 when asserted, and an active high signal changes from logic level 0 to logic level 1.

Negated means that an asserted signal changes logic state. An active low signal changes from logic level 0 to logic level 1 when negated, and an active high signal changes from logic level 1 to logic level 0.

ADDR is the mnemonic for address bus.

DATA is the mnemonic for data bus.

LSB means least significant bit or bits.

MSB means most significant bit or bits.

LSW means least significant word or words.

MSW means most significant word or words.

A specific bit location within a range is referred to by mnemonic and number. For example, A7 is bit 7 of accumulator A.

A range of bit locations is referred to by mnemonic and the numbers that define the range. For example, DATA[15:8] form the high byte of the data bus.

Section 2. Overview

2.1 Introduction

This section describes the CPU12 programming model, register set, the data types used, and basic memory organization.

2.2 Programming Model

The CPU12 programming model, shown in **Figure 2-1**, is the same as that of the M68HC11 CPU. The CPU has two 8-bit general-purpose accumulators (A and B) that can be concatenated into a single 16-bit accumulator (D) for certain instructions. It also has:

- Two index registers (X and Y)
- 16-bit stack pointer (SP)
- 16-bit program counter (PC)
- 8-bit condition code register (CCR)

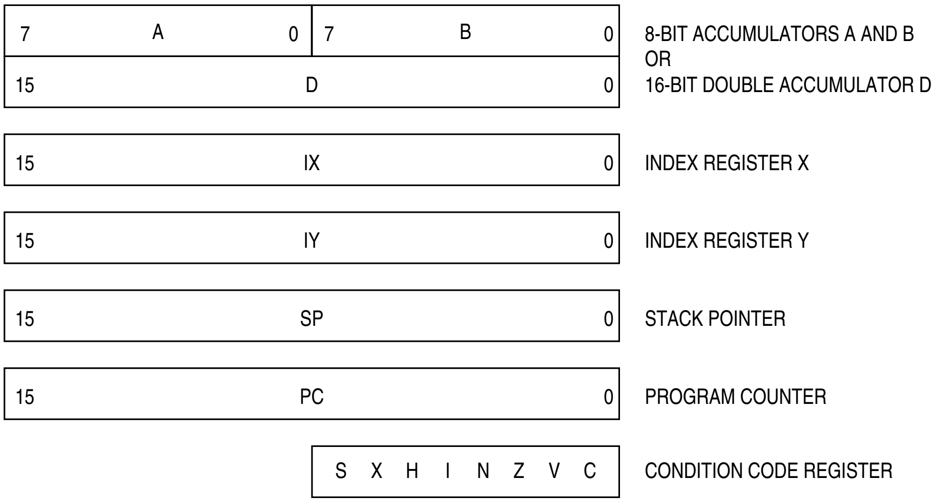


Figure 2-1. Programming Model

2.2.1 Accumulators

General-purpose 8-bit accumulators A and B are used to hold operands and results of operations. Some instructions treat the combination of these two 8-bit accumulators (A : B) as a 16-bit double accumulator (D).

Most operations can use accumulator A or B interchangeably. However, there are a few exceptions. Add, subtract, and compare instructions involving both A and B (ABA, SBA, and CBA) only operate in one direction, so it is important to make certain the correct operand is in the correct accumulator. The decimal adjust accumulator A (DAA) instruction is used after binary-coded decimal (BCD) arithmetic operations. There is no equivalent instruction to adjust accumulator B.

2.2.2 Index Registers

16-bit index registers X and Y are used for indexed addressing. In the indexed addressing modes, the contents of an index register are added to 5-bit, 9-bit, or 16-bit constants or to the content of an accumulator to form the effective address of the instruction operand. The second index register is especially useful for moves and in cases where operands from two separate tables are used in a calculation.

2.2.3 Stack Pointer

The CPU12 supports an automatic program stack. The stack is used to save system context during subroutine calls and interrupts and can also be used for temporary data storage. The stack can be located anywhere in the standard 64-Kbyte address space and can grow to any size up to the total amount of memory available in the system.

The stack pointer (SP) holds the 16-bit address of the last stack location used. Normally, the SP is initialized by one of the first instructions in an application program. The stack grows downward from the address pointed to by the SP. Each time a byte is pushed onto the stack, the stack pointer is automatically decremented, and each time a byte is pulled from the stack, the stack pointer is automatically incremented.

When a subroutine is called, the address of the instruction following the calling instruction is automatically calculated and pushed onto the stack. Normally, a return-from-subroutine (RTS) or a return-from-call (RTC) instruction is executed at the end of a subroutine. The return instruction loads the program counter with the previously stacked return address and execution continues at that address.

When an interrupt occurs, the current instruction finishes execution. The address of the next instruction is calculated and pushed onto the stack, all the CPU registers are pushed onto the stack, the program counter is loaded with the address pointed to by the interrupt vector, and execution continues at that address. The stacked registers are referred to as an interrupt stack frame. The CPU12 stack frame is the same as that of the M68HC11.

NOTE: *These instructions can be interrupted, and they resume execution once the interrupt has been serviced:*

- *REV (fuzzy logic rule evaluation)*
- *REVW (fuzzy logic rule evaluation (weighted))*
- *WAV (weighted average)*

2.2.4 Program Counter

The program counter (PC) is a 16-bit register that holds the address of the next instruction to be executed. It is automatically incremented each time an instruction is fetched.

2.2.5 Condition Code Register

The condition code register (CCR), named for its five status indicators, contains:

- Five status indicators
- Two interrupt masking bits
- STOP instruction control bit

The status bits reflect the results of CPU operation as it executes instructions. The five flags are:

- Half carry (H)
- Negative (N)
- Zero (Z)
- Overflow (V)
- Carry/borrow (C)

The half-carry flag is used only for BCD arithmetic operations. The N, Z, V, and C status bits allow for branching based on the results of a previous operation.

In some architectures, only a few instructions affect condition codes, so that multiple instructions must be executed in order to load and test a variable. Since most CPU12 instructions automatically update condition codes, it is rarely necessary to execute an extra instruction for this purpose. The challenge in using the CPU12 lies in finding instructions that do not alter the condition codes. The most important of these instructions are pushes, pulls, transfers, and exchanges.

It is always a good idea to refer to an instruction set summary (see **Appendix A. Instruction Reference**) to check which condition codes are affected by a particular instruction.

The following paragraphs describe normal uses of the condition codes. There are other, more specialized uses. For instance, the C status bit is used to enable weighted fuzzy logic rule evaluation. Specialized usages are described in the relevant portions of this manual and in **Section 6. Instruction Glossary**.

2.2.5.1 S Control Bit

Clearing the S bit enables the STOP instruction. Execution of a STOP instruction normally causes the on-chip oscillator to stop. This may be undesirable in some applications. If the CPU encounters a STOP instruction while the S bit is set, it is treated like a no-operation (NOP) instruction and continues to the next instruction. Reset sets the S bit.

2.2.5.2 X Mask Bit

The $\overline{\text{XIRQ}}$ input is an updated version of the $\overline{\text{NMI}}$ input found on earlier generations of MCUs. Non-maskable interrupts are typically used to deal with major system failures, such as loss of power. However, enabling non-maskable interrupts before a system is fully powered and initialized can lead to spurious interrupts. The X bit provides a mechanism for enabling non-maskable interrupts after a system is stable.

By default, the X bit is set to 1 during reset. As long as the X bit remains set, interrupt service requests made via the $\overline{\text{XIRQ}}$ pin are not recognized. An instruction must clear the X bit to enable non-maskable interrupt service requests made via the $\overline{\text{XIRQ}}$ pin. Once the X bit has been cleared to 0, software cannot reset it to 1 by writing to the CCR. The X bit is not affected by maskable interrupts.

When an $\overline{\text{XIRQ}}$ interrupt occurs after non-maskable interrupts are enabled, both the X bit and the I bit are set automatically to prevent other interrupts from being recognized during the interrupt service routine. The mask bits are set after the registers are stacked, but before the interrupt vector is fetched.

Normally, a return-from-interrupt (RTI) instruction at the end of the interrupt service routine restores register values that were present before the interrupt occurred. Since the CCR is stacked before the X bit is set, the RTI normally clears the X bit, and thus re-enables non-maskable interrupts. While it is possible to manipulate the stacked value of X so that X is set after an RTI, there is no software method to reset X (and disable $\overline{\text{XIRQ}}$) once X has been cleared.

2.2.5.3 H Status Bit

The H bit indicates a carry from accumulator A bit 3 during an addition operation. The DAA instruction uses the value of the H bit to adjust a result in accumulator A to correct BCD format. H is updated only by the add accumulator A to accumulator B (ABA), add without carry (ADD), and add with carry (ADC) instructions.

2.2.5.4 I Mask Bit

The I bit enables and disables maskable interrupt sources. By default, the I bit is set to 1 during reset. An instruction must clear the I bit to enable maskable interrupts. While the I bit is set, maskable interrupts can become pending and are remembered, but operation continues uninterrupted until the I bit is cleared.

When an interrupt occurs after interrupts are enabled, the I bit is automatically set to prevent other maskable interrupts during the interrupt service routine. The I bit is set after the registers are stacked, but before the first instruction in the interrupt service routine is executed.

Normally, an RTI instruction at the end of the interrupt service routine restores register values that were present before the interrupt occurred. Since the CCR is stacked before the I bit is set, the RTI normally clears the I bit, and thus re-enables interrupts. Interrupts can be re-enabled by clearing the I bit within the service routine.

2.2.5.5 N Status Bit

The N bit shows the state of the MSB of the result. N is most commonly used in two’s complement arithmetic, where the MSB of a negative number is 1 and the MSB of a positive number is 0, but it has other uses. For instance, if the MSB of a register or memory location is used as a status flag, the user can test status by loading an accumulator.

2.2.5.6 Z Status Bit

The Z bit is set when all the bits of the result are 0s. Compare instructions perform an internal implied subtraction, and the condition codes, including Z, reflect the results of that subtraction. The increment index register X (INX), decrement index register X (DEX), increment index register Y (INY), and decrement index register Y (DEY) instructions affect the Z bit and no other condition flags. These operations can only determine = (equal) and ≠ (not equal).

2.2.5.7 V Status Bit

The V bit is set when two’s complement overflow occurs as a result of an operation.

2.2.5.8 C Status Bit

The C bit is set when a carry occurs during addition or a borrow occurs during subtraction. The C bit also acts as an error flag for multiply and divide operations. Shift and rotate instructions operate through the C bit to facilitate multiple-word shifts.

2.3 Data Types

The CPU12 uses these types of data:

- Bits
- 5-bit signed integers
- 8-bit signed and unsigned integers
- 8-bit, 2-digit binary-coded decimal numbers
- 9-bit signed integers
- 16-bit signed and unsigned integers
- 16-bit effective addresses
- 32-bit signed and unsigned integers

Negative integers are represented in two’s complement form.

Five-bit and 9-bit signed integers are used only as offsets for indexed addressing modes.

Sixteen-bit effective addresses are formed during addressing mode computations.

Thirty-two-bit integer dividends are used by extended division instructions. Extended multiply and extended multiply-and-accumulate instructions produce 32-bit products.

2.4 Memory Organization

The standard CPU12 address space is 64 Kbytes. Some M68HC12 devices support a paged memory expansion scheme that increases the standard space by means of predefined windows in address space. The CPU12 has special instructions that support use of expanded memory.

Eight-bit values can be stored at any odd or even byte address in available memory.

Sixteen-bit values are stored in memory as two consecutive bytes; the high byte occupies the lowest address, but need not be aligned to an even boundary.

Thirty-two-bit values are stored in memory as four consecutive bytes; the high byte occupies the lowest address, but need not be aligned to an even boundary.

All input/output (I/O) and all on-chip peripherals are memory-mapped. No special instruction syntax is required to access these addresses. On-chip registers and memory typically are grouped in blocks which can be relocated within the standard 64-Kbyte address space. Refer to device documentation for specific information.

2.5 Instruction Queue

The CPU12 uses an instruction queue to buffer program information. The mechanism is called a queue rather than a pipeline because a typical pipelined CPU executes more than one instruction at the same time, while the CPU12 always finishes executing an instruction before beginning to execute another. Refer to **Section 4. Instruction Queue** for more information.

Section 3. Addressing Modes

3.1 Introduction

Addressing modes determine how the central processor unit (CPU) accesses memory locations to be operated upon. This section discusses the various modes and how they are used.

3.2 Mode Summary

Addressing modes are an implicit part of CPU12 instructions. Refer to **Appendix A. Instruction Reference** for the modes used by each instruction. All CPU12 addressing modes are shown in **Table 3-1**.

The CPU12 uses all M68HC11 modes as well as new forms of indexed addressing. Differences between M68HC11 and M68HC12 indexed modes are described in **3.9 Indexed Addressing Modes**. Instructions that use more than one mode are discussed in **3.10 Instructions Using Multiple Modes**.

3.3 Effective Address

Each addressing mode except inherent mode generates a 16-bit effective address which is used during the memory reference portion of the instruction. Effective address computations do not require extra execution cycles.

Table 3-1. M68HC12 Addressing Mode Summary

Addressing Mode	Source Format	Abbreviation	Description
Inherent	**INST** (no externally supplied operands)	INH	Operands (if any) are in CPU registers
Immediate	**INST** *#opr8i* or **INST** *#opr16i*	IMM	Operand is included in instruction stream 8- or 16-bit size implied by context
Direct	**INST** *opr8a*	DIR	Operand is the lower 8 bits of an address in the range \$0000–\$00FF
Extended	**INST** *opr16a*	EXT	Operand is a 16-bit address
Relative	**INST** *rel8* or **INST** *rel16*	REL	An 8-bit or 16-bit relative offset from the current pc is supplied in the instruction
Indexed (5-bit offset)	**INST** *oprx5*,*xysp*	IDX	5-bit signed constant offset from X, Y, SP, or PC
Indexed (pre-decrement)	**INST** *oprx3*,**–***xys*	IDX	Auto pre-decrement x, y, or sp by 1 ~ 8
Indexed (pre-increment)	**INST** *oprx3*,**+***xys*	IDX	Auto pre-increment x, y, or sp by 1 ~ 8
Indexed (post-decrement)	**INST** *oprx3*,*xys***–**	IDX	Auto post-decrement x, y, or sp by 1 ~ 8
Indexed (post-increment)	**INST** *oprx3*,*xys***+**	IDX	Auto post-increment x, y, or sp by 1 ~ 8
Indexed (accumulator offset)	**INST** *abd*,*xysp*	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from X, Y, SP, or PC
Indexed (9-bit offset)	**INST** *oprx9*,*xysp*	IDX1	9-bit signed constant offset from X, Y, SP, or PC (lower 8 bits of offset in one extension byte)
Indexed (16-bit offset)	**INST** *oprx16*,*xysp*	IDX2	16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (16-bit offset)	**INST [***oprx16*,*xysp***]**	[IDX2]	Pointer to operand is found at... 16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (D accumulator offset)	**INST [D,***xysp***]**	[D,IDX]	Pointer to operand is found at... X, Y, SP, or PC plus the value in D

3.4 Inherent Addressing Mode

Instructions that use this addressing mode either have no operands or all operands are in internal CPU registers. In either case, the CPU does not need to access any memory locations to complete the instruction.

Examples:

```
NOP   ;this instruction has no operands
INX   ;operand is a CPU register
```

3.5 Immediate Addressing Mode

Operands for immediate mode instructions are included in the instruction stream and are fetched into the instruction queue one 16-bit word at a time during normal program fetch cycles. Since program data is read into the instruction queue several cycles before it is needed, when an immediate addressing mode operand is called for by an instruction, it is already present in the instruction queue.

The pound symbol (#) is used to indicate an immediate addressing mode operand. One common programming error is to accidentally omit the # symbol. This causes the assembler to misinterpret the expression that follows it as an address rather than explicitly provided data. For example, LDAA #$55 means to load the immediate value $55 into the A accumulator, while LDAA $55 means to load the value from address $0055 into the A accumulator. Without the # symbol, the instruction is erroneously interpreted as a direct addressing mode instruction.

Examples:

```
LDAA        #$55
LDX         #$1234
LDY         #$67
```

These are common examples of 8-bit and 16-bit immediate addressing modes. The size of the immediate operand is implied by the instruction context. In the third example, the instruction implies a 16-bit immediate value but only an 8-bit value is supplied. In this case the assembler will generate the 16-bit value $0067 because the CPU expects a 16-bit value in the instruction stream.

Example:

```
BRSET       FOO,#$03,THERE
```

In this example, extended addressing mode is used to access the operand FOO, immediate addressing mode is used to access the mask value $03, and relative addressing mode is used to identify the destination address of a branch in case the branch-taken conditions are met. BRSET is listed as an extended mode instruction even though immediate and relative modes are also used.

3.6 Direct Addressing Mode

This addressing mode is sometimes called zero-page addressing because it is used to access operands in the address range $0000 through $00FF. Since these addresses always begin with $00, only the eight low-order bits of the address need to be included in the instruction, which saves program space and execution time. A system can be optimized by placing the most commonly accessed data in this area of memory. The eight low-order bits of the operand address are supplied with the instruction, and the eight high-order bits of the address are assumed to be 0.

Example:

```
LDAA          $55
```

This is a basic example of direct addressing. The value $55 is taken to be the low-order half of an address in the range $0000 through $00FF. The high order half of the address is assumed to be 0. During execution of this instruction, the CPU combines the value $55 from the instruction with the assumed value of $00 to form the address $0055, which is then used to access the data to be loaded into accumulator A.

Example:

```
LDX           $20
```

In this example, the value $20 is combined with the assumed value of $00 to form the address $0020. Since the LDX instruction requires a 16-bit value, a 16-bit word of data is read from addresses $0020 and $0021. After execution of this instruction, the X index register will have the value from address $0020 in its high-order half and the value from address $0021 in its low-order half.

3.7 Extended Addressing Mode

In this addressing mode, the full 16-bit address of the memory location to be operated on is provided in the instruction. This addressing mode can be used to access any location in the 64-Kbyte memory map.

Example:

```
    LDAA          $F03B
```

This is a basic example of extended addressing. The value from address $F03B is loaded into the A accumulator.

3.8 Relative Addressing Mode

The relative addressing mode is used only by branch instructions. Short and long conditional branch instructions use relative addressing mode exclusively, but branching versions of bit manipulation instructions (branch if bits set (BRSET) and branch if bits cleared (BRCLR)) use multiple addressing modes, including relative mode. Refer to **3.10 Instructions Using Multiple Modes** for more information.

Short branch instructions consist of an 8-bit opcode and a signed 8-bit offset contained in the byte that follows the opcode. Long branch instructions consist of an 8-bit prebyte, an 8-bit opcode, and a signed 16-bit offset contained in the two bytes that follow the opcode.

Each conditional branch instruction tests certain status bits in the condition code register. If the bits are in a specified state, the offset is added to the address of the next memory location after the offset to form an effective address, and execution continues at that address. If the bits are not in the specified state, execution continues with the instruction immediately following the branch instruction.

Bit-condition branches test whether bits in a memory byte are in a specific state. Various addressing modes can be used to access the memory location. An 8-bit mask operand is used to test the bits. If each bit in memory that corresponds to a 1 in the mask is either set (BRSET) or clear (BRCLR), an 8-bit offset is added to the address of the next memory location after the offset to form an effective address, and execution continues at that address. If all the bits in memory that correspond to a 1 in the mask are not in the specified state, execution continues with the instruction immediately following the branch instruction.

8-bit, 9-bit, and 16-bit offsets are signed two's complement numbers to support branching upward and downward in memory. The numeric range of short branch offset values is $80 (–128) to $7F (127). Loop primitive instructions support a 9-bit offset which allows a range of $100 (–256) to $0FF (255). The numeric range of long branch offset values is $8000 (–32,768) to $7FFF (32,767). If the offset is 0, the CPU executes the instruction immediately following the branch instruction, regardless of the test involved.

Since the offset is at the end of a branch instruction, using a negative offset value can cause the program counter (PC) to point to the opcode and initiate a loop. For instance, a branch always (BRA) instruction consists of two bytes, so using an offset of $FE sets up an infinite loop; the same is true of a long branch always (LBRA) instruction with an offset of $FFFC.

An offset that points to the opcode can cause a bit-condition branch to repeat execution until the specified bit condition is satisfied. Since bit-condition branches can consist of four, five, or six bytes depending on the addressing mode used to access the byte in memory, the offset value that sets up a loop can vary. For instance, using an offset of $FC with a BRCLR that accesses memory using an 8-bit indexed postbyte sets up a loop that executes until all the bits in the specified memory byte that correspond to 1s in the mask byte are cleared.

3.9 Indexed Addressing Modes

The CPU12 uses redefined versions of M68HC11 indexed modes that reduce execution time and eliminate code size penalties for using the Y index register. In most cases, CPU12 code size for indexed operations is the same or is smaller than that for the M68HC11. Execution time is shorter in all cases. Execution time improvements are due to both a reduced number of cycles for all indexed instructions and to faster system clock speed.

The indexed addressing scheme uses a postbyte plus zero, one, or two extension bytes after the instruction opcode. The postbyte and extensions do the following tasks:

1. Specify which index register is used
2. Determine whether a value in an accumulator is used as an offset
3. Enable automatic pre- or post-increment or pre- or post-decrement
4. Specify size of increment or decrement
5. Specify use of 5-, 9-, or 16-bit signed offsets

This approach eliminates the differences between X and Y register use while dramatically enhancing the indexed addressing capabilities.

Major advantages of the CPU12 indexed addressing scheme are:

- The stack pointer can be used as an index register in all indexed operations.
- The program counter can be used as an index register in all but autoincrement and autodecrement modes.
- A, B, or D accumulators can be used for accumulator offsets.
- Automatic pre- or post-increment or pre- or post-decrement by –8 to +8
- A choice of 5-, 9-, or 16-bit signed constant offsets
- Use of two new indexed-indirect modes:
  - Indexed-indirect mode with 16-bit offset
  - Indexed-indirect mode with accumulator D offset

Table 3-2 is a summary of indexed addressing mode capabilities and a description of postbyte encoding. The postbyte is noted as xb in instruction descriptions. Detailed descriptions of the indexed addressing mode variations follow the table.

All indexed addressing modes use a 16-bit CPU register and additional information to create an effective address. In most cases the effective address specifies the memory location affected by the operation. In some variations of indexed addressing, the effective address specifies the location of a value that points to the memory location affected by the operation.

Table 3-2. Summary of Indexed Operations

Postbyte Code (xb)	Source Code Syntax	Comments rr; 00 = X, 01 = Y, 10 = SP, 11 = PC
rr0nnnnn	,r n,r –n,r	**5-bit constant offset** n = –16 to +15 r can specify X, Y, SP, or PC
111rr0zs	n,r –n,r	**Constant offset** (9- or 16-bit signed) z- 0 = 9-bit with sign in LSB of postbyte(s) –256 ≤ n ≤ 255 1 = 16-bit –32,768 ≤ n ≤ 65,535 if z = s = 1, 16-bit offset indexed-indirect (see below) r can specify X, Y, SP, or PC
111rr011	[n,r]	**16-bit offset indexed-indirect** rr can specify X, Y, SP, or PC –32,768 ≤ n ≤ 65,535
rr1pnnnn	n,–r n,+r n,r– n,r+	**Auto predecrement**, **preincrement**, **postdecrement**, or **postincrement**; p = pre-(0) or post-(1), n = –8 to –1, +1 to +8 r can specify X, Y, or SP (PC not a valid choice) +8 = 0111 … +1 = 0000 –1 = 1111 … –8 = 1000
111rr1aa	A,r B,r D,r	**Accumulator offset** (unsigned 8-bit or 16-bit) aa-00 = A 01 = B 10 = D (16-bit) 11 = see accumulator D offset indexed-indirect r can specify X, Y, SP, or PC
111rr111	[D,r]	**Accumulator D offset indexed-indirect** r can specify X, Y, SP, or PC

Indexed addressing mode instructions use a postbyte to specify index registers (X and Y), stack pointer (SP), or program counter (PC) as the base index register and to further classify the way the effective address is formed. A special group of instructions cause this calculated effective address to be loaded into an index register for further calculations:

- Load stack pointer with effective address (LEAS)
- Load X with effective address (LEAX)
- Load Y with effective address (LEAY)

3.9.1 5-Bit Constant Offset Indexed Addressing

This indexed addressing mode uses a 5-bit signed offset which is included in the instruction postbyte. This short offset is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location that will be affected by the instruction. This gives a range of –16 through +15 from the value in the base index register. Although other indexed addressing modes allow 9- or 16-bit offsets, those modes also require additional extension bytes in the instruction for this extra information. The majority of indexed instructions in real programs use offsets that fit in the shortest 5-bit form of indexed addressing.

Examples:

```
LDAA          0,X
STAB          -8,Y
```

For these examples, assume X has a value of $1000 and Y has a value of $2000 before execution. The 5-bit constant offset mode does not change the value in the index register, so X will still be $1000 and Y will still be $2000 after execution of these instructions. In the first example, A will be loaded with the value from address $1000. In the second example, the value from the B accumulator will be stored at address $1FF8 ($2000 –$8).

3.9.2 9-Bit Constant Offset Indexed Addressing

This indexed addressing mode uses a 9-bit signed offset which is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction. This gives a range of –256 through +255 from the value in the base index register. The most significant bit (sign bit) of the offset is included in the instruction postbyte and the remaining eight bits are provided as an extension byte after the instruction postbyte in the instruction flow.

Examples:

```
LDAA          $FF,X
LDAB          -20,Y
```

For these examples, assume X is $1000 and Y is $2000 before execution of these instructions.

NOTE: *These instructions do not alter the index registers so they will still be $1000 and $2000, respectively, after the instructions.*

The first instruction will load A with the value from address $10FF and the second instruction will load B with the value from address $1FEC.

This variation of the indexed addressing mode in the CPU12 is similar to the M68HC11 indexed addressing mode, but is functionally enhanced. The M68HC11 CPU provides for unsigned 8-bit constant offset indexing from X or Y, and use of Y requires an extra instruction byte and thus, an extra execution cycle. The 9-bit signed offset used in the CPU12 covers the same range of positive offsets as the M68HC11, and adds negative offset capability. The CPU12 can use X, Y, SP, or PC as the base index register.

3.9.3 16-Bit Constant Offset Indexed Addressing

This indexed addressing mode uses a 16-bit offset which is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location affected by the instruction. This allows access to any address in the 64-Kbyte address space. Since the address bus and the offset are both 16 bits, it does not matter whether the offset value is considered to be a signed or an unsigned value ($FFFF may be thought of as +65,535 or as –1). The 16-bit offset is provided as two extension bytes after the instruction postbyte in the instruction flow.

3.9.4 16-Bit Constant Indirect Indexed Addressing

This indexed addressing mode adds a 16-bit instruction-supplied offset to the base index register to form the address of a memory location that contains a pointer to the memory location affected by the instruction. The instruction itself does not point to the address of the memory location to be acted upon, but rather to the location of a pointer to the address to be acted on. The square brackets distinguish this addressing mode from 16-bit constant offset indexing.

Example:

```
LDAA          [10,X]
```

In this example, X holds the base address of a table of pointers. Assume that X has an initial value of $1000, and that the value $2000 is stored at addresses $100A and $100B. The instruction first adds the value 10 to the value in X to form the address $100A. Next, an address pointer ($2000) is fetched from memory at $100A. Then, the value stored in location $2000 is read and loaded into the A accumulator.

3.9.5 Auto Pre/Post Decrement/Increment Indexed Addressing

This indexed addressing mode provides four ways to automatically change the value in a base index register as a part of instruction execution. The index register can be incremented or decremented by an integer value either before or after indexing takes place. The base index register may be X, Y, or SP. (Auto-modify modes would not make sense on PC.)

Pre-decrement and pre-increment versions of the addressing mode adjust the value of the index register before accessing the memory location affected by the instruction — the index register retains the changed value after the instruction executes. Post-decrement and post-increment versions of the addressing mode use the initial value in the index register to access the memory location affected by the instruction, then change the value of the index register.

The CPU12 allows the index register to be incremented or decremented by any integer value in the ranges –8 through –1 or 1 through 8. The value need not be related to the size of the operand for the current instruction. These instructions can be used to incorporate an index adjustment into an existing instruction rather than using an additional instruction and increasing execution time. This addressing mode is also used to perform operations on a series of data structures in memory.

When an LEAS, LEAX, or LEAY instruction is executed using this addressing mode, and the operation modifies the index register that is being loaded, the final value in the register is the value that would have been used to access a memory operand. (Premodification is seen in the result but postmodification is not.)

Examples:

```
    STAA    1,-SP      ;equivalent to PSHA
    STX     2,-SP      ;equivalent to PSHX
    LDX     2,SP+      ;equivalent to PULX
    LDAA    1,SP+      ;equivalent to PULA
```

For a “last-used” type of stack like the CPU12 stack, these four examples are equivalent to common push and pull instructions.

For a “next-available” stack like the M68HC11 stack, push A onto stack (PSHA) is equivalent to store accumulator A (STAA) 1,SP– and pull A from stack (PULA) is equivalent to load accumulator A (LDAA) 1,+SP. However, in the M68HC11, 16-bit operations like push register X onto stack (PSHX) and pull register X from stack (PULX) require multiple instructions to decrement the SP by one, then store X, then decrement SP by one again.

In the STAA 1,–SP example, the stack pointer is pre-decremented by one and then A is stored to the address contained in the stack pointer. Similarly the LDX 2,SP+ first loads X from the address in the stack pointer, then post-increments SP by two.

Example:

```
MOVW        2,X+,4,+Y
```

This example demonstrates how to work with data structures larger than bytes and words. With this instruction in a program loop, it is possible to move words of data from a list having one word per entry into a second table that has four bytes per table element. In this example the source pointer is updated after the data is read from memory (post-increment) while the destination pointer is updated before it is used to access memory (pre-increment).

3.9.6 Accumulator Offset Indexed Addressing

In this indexed addressing mode, the effective address is the sum of the values in the base index register and an unsigned offset in one of the accumulators. The value in the index register itself is not changed. The index register can be X, Y, SP, or PC and the accumulator can be either of the 8-bit accumulators (A or B) or the 16-bit D accumulator.

Example:

```
LDAA        B,X
```

This instruction internally adds B to X to form the address from which A will be loaded. B and X are not changed by this instruction. This example is similar to the following 2-instruction combination in an M68HC11.

Examples:

```
ABX
LDAA        0,X
```

However, this 2-instruction sequence alters the index register. If this sequence was part of a loop where B changed on each pass, the index register would have to be reloaded with the reference value on each loop pass. The use of LDAA B,X is more efficient in the CPU12.

3.9.7 Accumulator D Indirect Indexed Addressing

This indexed addressing mode adds the value in the D accumulator to the value in the base index register to form the address of a memory location that contains a pointer to the memory location affected by the instruction. The instruction operand does not point to the address of the memory location to be acted upon, but rather to the location of a pointer to the address to be acted upon. The square brackets distinguish this addressing mode from D accumulator offset indexing.

Examples:

```
    JMP         [D,PC]
    GO1         DC.W              PLACE1
    GO2         DC.W              PLACE2
    GO3         DC.W              PLACE3
```

This example is a computed GOTO. The values beginning at GO1 are addresses of potential destinations of the jump (JMP) instruction. At the time the JMP [D,PC] instruction is executed, PC points to the address GO1, and D holds one of the values $0000, $0002, or $0004 (determined by the program some time before the JMP).

Assume that the value in D is $0002. The JMP instruction adds the values in D and PC to form the address of GO2. Next the CPU reads the address PLACE2 from memory at GO2 and jumps to PLACE2. The locations of PLACE1 through PLACE3 were known at the time of program assembly but the destination of the JMP depends upon the value in D computed during program execution.

3.10 Instructions Using Multiple Modes

Several CPU12 instructions use more than one addressing mode in the course of execution.

3.10.1 Move Instructions

Move instructions use separate addressing modes to access the source and destination of a move. There are move variations for all practical combinations of immediate, extended, and indexed addressing modes.

The only combinations of addressing modes that are not allowed are those with an immediate mode destination (the operand of an immediate mode instruction is data, not an address). For indexed moves, the reference index register may be X, Y, SP, or PC.

Move instructions do not support indirect modes, 9-bit, or 16-bit offset modes requiring extra extension bytes. There are special considerations when using PC-relative addressing with move instructions. The original M68HC12 implemented the instruction queue slightly differently than the newer HCS12. In the older M68HC12 implementation, the CPU did not maintain a pointer to the start of the instruction after the current instruction (what the user thinks of as the PC value during execution). This caused an offset for PC-relative move instructions.

PC-relative addressing uses the address of the location immediately following the last byte of object code for the current instruction as a reference point. The CPU12 normally corrects for queue offset and for instruction alignment so that queue operation is transparent to the user. However, in the original M68HC12, move instructions pose three special problems:

- Some moves use an indexed source and an indexed destination.
- Some moves have object code that is too long to fit in the queue all at one time, so the PC value changes during execution.
- All moves do not have the indexed postbyte as the last byte of object code.

These cases are not handled by automatic queue pointer maintenance, but it is still possible to use PC-relative indexing with move instructions by providing for PC offsets in source code.

Table 3-3 shows PC offsets from the location immediately following the current instruction by addressing mode.

Table 3-3. PC Offsets for MOVE Instructions (M68HC12 Only)

MOVE Instruction	Addressing Modes	Offset Value
MOVB	IMM ⇒ IDX	+1
	EXT ⇒ IDX	+2
	IDX ⇒ EXT	–2
	IDX ⇒ IDX	–1 for first operand +1 for second operand
MOVW	IMM ⇒ IDX	+2
	EXT ⇒ IDX	+2
	IDX ⇒ EXT	–2
	IDX ⇒ IDX	–1 for first operand +1 for second operand

Example:

```
    1000   18 09 C2 20 00    MOVB  $2000 2,PC
```

Moves a byte of data from $2000 to $1009

The expected location of the PC = $1005. The offset = +2.
[1005 + 2 (for 2,PC) + 2 (for correction) = 1009]

$18 is the page pre-byte, 09 is the MOVB opcode for ext-idx, C2 is the indexed postbyte for 2,PC (without correction).

The Freescale MCUasm assembler produces corrected object code for PC-relative moves (18 09 C0 20 00 for the example shown).

NOTE: *Instead of assembling the 2,PC as C2, the correction has been applied to make it C0. Check whether an assembler makes the correction before using PC-relative moves.*

On the newer HCS12, the instruction queue was implemented such that an internal pointer, to the start of the next instruction, is always available. On the HCS12, PC-relative move instructions work as expected without any offset adjustment. Although this is different from the original M68HC12, it is unlikely to be a problem because PC-relative indexing is rarely, if ever, used with move instructions.

3.10.2 Bit Manipulation Instructions

Bit manipulation instructions use either a combination of two or a combination of three addressing modes.

The clear bits in memory (BCLR) and set bits in memory (BSET) instructions use an 8-bit mask to determine which bits in a memory byte are to be changed. The mask must be supplied with the instruction as an immediate mode value. The memory location to be modified can be specified by means of direct, extended, or indexed addressing modes.

The branch if bits cleared (BRCLR) and branch if bits set (BRSET) instructions use an 8-bit mask to test the states of bits in a memory byte. The mask is supplied with the instruction as an immediate mode value. The memory location to be tested is specified by means of direct, extended, or indexed addressing modes. Relative addressing mode is used to determine the branch address. A signed 8-bit offset must be supplied with the instruction.

3.11 Addressing More than 64 Kbytes

Some M68HC12 devices incorporate hardware that supports addressing a larger memory space than the standard 64 Kbytes. The expanded memory system uses fast on-chip logic to implement a transparent bank-switching scheme.

Increased code efficiency is the greatest advantage of using a switching scheme instead of a large linear address space. In systems with large linear address spaces, instructions require more bits of information to address a memory location, and CPU overhead is greater. Other advantages include the ability to change the size of system memory and the ability to use various types of external memory.

However, the add-on bank switching schemes used in other microcontrollers have known weaknesses. These include the cost of external glue logic, increased programming overhead to change banks, and the need to disable interrupts while banks are switched.

The M68HC12 system requires no external glue logic. Bank switching overhead is reduced by implementing control logic in the MCU. Interrupts do not need to be disabled during switching because switching tasks are incorporated in special instructions that greatly simplify program access to extended memory.

MCUs with expanded memory treat the 16 Kbytes of memory space from $8000 to $BFFF as a program memory window. Expanded-memory architecture includes an 8-bit program page register (PPAGE), which allows up to 256 16-Kbyte program memory pages to be switched into and out of the program memory window. This provides for up to 4 Megabytes of paged program memory.

The CPU12 instruction set includes call subroutine in expanded memory (CALL) and return from call (RTC) instructions, which greatly simplify the use of expanded memory space. These instructions also execute correctly on devices that do not have expanded-memory addressing capability, thus providing for portable code.

The CALL instruction is similar to the jump-to-subroutine (JSR) instruction. When CALL is executed, the current value in PPAGE is pushed onto the stack with a return address, and a new instruction-supplied value is written to PPAGE. This value selects the page the called subroutine resides upon and can be considered part of the effective address. For all addressing mode variations except indexed indirect modes, the new page value is provided by an immediate operand in the instruction. For indexed indirect variations of CALL, a pointer specifies memory locations where the new page value and the address of the called subroutine are stored. Use of indirect addressing for both the page value and the address within the page frees the program from keeping track of explicit values for either address.

The RTC instruction restores the saved program page value and the return address from the stack. This causes execution to resume at the next instruction after the original CALL instruction.

Section 4. Instruction Queue

4.1 Introduction

The CPU12 uses an instruction queue to increase execution speed. This section describes queue operation during normal program execution and changes in execution flow. These concepts augment the descriptions of instructions and cycle-by-cycle instruction execution in subsequent sections, but it is important to note that queue operation is automatic, and generally transparent to the user.

The material in this section is general. **Section 6. Instruction Glossary** contains detailed information concerning cycle-by-cycle execution of each instruction. **Section 8. Instruction Queue** contains detailed information about tracking queue operation and instruction execution.

4.2 Queue Description

The fetching mechanism in the CPU12 is best described as a queue rather than as a pipeline. Queue logic fetches program information and positions it for execution, but instructions are executed sequentially. A typical pipelined central processor unit (CPU) can execute more than one instruction at the same time, but interactions between the prefetch and execution mechanisms can make tracking and debugging difficult. The CPU12 thus gains the advantages of independent fetches, yet maintains a straightforward relationship between bus and execution cycles.

Each instruction refills the queue by fetching the same number of bytes that the instruction uses. Program information is fetched in aligned 16-bit words. Each program fetch (`P`) indicates that two bytes need to be replaced in the instruction queue. Each optional fetch (`O`) indicates that only one byte needs to be replaced. For example, an instruction composed of five bytes does two program fetches and one optional fetch. If the first byte of the five-byte instruction was even-aligned, the optional fetch is converted into a free cycle. If the first byte was odd-aligned, the optional fetch is executed as a program fetch.

Two external pins, IPIPE[1:0], provide time-multiplexed information about data movement in the queue and instruction execution. Decoding and use of these signals is discussed in **Section 8. Instruction Queue**.

4.2.1 Original M68HC12 Queue Implementation

There are two 16-bit queue stages and one 16-bit buffer. Program information is fetched in aligned 16-bit words. Unless buffering is required, program information is first queued into stage 1, then advanced to stage 2 for execution.

At least two words of program information are available to the CPU when execution begins. The first byte of object code is in either the even or odd half of the word in stage 2, and at least two more bytes of object code are in the queue.

The buffer is used when a program word arrives before the queue can advance. This occurs during execution of single-byte and odd-aligned instructions. For instance, the queue cannot advance after an aligned, single-byte instruction is executed, because the first byte of the next instruction is also in stage 2. In these cases, information is latched into the buffer until the queue can advance.

4.2.2 HCS12 Queue Implementation

There are three 16-bit stages in the instruction queue. Instructions enter the queue at stage 1 and shift out of stage 3 as the CPU executes instructions and fetches new ones into stage 1. Each byte in the queue is selectable. An opcode prediction algorithm determines the location of the next opcode in the instruction queue.

4.3 Data Movement in the Queue

All queue operations are combinations of four basic queue movement cycles. Descriptions of each of these cycles follows. Queue movement cycles are only one factor in instruction execution time and should not be confused with bus cycles.

4.3.1 No Movement

There is no data movement in the instruction queue during the cycle. This occurs during execution of instructions that must perform a number of internal operations, such as division instructions.

4.3.2 Latch Data from Bus (Applies Only to the M68HC12 Queue Implementation)

All instructions initiate fetches to refill the queue as execution proceeds. However, a number of conditions, including instruction alignment and the length of previous instructions, affect when the queue advances. If the queue is not ready to advance when fetched information arrives, the information is latched into the buffer. Later, when the queue does advance, stage 1 is refilled from the buffer. If more than one latch cycle occurs before the queue advances, the buffer is filled on the first latch event and subsequent latch events are ignored until the queue advances.

4.3.3 Advance and Load from Data Bus

The content of queue is advanced by one stage, and stage 1 is loaded with a word of program information from the data bus. The information was requested two bus cycles earlier but has only become available this cycle, due to access delay.

4.3.4 Advance and Load from Buffer (Applies Only to M68HC12 Queue Implementation)

The content of queue stage 1 advances to stage 2, and stage 1 is loaded with a word of program information from the buffer. The information in the buffer was latched from the data bus during a previous cycle because the queue was not ready to advance when it arrived.

4.4 Changes in Execution Flow

During normal instruction execution, queue operations proceed as a continuous sequence of queue movement cycles. However, situations arise which call for changes in flow. These changes are categorized as resets, interrupts, subroutine calls, conditional branches, and jumps. Generally speaking, resets and interrupts are considered to be related to events outside the current program context that require special processing, while subroutine calls, branches, and jumps are considered to be elements of program structure.

During design, great care is taken to assure that the mechanism that increases instruction throughput during normal program execution does not cause bottlenecks during changes of program flow, but internal queue operation is largely transparent to the user. The following information is provided to enhance subsequent descriptions of instruction execution.

4.4.1 Exceptions

Exceptions are events that require processing outside the normal flow of instruction execution. CPU12 exceptions include five types of exceptions:

- Reset (including COP, clock monitor, and pin)
- Unimplemented opcode trap
- Software interrupt instruction
- X-bit interrupts
- I-bit interrupts

All exceptions use the same microcode, but the CPU follows different execution paths for each type of exception.

CPU12 exception handling is designed to minimize the effect of queue operation on context switching. Thus, an exception vector fetch is the first part of exception processing, and fetches to refill the queue from the address pointed to by the vector are interleaved with the stacking operations that preserve context, so that program access time does not delay the switch. Refer to **Section 7. Exception Processing** for detailed information.

4.4.2 Subroutines

The CPU12 can branch to (BSR), jump to (JSR), or call (CALL) subroutines. BSR and JSR are used to access subroutines in the normal 64-Kbyte address space. The CALL instruction is intended for use in MCUs with expanded memory capability.

BSR uses relative addressing mode to generate the effective address of the subroutine, while JSR can use various other addressing modes. Both instructions calculate a return address, stack the address, then perform three program word fetches to refill the queue.

Subroutines in the normal 64-Kbyte address space are terminated with a return-from-subroutine (RTS) instruction. RTS unstacks the return address, then performs three program word fetches from that address to refill the queue.

CALL is similar to JSR. MCUs with expanded memory treat 16 Kbytes of addresses from $8000 to $BFFF as a memory window. An 8-bit PPAGE register switches memory pages into and out of the window. When CALL is executed, a return address is calculated, then it and the current PPAGE value are stacked, and a new instruction-supplied value is written to PPAGE. The subroutine address is calculated, then three program word fetches are made from that address to refill the instruction queue.

The return-from-call (RTC) instruction is used to terminate subroutines in expanded memory. RTC unstacks the PPAGE value and the return address, then performs three program word fetches from that address to refill the queue.

CALL and RTC execute correctly in the normal 64-Kbyte address space, thus providing for portable code. However, since extra execution cycles are required, routinely substituting CALL/RTC for JSR/RTS is not recommended.

4.4.3 Branches

Branch instructions cause execution flow to change when specific pre-conditions exist. The CPU12 instruction set includes:

- Short conditional branches
- Long conditional branches
- Bit-condition branches

Types and conditions of branch instructions are described in **5.19 Branch Instructions**. All branch instructions affect the queue similarly, but there are differences in overall cycle counts between the various types. Loop primitive instructions are a special type of branch instruction used to implement counter-based loops.

Branch instructions have two execution cases:

- The branch condition is satisfied, and a change of flow takes place.
- The branch condition is not satisfied, and no change of flow occurs.

4.4.3.1 Short Branches

The “not-taken” case for short branches is simple. Since the instruction consists of a single word containing both an opcode and an 8-bit offset, the queue advances, another program word is fetched, and execution continues with the next instruction.

The “taken” case for short branches requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is calculated using the relative offset in the instruction. Then, the address is loaded into the program counter, and the CPU performs three program word fetches at the new address to refill the instruction queue.

4.4.3.2 Long Branches

The “not-taken” case for all long branches requires three cycles, while the “taken” case requires four cycles. This is due to differences in the amount of program information needed to fill the queue.

Long branch instructions begin with a $18 prebyte which indicates that the opcode is on page 2 of the opcode map. The CPU12 treats the prebyte as a special one-byte instruction. If the prebyte is not aligned, the first cycle is used to perform a program word access; if the prebyte is aligned, the first cycle is used to perform a free cycle. The first cycle for the prebyte is executed whether or not the branch is taken.

The first cycle of the branch instruction is an optional cycle. Optional cycles make the effects of byte-sized and misaligned instructions consistent with those of aligned word-length instructions. Program information is always fetched as aligned 16-bit words. When an instruction has an odd number of bytes, and the first byte is not aligned with an even byte boundary, the optional cycle makes an additional program word access that maintains queue order. In all other cases, the optional cycle is a free cycle.

In the “not-taken” case, the queue must advance so that execution can continue with the next instruction. Two cycles are used to refill the queue. Alignment determines how the second of these cycles is used.

In the “taken” case, the effective address of the branch is calculated using the 16-bit relative offset contained in the second word of the instruction. This address is loaded into the program counter, then the CPU performs three program word fetches at the new address.

4.4.3.3 Bit Condition Branches

Bit condition branch instructions read a location in memory, and branch if the bits in that location are in a certain state. These instructions can use direct, extended, or indexed addressing modes. Indexed operations require varying amounts of information to determine the effective address, so instruction length varies according to the mode used, which in turn affects the amount of program information fetched. To shorten execution time, these branches perform one program word fetch in anticipation of the “taken” case. The data from this fetch is ignored in the “not-taken” case. If the branch is taken, the CPU fetches three program word fetches at the new address to fill the instruction queue.

4.4.3.4 Loop Primitives

The loop primitive instructions test a counter value in a register or accumulator and branch to an address specified by a 9-bit relative offset contained in the instruction if a specified condition is met. There are auto-increment and auto-decrement versions of these instructions. The test and increment/decrement operations are performed on internal CPU registers, and require no additional program information. To shorten execution time, these branches perform one program word fetch in anticipation of the “taken” case. The data from this fetch is ignored if the branch is not taken, and the CPU does one program fetch and one optional fetch to refill the queue[1]. If the branch is taken, the CPU finishes refilling the queue with two additional program word fetches at the new address.

4.4.4 Jumps

Jump (JMP) is the simplest change of flow instruction. JMP can use extended or indexed addressing. Indexed operations require varying amounts of information to determine the effective address, so instruction length varies according to the mode used, which in turn affects the amount of program information fetched. All forms of JMP perform three program word fetches at the new address to refill the instruction queue.

[1] In the original M68HC12, the implementation of these two cycles are both program word fetches.

Section 5. Instruction Set Overview

5.1 Introduction

This section contains general information about the central processor unit (CPU12) instruction set. It is organized into instruction categories grouped by function.

5.2 Instruction Set Description

CPU12 instructions are a superset of the M68HC11 instruction set. Code written for an M68HC11 can be reassembled and run on a CPU12 with no changes. The CPU12 provides expanded functionality and increased code efficiency. There are two implementations of the CPU12, the original M68HC12 and the newer HCS12. Both implementations have the same instruction set, although there are small differences in cycle-by-cycle access details (the order of some bus cycles changed to accommodate differences in the way the instruction queue was implemented). These minor differences are transparent for most users.

In the M68HC12 and HCS12 architecture, all memory and input/output (I/O) are mapped in a common 64-Kbyte address space (memory-mapped I/O). This allows the same set of instructions to be used to access memory, I/O, and control registers. General-purpose load, store, transfer, exchange, and move instructions facilitate movement of data to and from memory and peripherals.

The CPU12 has a full set of 8-bit and 16-bit mathematical instructions. There are instructions for signed and unsigned arithmetic, division, and multiplication with 8-bit, 16-bit, and some larger operands.

Special arithmetic and logic instructions aid stacking operations, indexing, binary-coded decimal (BCD) calculation, and condition code register manipulation. There are also dedicated instructions for multiply and accumulate operations, table interpolation, and specialized fuzzy logic operations that involve mathematical calculations.

Refer to **Section 6. Instruction Glossary** for detailed information about individual instructions. **Appendix A. Instruction Reference** contains quick-reference material, including an opcode map and postbyte encoding for indexed addressing, transfer/exchange instructions, and loop primitive instructions.

5.3 Load and Store Instructions

Load instructions copy memory content into an accumulator or register. Memory content is not changed by the operation. Load instructions (but not LEA_ instructions) affect condition code bits so no separate test instructions are needed to check the loaded values for negative or 0 conditions.

Store instructions copy the content of a CPU register to memory. Register/accumulator content is not changed by the operation. Store instructions automatically update the N and Z condition code bits, which can eliminate the need for a separate test instruction in some programs.

Table 5-1 is a summary of load and store instructions.

Table 5-1. Load and Store Instructions

Mnemonic	Function	Operation
Load Instructions		
LDAA	Load A	$(M) \Rightarrow A$
LDAB	Load B	$(M) \Rightarrow B$
LDD	Load D	$(M : M + 1) \Rightarrow (A:B)$
LDS	Load SP	$(M : M + 1) \Rightarrow SP_H:SP_L$
LDX	Load index register X	$(M : M + 1) \Rightarrow X_H:X_L$
LDY	Load index register Y	$(M : M + 1) \Rightarrow Y_H:Y_L$
LEAS	Load effective address into SP	Effective address ⇒ SP
LEAX	Load effective address into X	Effective address ⇒ X
LEAY	Load effective address into Y	Effective address ⇒ Y

Continued on next page

Table 5-1. Load and Store Instructions (Continued)

Store Instructions		
STAA	Store A	$(A) \Rightarrow M$
STAB	Store B	$(B) \Rightarrow M$
STD	Store D	$(A) \Rightarrow M, (B) \Rightarrow M + 1$
STS	Store SP	$(SP_H:SP_L) \Rightarrow M : M + 1$
STX	Store X	$(X_H:X_L) \Rightarrow M : M + 1$
STY	Store Y	$(Y_H:Y_L) \Rightarrow M : M + 1$

5.4 Transfer and Exchange Instructions

Transfer instructions copy the content of a register or accumulator into another register or accumulator. Source content is not changed by the operation. Transfer register to register (TFR) is a universal transfer instruction, but other mnemonics are accepted for compatibility with the M68HC11. The transfer A to B (TAB) and transfer B to A (TBA) instructions affect the N, Z, and V condition code bits in the same way as M68HC11 instructions. The TFR instruction does not affect the condition code bits.

The sign extend 8-bit operand (SEX) instruction is a special case of the universal transfer instruction that is used to sign extend 8-bit two’s complement numbers so that they can be used in 16-bit operations. The 8-bit number is copied from accumulator A, accumulator B, or the condition code register to accumulator D, the X index register, the Y index register, or the stack pointer. All the bits in the upper byte of the 16-bit result are given the value of the most-significant bit (MSB) of the 8-bit number.

Exchange instructions exchange the contents of pairs of registers or accumulators. When the first operand in an EXG instruction is 8-bits and the second operand is 16 bits, a zero-extend operation is performed on the 8-bit register as it is copied into the 16-bit register.

Section 6. Instruction Glossary contains information concerning other transfers and exchanges between 8- and 16-bit registers.

Table 5-2 is a summary of transfer and exchange instructions.

Table 5-2. Transfer and Exchange Instructions

Mnemonic	Function	Operation
Transfer Instructions		
TAB	Transfer A to B	(A) ⇒ B
TAP	Transfer A to CCR	(A) ⇒ CCR
TBA	Transfer B to A	(B) ⇒ A
TFR	Transfer register to register	(A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP
TPA	Transfer CCR to A	(CCR) ⇒ A
TSX	Transfer SP to X	(SP) ⇒ X
TSY	Transfer SP to Y	(SP) ⇒ Y
TXS	Transfer X to SP	(X) ⇒ SP
TYS	Transfer Y to SP	(Y) ⇒ SP
Exchange Instructions		
EXG	Exchange register to register	(A, B, CCR, D, X, Y, or SP) ⇔ (A, B, CCR, D, X, Y, or SP)
XGDX	Exchange D with X	(D) ⇔ (X)
XGDY	Exchange D with Y	(D) ⇔ (Y)
Sign Extension Instruction		
SEX	Sign extend 8-Bit operand	Sign-extended (A, B, or CCR) ⇒ D, X, Y, or SP

5.5 Move Instructions

Move instructions move (copy) data bytes or words from a source (M_1 or M : M +1_1) to a destination (M_2 or M : M +1_2) in memory. Six combinations of immediate, extended, and indexed addressing are allowed to specify source and destination addresses (IMM ⇒ EXT, IMM ⇒ IDX, EXT ⇒ EXT, EXT ⇒ IDX, IDX ⇒ EXT, IDX ⇒ IDX). Addressing mode combinations with immediate for the destination would not be useful.

Table 5-3 shows byte and word move instructions.

Table 5-3. Move Instructions

Mnemonic	Function	Operation
MOVB	Move byte (8-bit)	$(M_1) \Rightarrow M_2$
MOVW	Move word (16-bit)	$(M : M + 1_1) \Rightarrow M : M + 1_2$

5.6 Addition and Subtraction Instructions

Signed and unsigned 8- and 16-bit addition can be performed between registers or between registers and memory. Special instructions support index calculation. Instructions that add the carry bit in the condition code register (CCR) facilitate multiple precision computation.

Signed and unsigned 8- and 16-bit subtraction can be performed between registers or between registers and memory. Special instructions support index calculation. Instructions that subtract the carry bit in the CCR facilitate multiple precision computation. Refer to **Table 5-4** for addition and subtraction instructions.

Load effective address (LEAS, LEAX, and LEAY) instructions could also be considered as specialized addition and subtraction instructions. See **5.25 Pointer and Index Calculation Instructions** for more information.

Table 5-4. Addition and Subtraction Instructions

Mnemonic	Function	Operation
Addition Instructions		
ABA	Add B to A	(A) + (B) ⇒ A
ABX	Add B to X	(B) + (X) ⇒ X
ABY	Add B to Y	(B) + (Y) ⇒ Y
ADCA	Add with carry to A	(A) + (M) + C ⇒ A
ADCB	Add with carry to B	(B) + (M) + C ⇒ B
ADDA	Add without carry to A	(A) + (M) ⇒ A
ADDB	Add without carry to B	(B) + (M) ⇒ B
ADDD	Add to D	(A:B) + (M : M + 1) ⇒ A : B
Subtraction Instructions		
SBA	Subtract B from A	(A) – (B) ⇒ A
SBCA	Subtract with borrow from A	(A) – (M) – C ⇒ A
SBCB	Subtract with borrow from B	(B) – (M) – C ⇒ B
SUBA	Subtract memory from A	(A) – (M) ⇒ A
SUBB	Subtract memory from B	(B) – (M) ⇒ B
SUBD	Subtract memory from D (A:B)	(D) – (M : M + 1) ⇒ D

5.7 Binary-Coded Decimal Instructions

To add binary-coded decimal (BCD) operands, use addition instructions that set the half-carry bit in the CCR, then adjust the result with the decimal adjust A (DAA) instruction. **Table 5-5** is a summary of instructions that can be used to perform BCD operations.

Table 5-5. BCD Instructions

Mnemonic	Function	Operation
ABA	Add B to A	(A) + (B) ⇒ A
ADCA	Add with carry to A	(A) + (M) + C ⇒ A
ADCB[1]	Add with carry to B	(B) + (M) + C ⇒ B
ADDA[1]	Add memory to A	(A) + (M) ⇒ A
ADDB	Add memory to B	(B) + (M) ⇒ B
DAA	Decimal adjust A	$(A)_{10}$

[1] These instructions are not normally used for BCD operations because, although they affect H correctly, they do not leave the result in the correct accumulator (A) to be used with the DAA instruction. Thus additional steps would be needed to adjust the result to correct BCD form.

5.8 Decrement and Increment Instructions

The decrement and increment instructions are optimized 8- and 16-bit addition and subtraction operations. They are generally used to implement counters. Because they do not affect the carry bit in the CCR, they are particularly well suited for loop counters in multiple-precision computation routines. Refer to **5.20 Loop Primitive Instructions** for information concerning automatic counter branches. **Table 5-6** is a summary of decrement and increment instructions.

Table 5-6. Decrement and Increment Instructions

Mnemonic	Function	Operation
Decrement Instructions		
DEC	Decrement memory	(M) – \$01 ⇒ M
DECA	Decrement A	(A) – \$01 ⇒ A
DECB	Decrement B	(B) – \$01 ⇒ B
DES	Decrement SP	(SP) – \$0001 ⇒ SP
DEX	Decrement X	(X) – \$0001 ⇒ X
DEY	Decrement Y	(Y) – \$0001 ⇒ Y
Increment Instructions		
INC	Increment memory	(M) + \$01 ⇒ M
INCA	Increment A	(A) + \$01 ⇒ A
INCB	Increment B	(B) + \$01 ⇒ B
INS	Increment SP	(SP) + \$0001 ⇒ SP
INX	Increment X	(X) + \$0001 ⇒ X
INY	Increment Y	(Y) + \$0001 ⇒ Y

5.9 Compare and Test Instructions

Compare and test instructions perform subtraction between a pair of registers or between a register and memory. The result is not stored, but condition codes are set by the operation. These instructions are generally used to establish conditions for branch instructions. In this architecture, most instructions update condition code bits automatically, so it is often unnecessary to include separate test or compare instructions. **Table 5-7** is a summary of compare and test instructions.

Table 5-7. Compare and Test Instructions

Mnemonic	Function	Operation
Compare Instructions		
CBA	Compare A to B	(A) – (B)
CMPA	Compare A to memory	(A) – (M)
CMPB	Compare B to memory	(B) – (M)
CPD	Compare D to memory (16-bit)	(A : B) – (M : M + 1)
CPS	Compare SP to memory (16-bit)	(SP) – (M : M + 1)
CPX	Compare X to memory (16-bit)	(X) – (M : M + 1)
CPY	Compare Y to memory (16-bit)	(Y) – (M : M + 1)
Test Instructions		
TST	Test memory for zero or minus	(M) – $00
TSTA	Test A for zero or minus	(A) – $00
TSTB	Test B for zero or minus	(B) – $00

5.10 Boolean Logic Instructions

The Boolean logic instructions perform a logic operation between an 8-bit accumulator or the CCR and a memory value. AND, OR, and exclusive OR functions are supported. Table 5-8 summarizes logic instructions.

Table 5-8. Boolean Logic Instructions

Mnemonic	Function	Operation
ANDA	AND A with memory	(A) • (M) ⇒ A
ANDB	AND B with memory	(B) • (M) ⇒ B
ANDCC	AND CCR with memory (clear CCR bits)	(CCR) • (M) ⇒ CCR
EORA	Exclusive OR A with memory	(A) ⊕ (M) ⇒ A
EORB	Exclusive OR B with memory	(B) ⊕ (M) ⇒ B
ORAA	OR A with memory	(A) + (M) ⇒ A
ORAB	OR B with memory	(B) + (M) ⇒ B
ORCC	OR CCR with memory (set CCR bits)	(CCR) + (M) ⇒ CCR

5.11 Clear, Complement, and Negate Instructions

Each of the clear, complement, and negate instructions performs a specific binary operation on a value in an accumulator or in memory. Clear operations clear the value to 0, complement operations replace the value with its one’s complement, and negate operations replace the value with its two’s complement. Table 5-9 is a summary of clear, complement, and negate instructions.

Table 5-9. Clear, Complement, and Negate Instructions

Mnemonic	Function	Operation
CLC	Clear C bit in CCR	0 ⇒ C
CLI	Clear I bit in CCR	0 ⇒ I
CLR	Clear memory	$00 ⇒ M
CLRA	Clear A	$00 ⇒ A
CLRB	Clear B	$00 ⇒ B
CLV	Clear V bit in CCR	0 ⇒ V
COM	One’s complement memory	$\$FF - (M) \Rightarrow M \text{ or } (\overline{M}) \Rightarrow M$
COMA	One’s complement A	$\$FF - (A) \Rightarrow A \text{ or } (\overline{A}) \Rightarrow A$
COMB	One’s complement B	$\$FF - (B) \Rightarrow B \text{ or } (\overline{B}) \Rightarrow B$
NEG	Two’s complement memory	$\$00 - (M) \Rightarrow M \text{ or } (\overline{M}) + 1 \Rightarrow M$
NEGA	Two’s complement A	$\$00 - (A) \Rightarrow A \text{ or } (\overline{A}) + 1 \Rightarrow A$
NEGB	Two’s complement B	$\$00 - (B) \Rightarrow B \text{ or } (\overline{B}) + 1 \Rightarrow B$

5.12 Multiplication and Division Instructions

There are instructions for signed and unsigned 8- and 16-bit multiplication. Eight-bit multiplication operations have a 16-bit product. Sixteen-bit multiplication operations have 32-bit products.

Integer and fractional division instructions have 16-bit dividend, divisor, quotient, and remainder. Extended division instructions use a 32-bit dividend and a 16-bit divisor to produce a 16-bit quotient and a 16-bit remainder.

Table 5-10 is a summary of multiplication and division instructions.

Table 5-10. Multiplication and Division Instructions

Mnemonic	Function	Operation
Multiplication Instructions		
EMUL	16 by 16 multiply (unsigned)	(D) × (Y) ⇒ Y : D
EMULS	16 by 16 multiply (signed)	(D) × (Y) ⇒ Y : D
MUL	8 by 8 multiply (unsigned)	(A) × (B) ⇒ A : B
Division Instructions		
EDIV	32 by 16 divide (unsigned)	(Y : D) ÷ (X) ⇒ Y Remainder ⇒ D
EDIVS	32 by 16 divide (signed)	(Y : D) ÷ (X) ⇒ Y Remainder ⇒ D
FDIV	16 by 16 fractional divide	(D) ÷ (X) ⇒ X Remainder ⇒ D
IDIV	16 by 16 integer divide (unsigned)	(D) ÷ (X) ⇒ X Remainder ⇒ D
IDIVS	16 by 16 integer divide (signed)	(D) ÷ (X) ⇒ X Remainder ⇒ D

5.13 Bit Test and Manipulation Instructions

The bit test and manipulation operations use a mask value to test or change the value of individual bits in an accumulator or in memory. Bit test A (BITA) and bit test B (BITB) provide a convenient means of testing bits without altering the value of either operand. **Table 5-11** is a summary of bit test and manipulation instructions.

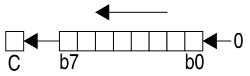
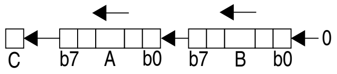
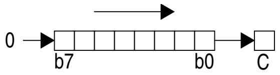
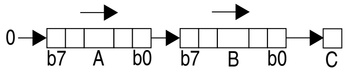
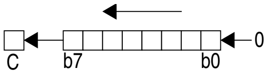
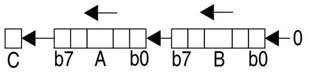
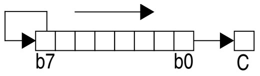
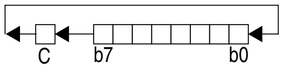
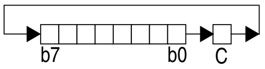
Table 5-11. Bit Test and Manipulation Instructions

Mnemonic	Function	Operation
BCLR	Clear bits in memory	$(M) \bullet (\overline{mm}) \Rightarrow M$
BITA	Bit test A	$(A) \bullet (M)$
BITB	Bit test B	$(B) \bullet (M)$
BSET	Set bits in memory	$(M) + (mm) \Rightarrow M$

5.14 Shift and Rotate Instructions

There are shifts and rotates for all accumulators and for memory bytes. All pass the shifted-out bit through the C status bit to facilitate multiple-byte operations. Because logical and arithmetic left shifts are identical, there are no separate logical left shift operations. Logic shift left (LSL) mnemonics are assembled as arithmetic shift left memory (ASL) operations. **Table 5-12** shows shift and rotate instructions.

Table 5-12. Shift and Rotate Instructions

Mnemonic	Function	Operation
Logical Shifts		
LSL LSLA LSLB	Logic shift left memory Logic shift left A Logic shift left B	
LSLD	Logic shift left D	
LSR LSRA LSRB	Logic shift right memory Logic shift right A Logic shift right B	
LSRD	Logic shift right D	
Arithmetic Shifts		
ASL ASLA ASLB	Arithmetic shift left memory Arithmetic shift left A Arithmetic shift left B	
ASLD	Arithmetic shift left D	
ASR ASRA ASRB	Arithmetic shift right memory Arithmetic shift right A Arithmetic shift right B	
Rotates		
ROL ROLA ROLB	Rotate left memory through carry Rotate left A through carry Rotate left B through carry	
ROR RORA RORB	Rotate right memory through carry Rotate right A through carry Rotate right B through carry	

5.15 Fuzzy Logic Instructions

The CPU12 instruction set includes instructions that support efficient processing of fuzzy logic operations. The descriptions of fuzzy logic instructions given here are functional overviews. **Table 5-13** summarizes the fuzzy logic instructions. Refer to **Section 9. Fuzzy Logic Support** for detailed discussion.

5.15.1 Fuzzy Logic Membership Instruction

The membership function (MEM) instruction is used during the fuzzification process. During fuzzification, current system input values are compared against stored input membership functions to determine the degree to which each label of each system input is true. This is accomplished by finding the y value for the current input on a trapezoidal membership function for each label of each system input. The MEM instruction performs this calculation for one label of one system input. To perform the complete fuzzification task for a system, several MEM instructions must be executed, usually in a program loop structure.

5.15.2 Fuzzy Logic Rule Evaluation Instructions

The MIN-MAX rule evaluation (REV and REVW) instructions perform MIN-MAX rule evaluations that are central elements of a fuzzy logic inference program. Fuzzy input values are processed using a list of rules from the knowledge base to produce a list of fuzzy outputs. The REV instruction treats all rules as equally important. The REVW instruction allows each rule to have a separate weighting factor. The two rule evaluation instructions also differ in the way rules are encoded into the knowledge base. Because they require a number of cycles to execute, rule evaluation instructions can be interrupted. Once the interrupt has been serviced, instruction execution resumes at the point the interrupt occurred.

5.15.3 Fuzzy Logic Weighted Average Instruction

The weighted average (WAV) instruction computes a sum-of-products and a sum-of-weights used for defuzzification. To be usable, the fuzzy outputs produced by rule evaluation must be defuzzified to produce a single output value which represents the combined effect of all of the fuzzy outputs. Fuzzy outputs correspond to the labels of a system output and each is defined by a membership function in the knowledge base. The CPU12 typically uses singletons for output membership functions rather than the trapezoidal shapes used for inputs. As with inputs, the x-axis represents the range of possible values for a system output. Singleton membership functions consist of the x-axis position for a label of the system output. Fuzzy outputs correspond to the y-axis height of the corresponding output membership function. The WAV instruction calculates the numerator and denominator sums for a weighted average of the fuzzy outputs. Because WAV requires a number of cycles to execute, it can be interrupted. The WAVR pseudo-instruction causes execution to resume at the point where it was interrupted.

Table 5-13. Fuzzy Logic Instructions

Mnemonic	Function	Operation
MEM	Membership function	μ (grade) $\Rightarrow M_{(Y)}$ $(X) + 4 \Rightarrow X$; $(Y) + 1 \Rightarrow Y$; A unchanged if $(A) < P1$ or $(A) > P2$, then $\mu = 0$, else $\mu = \text{MIN}\ [((A) - P1) \times S1, (P2 - (A)) \times S2, \$FF]$ where: A = current crisp input value X points to a 4-byte data structure that describes a trapezoidal membership function as base intercept points and slopes (P1, P2, S1, S2) Y points at fuzzy input (RAM location)

Continued on next page

Table 5-13. Fuzzy Logic Instructions (Continued)

Mnemonic	Function	Operation
REV	MIN-MAX rule evaluation	Find smallest rule input (MIN) Store to rule outputs unless fuzzy output is larger (MAX) Rules are unweighted Each rule input is an 8-bit offset from a base address in Y Each rule output is an 8-bit offset from a base address in Y $FE separates rule inputs from rule outputs $FF terminates the rule list REV can be interrupted
REVW	MIN-MAX rule evaluation	Find smallest rule input (MIN) Multiply by a rule weighting factor (optional) Store to rule outputs unless fuzzy output is larger (MAX) Each rule input is the 16-bit address of a fuzzy input Each rule output is the 16-bit address of a fuzzy output Address $FFFE separates rule inputs from rule outputs $FFFF terminates the rule list Weights are 8-bit values in a separate table REVW can be interrupted
WAV	Calculates numerator (sum of products) and denominator (sum of weights) for weighted average calculation Results are placed in correct registers for EDIV immediately after WAV	$\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ $\sum_{i=1}^{B} F_i \Rightarrow X$
WAVR	Resumes execution of interrupted WAV instruction	Recover immediate results from stack rather than initializing them to 0.

5.16 Maximum and Minimum Instructions

The maximum (MAX) and minimum (MIN) instructions are used to make comparisons between an accumulator and a memory location. These instructions can be used for linear programming operations, such as simplex-method optimization, or for fuzzification.

MAX and MIN instructions use accumulator A to perform 8-bit comparisons, while EMAX and EMIN instructions use accumulator D to perform 16-bit comparisons. The result (maximum or minimum value) can be stored in the accumulator (EMAXD, EMIND, MAXA, MINA) or the memory address (EMAXM, EMINM, MAXM, MINM).

Table 5-14 is a summary of minimum and maximum instructions.

Table 5-14. Minimum and Maximum Instructions

Mnemonic	Function	Operation
Minimum Instructions		
EMIND	MIN of two unsigned 16-bit values result to accumulator	MIN ((D), (M : M + 1)) ⇒ D
EMINM	MIN of two unsigned 16-bit values result to memory	MIN ((D), (M : M + 1)) ⇒ M : M+1
MINA	MIN of two unsigned 8-bit values result to accumulator	MIN ((A), (M)) ⇒ A
MINM	MIN of two unsigned 8-bit values result to memory	MIN ((A), (M)) ⇒ M
Maximum Instructions		
EMAXD	MAX of two unsigned 16-bit values result to accumulator	MAX ((D), (M : M + 1)) ⇒ D
EMAXM	MAX of two unsigned 16-bit values result to memory	MAX ((D), (M : M + 1)) ⇒ M : M + 1
MAXA	MAX of two unsigned 8-bit values result to accumulator	MAX ((A), (M)) ⇒ A
MAXM	MAX of two unsigned 8-bit values result to memory	MAX ((A), (M)) ⇒ M

5.17 Multiply and Accumulate Instruction

The multiply and accumulate (EMACS) instruction multiplies two 16-bit operands stored in memory and accumulates the 32-bit result in a third memory location. EMACS can be used to implement simple digital filters and defuzzification routines that use 16-bit operands. The WAV instruction incorporates an 8- to 16-bit multiply and accumulate operation that obtains a numerator for the weighted average calculation. The EMACS instruction can automate this portion of the averaging operation when 16-bit operands are used. **Table 5-15** shows the EMACS instruction.

Table 5-15. Multiply and Accumulate Instructions

Mnemonic	Function	Operation
EMACS	Multiply and accumulate (signed) 16 bit by 16 bit ⇒ 32 bit	$((M_{(X)}:M_{(X+1)}) \times (M_{(Y)}:M_{(Y+1)}))$ $+ (M \sim M + 3) \Rightarrow M \sim M + 3$

5.18 Table Interpolation Instructions

The table interpolation instructions (TBL and ETBL) interpolate values from tables stored in memory. Any function that can be represented as a series of linear equations can be represented by a table of appropriate size. Interpolation can be used for many purposes, including tabular fuzzy logic membership functions. TBL uses 8-bit table entries and returns an 8-bit result; ETBL uses 16-bit table entries and returns a 16-bit result. Use of indexed addressing mode provides great flexibility in structuring tables.

Consider each of the successive values stored in a table to be y-values for the endpoint of a line segment. The value in the B accumulator before instruction execution begins represents the change in x from the beginning of the line segment to the lookup point divided by total change in x from the beginning 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 smaller segments. During instruction execution, the change in y between the beginning and end of the segment (a signed byte for TBL or a signed word for ETBL) is multiplied by the content of the B accumulator to obtain an intermediate delta-y term. The result (stored in the A accumulator by TBL, and in the D accumulator by ETBL) is the y-value of the beginning point plus the signed intermediate delta-y value. **Table 5-16** shows the table interpolation instructions.

Table 5-16. Table Interpolation Instructions

Mnemonic	Function	Operation
ETBL	16-bit table lookup and interpolate (no indirect addressing modes allowed)	$(M : M + 1) + [(B) \times ((M + 2 : M + 3) - (M : M + 1))] \Rightarrow D$ Initialize B, and index before ETBL. <ea> points to the first table entry (M : M + 1) B is fractional part of lookup value
TBL	8-bit table lookup and interpolate (no indirect addressing modes allowed)	$(M) + [(B) \times ((M + 1) - (M))] \Rightarrow A$ Initialize B, and index before TBL. <ea> points to the first 8-bit table entry (M) B is fractional part of lookup value.

5.19 Branch Instructions

Branch instructions cause a sequence to change when specific conditions exist. The CPU12 uses three kinds of branch instructions. These are short branches, long branches, and bit condition branches.

Branch instructions can also be classified by the type of condition that must be satisfied in order for a branch to be taken. Some instructions belong to more than one classification. For example:

- Unary branch instructions always execute.
- Simple branches are taken when a specific bit in the condition code register is in a specific state as a result of a previous operation.
- Unsigned branches are taken when comparison or test of unsigned quantities results in a specific combination of condition code register bits.
- Signed branches are taken when comparison or test of signed quantities results in a specific combination of condition code register bits.

5.19.1 Short Branch Instructions

Short branch instructions operate this way: When a specified condition is met, a signed 8-bit offset is added to the value in the program counter. Program execution continues at the new address.

The numeric range of short branch offset values is \$80 (–128) to \$7F (127) from the address of the next memory location after the offset value.

Table 5-17 is a summary of the short branch instructions.

Table 5-17. Short Branch Instructions

Mnemonic	Function		Equation or Operation
Unary Branches			
BRA	Branch always		$1 = 1$
BRN	Branch never		$1 = 0$
Simple Branches			
BCC	Branch if carry clear		$C = 0$
BCS	Branch if carry set		$C = 1$
BEQ	Branch if equal		$Z = 1$
BMI	Branch if minus		$N = 1$
BNE	Branch if not equal		$Z = 0$
BPL	Branch if plus		$N = 0$
BVC	Branch if overflow clear		$V = 0$
BVS	Branch if overflow set		$V = 1$
Unsigned Branches			
		Relation	
BHI	Branch if higher	$R > M$	$C + Z = 0$
BHS	Branch if higher or same	$R \geq M$	$C = 0$
BLO	Branch if lower	$R < M$	$C = 1$
BLS	Branch if lower or same	$R \leq M$	$C + Z = 1$
Signed Branches			
BGE	Branch if greater than or equal	$R \geq M$	$N \oplus V = 0$
BGT	Branch if greater than	$R > M$	$Z + (N \oplus V) = 0$
BLE	Branch if less than or equal	$R \leq M$	$Z + (N \oplus V) = 1$
BLT	Branch if less than	$R < M$	$N \oplus V = 1$

5.19.2 Long Branch Instructions

Long branch instructions operate this way: When a specified condition is met, a signed 16-bit offset is added to the value in the program counter. Program execution continues at the new address. Long branches are used when large displacements between decision-making steps are necessary.

The numeric range of long branch offset values is $8000 (–32,768) to $7FFF (32,767) from the address of the next memory location after the offset value. This permits branching from any location in the standard 64-Kbyte address map to any other location in the 64-Kbyte map.

Table 5-18 is a summary of the long branch instructions.

Table 5-18. Long Branch Instructions

Mnemonic	Function	Equation or Operation
Unary Branches		
LBRA	Long branch always	$1 = 1$
LBRN	Long branch never	$1 = 0$
Simple Branches		
LBCC	Long branch if carry clear	$C = 0$
LBCS	Long branch if carry set	$C = 1$
LBEQ	Long branch if equal	$Z = 1$
LBMI	Long branch if minus	$N = 1$
LBNE	Long branch if not equal	$Z = 0$
LBPL	Long branch if plus	$N = 0$
LBVC	Long branch if overflow clear	$V = 0$
LBVS	Long branch if overflow set	$V = 1$
Unsigned Branches		
LBHI	Long branch if higher	$C + Z = 0$
LBHS	Long branch if higher or same	$C = 0$
LBLO	Long branch if lower	$Z = 1$
LBLS	Long branch if lower or same	$C + Z = 1$
Signed Branches		
LBGE	Long branch if greater than or equal	$N \oplus V = 0$
LBGT	Long branch if greater than	$Z + (N \oplus V) = 0$
LBLE	Long branch if less than or equal	$Z + (N \oplus V) = 1$
LBLT	Long branch if less than	$N \oplus V = 1$

5.19.3 Bit Condition Branch Instructions

The bit condition branches are taken when bits in a memory byte are in a specific state. A mask operand is used to test the location. If all bits in that location that correspond to ones in the mask are set (BRSET) or cleared (BRCLR), the branch is taken.

The numeric range of 8-bit offset values is $80 (–128) to $7F (127) from the address of the next memory location after the offset value.

Table 5-19 is a summary of bit condition branches.

Table 5-19. Bit Condition Branch Instructions

Mnemonic	Function	Equation or Operation
BRCLR	Branch if selected bits clear	$(M) \bullet (mm) = 0$
BRSET	Branch if selected bits set	$(\overline{M}) \bullet (mm) = 0$

5.20 Loop Primitive Instructions

The loop primitives can also be thought of as counter branches. The instructions test a counter value in a register or accumulator (A, B, D, X, Y, or SP) for zero or non-zero value as a branch condition. There are predecrement, preincrement, and test-only versions of these instructions.

The numeric range of 9-bit offset values is $100 (–256) to $0FF (255) from the address of the next memory location after the offset value.

Table 5-20 is a summary of loop primitive branches.

Table 5-20. Loop Primitive Instructions

Mnemonic	Function	Equation or Operation
DBEQ	Decrement counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	(counter) – 1⇒ counter If (counter) = 0, then branch; else continue to next instruction
DBNE	Decrement counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	(counter) – 1⇒ counter If (counter) not = 0, then branch; else continue to next instruction
IBEQ	Increment counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	(counter) + 1⇒ counter If (counter) = 0, then branch; else continue to next instruction
IBNE	Increment counter and branch if ≠ 0 (counter = A, B, D, X, Y, or SP)	(counter) + 1⇒ counter If (counter) not = 0, then branch; else continue to next instruction
TBEQ	Test counter and branch if = 0 (counter = A, B, D, X,Y, or SP)	If (counter) = 0, then branch; else continue to next instruction
TBNE	Test counter and branch if ≠ 0 (counter = A, B, D, X,Y, or SP)	If (counter) not = 0, then branch; else continue to next instruction

5.21 Jump and Subroutine Instructions

Jump (JMP) instructions cause immediate changes in sequence. The JMP instruction loads the PC with an address in the 64-Kbyte memory map, and program execution continues at that address. The address can be provided as an absolute 16-bit address or determined by various forms of indexed addressing.

Subroutine instructions optimize the process of transferring control to a code segment that performs a particular task. A short branch (BSR), a jump to subroutine (JSR), or an expanded-memory call (CALL) can be used to initiate subroutines. There is no LBSR instruction, but a PC-relative JSR performs the same function. A return address is stacked, then execution begins at the subroutine address. Subroutines in the normal 64-Kbyte address space are terminated with a return-from-subroutine (RTS) instruction. RTS unstacks the return address so that execution resumes with the instruction after BSR or JSR.

The call subroutine in expanded memory (CALL) instruction is intended for use with expanded memory. CALL stacks the value in the PPAGE register and the return address, then writes a new value to PPAGE to select the memory page where the subroutine resides. The page value is an immediate operand in all addressing modes except indexed indirect modes; in these modes, an operand points to locations in memory where the new page value and subroutine address are stored. The return from call (RTC) instruction is used to terminate subroutines in expanded memory. RTC unstacks the PPAGE value and the return address so that execution resumes with the next instruction after CALL. For software compatibility, CALL and RTC execute correctly on devices that do not have expanded addressing capability. **Table 5-21** summarizes the jump and subroutine instructions.

Table 5-21. Jump and Subroutine Instructions

Mnemonic	Function	Operation
BSR	Branch to subroutine	$SP - 2 \Rightarrow SP$ $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address ⇒ PC
CALL	Call subroutine in expanded memory	$SP - 2 \Rightarrow SP$ $RTN_H:RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP$ $(PPAGE) \Rightarrow M_{(SP)}$ Page ⇒ PPAGE Subroutine address ⇒ PC
JMP	Jump	Address ⇒ PC
JSR	Jump to subroutine	$SP - 2 \Rightarrow SP$ $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address ⇒ PC
RTC	Return from call	$M_{(SP)} \Rightarrow PPAGE$ $SP + 1 \Rightarrow SP$ $M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ $SP + 2 \Rightarrow SP$
RTS	Return from subroutine	$M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ $SP + 2 \Rightarrow SP$

5.22 Interrupt Instructions

Interrupt instructions handle transfer of control to a routine that performs a critical task. Software interrupts are a type of exception. **Section 7. Exception Processing** covers interrupt exception processing in detail.

The software interrupt (SWI) instruction initiates synchronous exception processing. First, the return PC value is stacked. After CPU context is stacked, execution continues at the address pointed to by the SWI vector.

Execution of the SWI instruction causes an interrupt without an interrupt service request. SWI is not inhibited by global mask bits I and X in the CCR, and execution of SWI sets the I mask bit. Once an SWI interrupt begins, maskable interrupts are inhibited until the I bit in the CCR is cleared. This typically occurs when a return from interrupt (RTI) instruction at the end of the SWI service routine restores context.

The CPU12 uses a variation of the software interrupt for unimplemented opcode trapping. There are opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the unimplemented opcodes on page 2, an opcode trap interrupt occurs. Traps are essentially interrupts that share the \$FFF8:\$FFF9 interrupt vector.

The RTI instruction is used to terminate all exception handlers, including interrupt service routines. RTI first restores the CCR, B:A, X, Y, and the return address from the stack. If no other interrupt is pending, normal execution resumes with the instruction following the last instruction that executed prior to interrupt.

Table 5-22 is a summary of interrupt instructions.

Table 5-22. Interrupt Instructions

Mnemonic	Function	Operation
RTI	Return from interrupt	$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$
SWI	Software interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$
TRAP	Unimplemented opcode interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$

5.23 Index Manipulation Instructions

The index manipulation instructions perform 8- and 16-bit operations on the three index registers and accumulators, other registers, or memory, as shown in **Table 5-23**.

Table 5-23. Index Manipulation Instructions

Mnemonic	Function	Operation
Addition Instructions		
ABX	Add B to X	(B) + (X) ⇒ X
ABY	Add B to Y	(B) + (Y) ⇒ Y
Compare Instructions		
CPS	Compare SP to memory	(SP) – (M : M + 1)
CPX	Compare X to memory	(X) – (M : M + 1)
CPY	Compare Y to memory	(Y) – (M : M + 1)
Load Instructions		
LDS	Load SP from memory	M : M+1 ⇒ SP
LDX	Load X from memory	(M : M + 1) ⇒ X
LDY	Load Y from memory	(M : M + 1) ⇒ Y
LEAS	Load effective address into SP	Effective address ⇒ SP
LEAX	Load effective address into X	Effective address ⇒ X
LEAY	Load effective address into Y	Effective address ⇒ Y
Store Instructions		
STS	Store SP in memory	(SP) ⇒ M:M+1
STX	Store X in memory	(X) ⇒ M : M + 1
STY	Store Y in memory	(Y) ⇒ M : M + 1
Transfer Instructions		
TFR	Transfer register to register	(A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP
TSX	Transfer SP to X	(SP) ⇒ X
TSY	Transfer SP to Y	(SP) ⇒ Y
TXS	transfer X to SP	(X) ⇒ SP
TYS	transfer Y to SP	(Y) ⇒ SP
Exchange Instructions		
EXG	Exchange register to register	(A, B, CCR, D, X, Y, or SP) ⇔ (A, B, CCR, D, X, Y, or SP)
XGDX	EXchange D with X	(D) ⇔ (X)
XGDY	EXchange D with Y	(D) ⇔ (Y)

5.24 Stacking Instructions

The two types of stacking instructions, are shown in **Table 5-24**. Stack pointer instructions use specialized forms of mathematical and data transfer instructions to perform stack pointer manipulation. Stack operation instructions save information on and retrieve information from the system stack.

Table 5-24. Stacking Instructions

Mnemonic	Function	Operation
Stack Pointer Instructions		
CPS	Compare SP to memory	(SP) – (M : M + 1)
DES	Decrement SP	(SP) – 1 ⇒ SP
INS	Increment SP	(SP) + 1 ⇒ SP
LDS	Load SP	(M : M + 1) ⇒ SP
LEAS	Load effective address into SP	Effective address ⇒ SP
STS	Store SP	(SP) ⇒ M : M + 1
TSX	Transfer SP to X	(SP) ⇒ X
TSY	Transfer SP to Y	(SP) ⇒ Y
TXS	Transfer X to SP	(X) ⇒ SP
TYS	Transfer Y to SP	(Y) ⇒ SP
Stack Operation Instructions		
PSHA	Push A	(SP) – 1 ⇒ SP; (A) ⇒ $M_{(SP)}$
PSHB	Push B	(SP) – 1 ⇒ SP; (B) ⇒ $M_{(SP)}$
PSHC	Push CCR	(SP) – 1 ⇒ SP; (A) ⇒ $M_{(SP)}$
PSHD	Push D	(SP) – 2 ⇒ SP; (A : B) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
PSHX	Push X	(SP) – 2 ⇒ SP; (X) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
PSHY	Push Y	(SP) – 2 ⇒ SP; (Y) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
PULA	Pull A	($M_{(SP)}$) ⇒ A; (SP) + 1 ⇒ SP
PULB	Pull B	($M_{(SP)}$) ⇒ B; (SP) + 1 ⇒ SP
PULC	Pull CCR	($M_{(SP)}$) ⇒ CCR; (SP) + 1 ⇒ SP
PULD	Pull D	($M_{(SP)}$: $M_{(SP+1)}$) ⇒ A : B; (SP) + 2 ⇒ SP
PULX	Pull X	($M_{(SP)}$: $M_{(SP+1)}$) ⇒ X; (SP) + 2 ⇒ SP
PULY	Pull Y	($M_{(SP)}$: $M_{(SP+1)}$) ⇒ Y; (SP) + 2 ⇒ SP

5.25 Pointer and Index Calculation Instructions

The load effective address instructions allow 5-, 8-, or 16-bit constants or the contents of 8-bit accumulators A and B or 16-bit accumulator D to be added to the contents of the X and Y index registers, or to the SP.

Table 5-25 is a summary of pointer and index instructions.

Table 5-25. Pointer and Index Calculation Instructions

Mnemonic	Function	Operation
LEAS	Load result of indexed addressing mode effective address calculation into stack pointer	r ± constant ⇒ SP or (r) + (accumulator) ⇒ SP r = X, Y, SP, or PC
LEAX	Load result of indexed addressing mode effective address calculation into x index register	r ± constant ⇒X or (r) + (accumulator) ⇒X r = X, Y, SP, or PC
LEAY	Load result of indexed addressing mode effective address calculation into y index register	r ± constant ⇒Y or (r) + (accumulator) ⇒ Y r = X, Y, SP, or PC

5.26 Condition Code Instructions

Condition code instructions are special forms of mathematical and data transfer instructions that can be used to change the condition code register. **Table 5-26** shows instructions that can be used to manipulate the CCR.

Table 5-26. Condition Code Instructions

Mnemonic	Function	Operation
ANDCC	Logical AND CCR with memory	$(CCR) \bullet (M) \Rightarrow CCR$
CLC	Clear C bit	$0 \Rightarrow C$
CLI	Clear I bit	$0 \Rightarrow I$
CLV	Clear V bit	$0 \Rightarrow V$
ORCC	Logical OR CCR with memory	$(CCR) + (M) \Rightarrow CCR$
PSHC	Push CCR onto stack	$(SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M_{(SP)}$
PULC	Pull CCR from stack	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$
SEC	Set C bit	$1 \Rightarrow C$
SEI	Set I bit	$1 \Rightarrow I$
SEV	Set V bit	$1 \Rightarrow V$
TAP	Transfer A to CCR	$(A) \Rightarrow CCR$
TPA	Transfer CCR to A	$(CCR) \Rightarrow A$

5.27 Stop and Wait Instructions

As shown in **Table 5-27**, two instructions put the CPU12 in an inactive state that reduces power consumption.

The stop instruction (STOP) stacks a return address and the contents of CPU registers and accumulators, then halts all system clocks.

The wait instruction (WAI) stacks a return address and the contents of CPU registers and accumulators, then waits for an interrupt service request; however, system clock signals continue to run.

Both STOP and WAI require that either an interrupt or a reset exception occur before normal execution of instructions resumes. Although both instructions require the same number of clock cycles to resume normal program execution after an interrupt service request is made, restarting after a STOP requires extra time for the oscillator to reach operating speed.

Table 5-27. Stop and Wait Instructions

Mnemonic	Function	Operation
STOP	Stop	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 CPU clocks
WAI	Wait for interrupt	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 ⇒ $M_{(SP)}$

5.28 Background Mode and Null Operations

Background debug mode (BDM) is a special CPU12 operating mode that is used for system development and debugging. Executing enter background debug mode (BGND) when BDM is enabled puts the CPU12 in this mode. For complete information, refer to **Section 8. Instruction Queue**.

Null operations are often used to replace other instructions during software debugging. Replacing conditional branch instructions with branch never (BRN), for instance, permits testing a decision-making routine by disabling the conditional branch without disturbing the offset value.

Null operations can also be used in software delay programs to consume execution time without disturbing the contents of other CPU registers or memory.

Table 5-28 shows the BGND and null operation (NOP) instructions.

Table 5-28. Background Mode and Null Operation Instructions

Mnemonic	Function	Operation
BGND	Enter background debug mode	If BDM enabled, enter BDM; else resume normal processing
BRN	Branch never	Does not branch
LBRN	Long branch never	Does not branch
NOP	Null operation	—

Section 6. Instruction Glossary

6.1 Introduction

This section is a comprehensive reference to the CPU12 instruction set.

6.2 Glossary Information

The glossary contains an entry for each assembler mnemonic, in alphabetic order. Figure 6-1 is a representation of a glossary page.

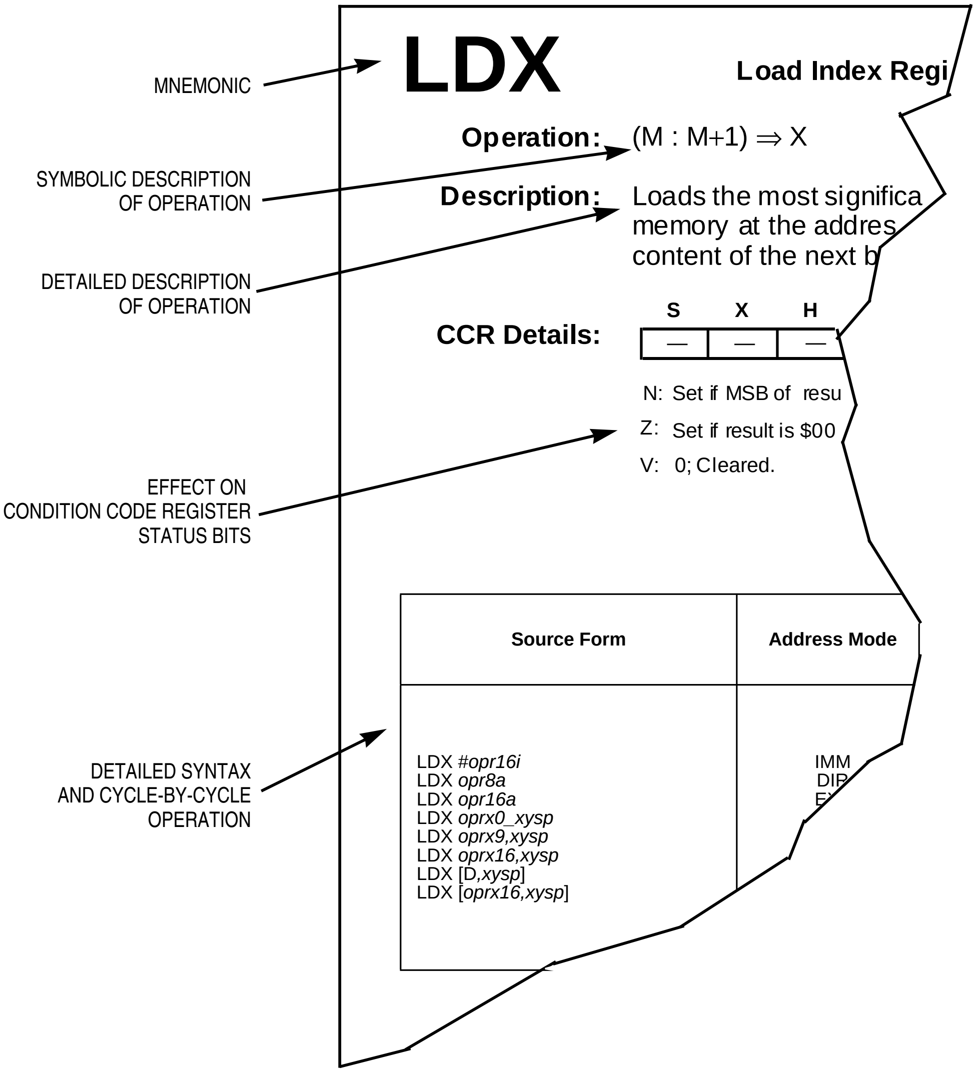


Figure 6-1. Example Glossary Page

Each entry contains symbolic and textual descriptions of operation, information concerning the effect of operation on status bits in the condition code register, and a table that describes assembler syntax, address mode variations, and cycle-by-cycle execution of the instruction.

6.3 Condition Code Changes

The following special characters are used to describe the effects of instruction execution on the status bits in the condition code register.

– — Status bit not affected by operation

0 — Status bit cleared by operation

1 — Status bit set by operation

∆ — Status bit affected by operation

⇓ — Status bit may be cleared or remain set, but is not set by operation.

⇑ — Status bit may be set or remain cleared, but is not cleared by operation.

? — Status bit may be changed by operation, but the final state is not defined.

! — Status bit used for a special purpose

6.4 Object Code Notation

The digits 0 to 9 and the uppercase letters A to F are used to express hexadecimal values. Pairs of lowercase letters represent the 8-bit values as described here.

`dd` — 8-bit direct address $0000 to $00FF; high byte assumed to be $00

`ee` — High-order byte of a 16-bit constant offset for indexed addressing

`eb` — Exchange/transfer post-byte

`ff` — Low-order eight bits of a 9-bit signed constant offset for indexed addressing, or low-order byte of a 16-bit constant offset for indexed addressing

`hh` — High-order byte of a 16-bit extended address

`ii` — 8-bit immediate data value

`jj` — High-order byte of a 16-bit immediate data value

`kk` — Low-order byte of a 16-bit immediate data value

`lb` — Loop primitive (DBNE) post-byte

`ll` — Low-order byte of a 16-bit extended address

`mm` — 8-bit immediate mask value for bit manipulation instructions; set bits indicate bits to be affected

`pg` — Program overlay page (bank) number used in CALL instruction

`qq` — High-order byte of a 16-bit relative offset for long branches

`tn` — Trap number $30–$39 or $40–$FF

`rr` — Signed relative offset $80 (–128) to $7F (+127) offset relative to the byte following the relative offset byte, or low-order byte of a 16-bit relative offset for long branches

`xb` — Indexed addressing post-byte

6.5 Source Forms

The glossary pages provide only essential information about assembler source forms. Assemblers generally support a number of assembler directives, allow definition of program labels, and have special conventions for comments. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Assemblers are typically flexible about the use of spaces and tabs. Often, any number of spaces or tabs can be used where a single space is shown on the glossary pages. Spaces and tabs are also normally allowed before and after commas. When program labels are used, there must also be at least one tab or space before all instruction mnemonics. This required space is not apparent in the source forms.

Everything in the source forms columns, *except expressions in italic characters*, is literal information which must appear in the assembly source file exactly as shown. The initial 3- to 5-letter mnemonic is always a literal expression. All commas, pound signs (#), parentheses, square brackets ([or]), plus signs (+), minus signs (–), and the register designation D (as in [D,...), are literal characters.

Groups of italic characters in the columns represent variable information to be supplied by the programmer. These groups can include any alphanumeric character or the underscore character, but cannot include a space or comma. For example, the groups *xysp* and *oprx0_xysp* are both valid, but the two groups *oprx0 xysp* are not valid because there is a space between them. Permitted syntax is described here.

The definition of a legal label or expression varies from assembler to assembler. Assemblers also vary in the way CPU registers are specified. Refer to assembler documentation for detailed information. Recommended register designators are a, A, b, B, ccr, CCR, d, D, x, X, y, Y, sp, SP, pc, and PC.

abc — Any one legal register designator for accumulators A or B or the CCR

abcdxys — Any one legal register designator for accumulators A or B, the CCR, the double accumulator D, index registers X or Y, or the SP. Some assemblers may accept t2, T2, t3, or T3 codes in certain cases of transfer and exchange instructions, but these forms are intended for Freescale use only.

abd — Any one legal register designator for accumulators A or B or the double accumulator D

abdxys — Any one legal register designator for accumulators A or B, the double accumulator D, index register X or Y, or the SP

dxys — Any one legal register designation for the double accumulator D, index registers X or Y, or the SP

msk8 — Any label or expression that evaluates to an 8-bit value. Some assemblers require a # symbol before this value.

opr8i — Any label or expression that evaluates to an 8-bit immediate value

opr16i — Any label or expression that evaluates to a 16-bit immediate value

opr8a — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low-order 8 bits of an address in the direct page of the 64-Kbyte address space ($00xx).

opr16a — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.

oprx0_xysp — This word breaks down into one of the following alternative forms that assemble to an 8-bit indexed addressing postbyte code. These forms generate the same object code except for the value of the postbyte code, which is designated as xb in the object code columns of the glossary pages. As with the source forms, treat all commas, plus signs, and minus signs as literal syntax elements. The italicized words used in these forms are included in this key.

oprx5,*xysp*
oprx3,–*xys*
oprx3,+*xys*
oprx3,*xys*–
oprx3,*xys*+
abd,*xysp*

oprx3 — Any label or expression that evaluates to a value in the range +1 to +8

oprx5 — Any label or expression that evaluates to a 5-bit value in the range –16 to +15

oprx9 — Any label or expression that evaluates to a 9-bit value in the range –256 to +255

oprx16 — Any label or expression that evaluates to a 16-bit value. Since the CPU12 has a 16-bit address bus, this can be either a signed or an unsigned value.

page — Any label or expression that evaluates to an 8-bit value. The CPU12 recognizes up to an 8-bit page value for memory expansion but not all MCUs that include the CPU12 implement all of these bits. It is the programmer's responsibility to limit the page value to legal values for the intended MCU system. Some assemblers require a # symbol before this value.

rel8 — Any label or expression that refers to an address that is within –128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.

rel9 — Any label or expression that refers to an address that is within –256 to +255 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 9-bit signed offset and include it in the object code for this instruction. The sign bit for this 9-bit value is encoded by the assembler as a bit in the looping postbyte (lb) of one of the loop control instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE. The remaining eight bits of the offset are included as an extra byte of object code.

rel16 — Any label or expression that refers to an address anywhere in the 64-Kbyte address space. The assembler will calculate the 16-bit signed offset between this address and the next address after the last byte of object code for this instruction and include it in the object code for this instruction.

trapnum — Any label or expression that evaluates to an 8-bit number in the range $30–$39 or $40–$FF. Used for TRAP instruction.

xys — Any one legal register designation for index registers X or Y or the SP

xysp — Any one legal register designation for index registers X or Y, the SP, or the PC. The reference point for PC-relative instructions is the next address after the last byte of object code for the current instruction.

6.6 Cycle-by-Cycle Execution

This information is found in the tables at the bottom of each instruction glossary page. Entries show how many bytes of information are accessed from different areas of memory during the course of instruction execution. With this information and knowledge of the type and speed of memory in the system, a user can determine the execution time for any instruction in any system.

A single letter code in the column represents a single CPU cycle. Uppercase letters indicate 16-bit access cycles. There are cycle codes for each addressing mode variation of each instruction. Simply count code letters to determine the execution time of an instruction in a best-case system. An example of a best-case system is a single-chip 16-bit system with no 16-bit off-boundary data accesses to any locations other than on-chip RAM.

Many conditions can cause one or more instruction cycles to be stretched, but the CPU is not aware of the stretch delays because the clock to the CPU is temporarily stopped during these delays.

The following paragraphs explain the cycle code letters used and note conditions that can cause each type of cycle to be stretched.

f — Free cycle. This indicates a cycle where the CPU does not require use of the system buses. An f cycle is always one cycle of the system bus clock. These cycles can be used by a queue controller or the background debug system to perform single cycle accesses without disturbing the CPU.

g — Read 8-bit PPAGE register. These cycles are used only with the CALL instruction to read the current value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.

I — Read indirect pointer. Indexed indirect instructions use this 16-bit pointer from memory to address the operand for the instruction. These are always 16-bit reads but they can be either aligned or misaligned. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned access to a memory that is not designed for single-cycle misaligned access.

i — Read indirect PPAGE value. These cycles are only used with indexed indirect versions of the CALL instruction, where the 8-bit value for the memory expansion page register of the CALL destination is fetched from an indirect memory location. These cycles are stretched only when controlled by a chip-select circuit that is programmed for slow memory.

n — Write 8-bit PPAGE register. These cycles are used only with the CALL and RTC instructions to write the destination value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.

O — Optional cycle. Program information is always fetched as aligned 16-bit words. When an instruction consists of an odd number of bytes, and the first byte is misaligned, an O cycle is used to make an additional program word access (P) cycle that maintains queue order. In all other cases, the O cycle appears as a free (f) cycle. The $18 prebyte for page two opcodes is treated as a special 1-byte instruction. If the prebyte is misaligned, the O cycle is used as a program word access for the prebyte; if the prebyte is aligned, the O cycle appears as a free cycle. If the remainder of the instruction consists of an odd number of bytes, another O cycle is required some time before the instruction is completed. If the O cycle for the prebyte is treated as a P cycle, any subsequent O cycle in the same instruction is treated as an f cycle; if the O cycle for the prebyte is treated as an f cycle, any subsequent O cycle in the same instruction is treated as a P cycle. Optional cycles used for program word accesses can be extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. Optional cycles used as free cycles are never stretched.

P — Program word access. Program information is fetched as aligned 16-bit words. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored externally. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory.

r — 8-bit data read. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.

R — 16-bit data read. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to memory that is not designed for single-cycle misaligned access.

s — Stack 8-bit data. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.

S — Stack 16-bit data. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the SP is pointing to external memory. There can be additional stretching if the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single cycle misaligned access. The internal RAM is designed to allow single cycle misaligned word access.

w — 8-bit data write. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.

W — 16-bit data write. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned access to a memory that is not designed for single-cycle misaligned access.

u — Unstack 8-bit data. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.

U — Unstack 16-bit data. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the SP is pointing to external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single-cycle misaligned access. The internal RAM is designed to allow single-cycle misaligned word access.

V — Vector fetch. Vectors are always aligned 16-bit words. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory.

t — 8-bit conditional read. These cycles are either data read cycles or unused cycles, depending on the data and flow of the REVW instruction. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.

T — 16-bit conditional read. These cycles are either data read cycles or free cycles, depending on the data and flow of the REV or REVW instruction. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single-cycle misaligned access.

x — 8-bit conditional write. These cycles are either data write cycles or free cycles, depending on the data and flow of the REV or REVW instruction. These cycles are only stretched when controlled by a chip-select circuit programmed for slow memory.

Special Notation for Branch Taken/Not Taken Cases

PPP/P — Short branches require three cycles if taken, one cycle if not taken. Since the instruction consists of a single word containing both an opcode and an 8-bit offset, the not-taken case is simple — the queue advances, another program word fetch is made, and execution continues with the next instruction. The taken case requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is determined, then the CPU performs three program word fetches from that address.

OPPP/OPO — Long branches require four cycles if taken, three cycles if not taken. Optional cycles are required because all long branches are page two opcodes, and thus include the $18 prebyte. The CPU12 treats the prebyte as a special 1-byte instruction. If the prebyte is misaligned, the optional cycle is used to perform a program word access; if the prebyte is aligned, the optional cycle is used to perform a free cycle. As a result, both the taken and not-taken cases use one optional cycle for the prebyte. In the not-taken case, the queue must advance so that execution can continue with the next instruction, and another optional cycle is required to maintain the queue. The taken case requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is determined, then the CPU performs three program word fetches from that address.

6.7 Glossary

This subsection contains an entry for each assembler mnemonic, in alphabetic order.

ABA Add Accumulator B to Accumulator A ABA

Operation: (A) + (B) ⇒ A

Description: Adds the content of accumulator B to the content of accumulator A and places the result in A. The content of B is not changed. This instruction affects the H status bit so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: $A3 \bullet B3 + B3 \bullet \overline{R3} + \overline{R3} \bullet A3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet B7 \bullet \overline{R7} + \overline{A7} \bullet \overline{B7} \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $A7 \bullet B7 + B7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ABA	INH	18 06	OO	OO

ABX Add Accumulator B to Index Register X ABX

Operation: (B) + (X) ⇒ X

Description: Adds the 8-bit unsigned content of accumulator B to the content of index register X considering the possible carry out of the low-order byte of X; places the result in X. The content of B is not changed.

This mnemonic is implemented by the LEAX B,X instruction. The LEAX instruction allows A, B, D, or a constant to be added to X. For compatibility with the M68HC11, the mnemonic ABX is translated into the LEAX B,X instruction by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ABX *translates to...* LEAX B,X	IDX	`1A E5`	`Pf`	`PP`[1]

[1] Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

ABY Add Accumulator B to Index Register Y ABY

Operation: (B) + (Y) ⇒ Y

Description: Adds the 8-bit unsigned content of accumulator B to the content of index register Y considering the possible carry out of the low-order byte of Y; places the result in Y. The content of B is not changed.

This mnemonic is implemented by the LEAY B,Y instruction. The LEAY instruction allows A, B, D, or a constant to be added to Y. For compatibility with the M68HC11, the mnemonic ABY is translated into the LEAY B,Y instruction by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ABY *translates to...* LEAY B,Y	IDX	`19 ED`	`Pf`	`PP`[1]

[1] Due to internal M68HC12CPU requirements, the program word fetch is performed twice to the same address during this instruction.

ADCA Add with Carry to A ADCA

Operation: (A) + (M) + C ⇒ A

Description: Adds the content of accumulator A to the content of memory location M, then adds the value of the C bit and places the result in A. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: $A3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet A3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ADCA *#opr8i*	IMM	89 ii	P	P
ADCA *opr8a*	DIR	99 dd	rPf	rfP
ADCA *opr16a*	EXT	B9 hh ll	rPO	rOP
ADCA *oprx0_xysp*	IDX	A9 xb	rPf	rfP
ADCA *oprx9,xysp*	IDX1	A9 xb ff	rPO	rPO
ADCA *oprx16,xysp*	IDX2	A9 xb ee ff	frPP	frPP
ADCA [D,*xysp*]	[D,IDX]	A9 xb	fIfrPf	fIfrfP
ADCA [*oprx16,xysp*]	[IDX2]	A9 xb ee ff	fIPrPf	fIPrfP

ADCB Add with Carry to B ADCB

Operation: (B) + (M) + C $\Rightarrow$ B

Description: Adds the content of accumulator B to the content of memory location M, then adds the value of the C bit and places the result in B. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: $X3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet X3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $X7 \bullet M7 \bullet \overline{R7} + \overline{X7} \bullet \overline{M7} \bullet R7$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $X7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet X7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ADCB #*opr8i*	IMM	`C9 ii`	`P`	`P`
ADCB *opr8a*	DIR	`D9 dd`	`rPf`	`rfP`
ADCB *opr16a*	EXT	`F9 hh ll`	`rPO`	`rOP`
ADCB *oprx0_xysp*	IDX	`E9 xb`	`rPf`	`rfP`
ADCB *oprx9,xysp*	IDX1	`E9 xb ff`	`rPO`	`rPO`
ADCB *oprx16,xysp*	IDX2	`E9 xb ee ff`	`frPP`	`frPP`
ADCB [D,*xysp*]	[D,IDX]	`E9 xb`	`fIfrPf`	`fIfrfP`
ADCB [*oprx16,xysp*]	[IDX2]	`E9 xb ee ff`	`fIPrPf`	`fIPrfP`

ADDA Add without Carry to A ADDA

Operation: (A) + (M) ⇒ A

Description: Adds the content of memory location M to accumulator A and places the result in A. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: $A3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet A3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ADDA #*opr8i*	IMM	`8B ii`	`P`	`P`
ADDA *opr8a*	DIR	`9B dd`	`rPf`	`rfP`
ADDA *opr16a*	EXT	`BB hh ll`	`rPO`	`rOP`
ADDA *oprx0_xysp*	IDX	`AB xb`	`rPf`	`rfP`
ADDA *oprx9,xysp*	IDX1	`AB xb ff`	`rPO`	`rPO`
ADDA *oprx16,xysp*	IDX2	`AB xb ee ff`	`frPP`	`frPP`
ADDA [D,*xysp*]	[D,IDX]	`AB xb`	`fIfrPf`	`fIfrfP`
ADDA [*oprx16,xysp*]	[IDX2]	`AB xb ee ff`	`fIPrPf`	`fIPrfP`

ADDB Add without Carry to B ADDB

Operation: (B) + (M) ⇒ B

Description: Adds the content of memory location M to accumulator B and places the result in B. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: $B3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet B3$
Set if there was a carry from bit 3; cleared otherwise

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $B7 \bullet M7 \bullet \overline{R7} + \overline{B7} \bullet \overline{M7} \bullet R7$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $B7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet B7$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ADDB #*opr8i*	IMM	`CB ii`	`P`	`P`
ADDB *opr8a*	DIR	`DB dd`	`rPf`	`rfP`
ADDB *opr16a*	EXT	`FB hh ll`	`rPO`	`rOP`
ADDB *oprx0_xysp*	IDX	`EB xb`	`rPf`	`rfP`
ADDB *oprx9,xysp*	IDX1	`EB xb ff`	`rPO`	`rPO`
ADDB *oprx16,xysp*	IDX2	`EB xb ee ff`	`frPP`	`frPP`
ADDB [D,*xysp*]	[D,IDX]	`EB xb`	`fIfrPf`	`fIfrfP`
ADDB [*oprx16,xysp*]	[IDX2]	`EB xb ee ff`	`fIPrPf`	`fIPrfP`

ADDD Add Double Accumulator ADDD

Operation: (A : B) + (M : M+1) ⇒ A : B

Description: Adds the content of memory location M concatenated with the content of memory location M +1 to the content of double accumulator D and places the result in D. Accumulator A forms the high-order half of 16-bit double accumulator D; accumulator B forms the low-order half.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet M15 \bullet \overline{R15} + \overline{D15} \bullet \overline{M15} \bullet R15$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $D15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet D15$
Set if there was a carry from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ADDD *#opr16i*	IMM	`C3 jj kk`	`PO`	`OP`
ADDD *opr8a*	DIR	`D3 dd`	`RPf`	`RfP`
ADDD *opr16a*	EXT	`F3 hh ll`	`RPO`	`ROP`
ADDD *oprx0_xysp*	IDX	`E3 xb`	`RPf`	`RfP`
ADDD *oprx9,xysp*	IDX1	`E3 xb ff`	`RPO`	`RPO`
ADDD *oprx16,xysp*	IDX2	`E3 xb ee ff`	`fRPP`	`fRPP`
ADDD [D,*xysp*]	[D,IDX]	`E3 xb`	`fIfRPF`	`fIfRfP`
ADDD [*oprx16,xysp*]	[IDX2]	`E3 xb ee ff`	`fIPRPf`	`fIPRfP`

ANDA — Logical AND A — ANDA

Operation: (A) • (M) ⇒ A

Description: Performs logical AND between the content of memory location M and the content of accumulator A. The result is placed in A. After the operation is performed, each bit of A is the logical AND of the corresponding bits of M and of A before the operation began.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ANDA #*opr8i*	IMM	`84 ii`	`P`	`P`
ANDA *opr8a*	DIR	`94 dd`	`rPf`	`rfP`
ANDA *opr16a*	EXT	`B4 hh ll`	`rPO`	`rOP`
ANDA *oprx0_xysp*	IDX	`A4 xb`	`rPf`	`rfP`
ANDA *oprx9,xysp*	IDX1	`A4 xb ff`	`rPO`	`rPO`
ANDA *oprx16,xysp*	IDX2	`A4 xb ee ff`	`frPP`	`frPP`
ANDA [D,*xysp*]	[D,IDX]	`A4 xb`	`fIfrPf`	`fIfrfP`
ANDA [*oprx16,xysp*]	[IDX2]	`A4 xb ee ff`	`fIPrPf`	`fIPrfP`

ANDB — Logical AND B — ANDB

Operation: (B) • (M) ⇒ B

Description: Performs logical AND between the content of memory location M and the content of accumulator B. The result is placed in B. After the operation is performed, each bit of B is the logical AND of the corresponding bits of M and of B before the operation began.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ANDB #*opr8i*	IMM	`C4 ii`	`P`	`P`
ANDB *opr8a*	DIR	`D4 dd`	`rPf`	`rfP`
ANDB *opr16a*	EXT	`F4 hh ll`	`rPO`	`rOP`
ANDB *oprx0_xysp*	IDX	`E4 xb`	`rPf`	`rfP`
ANDB *oprx9,xysp*	IDX1	`E4 xb ff`	`rPO`	`rPO`
ANDB *oprx16,xysp*	IDX2	`E4 xb ee ff`	`frPP`	`frPP`
ANDB [D,*xysp*]	[D,IDX]	`E4 xb`	`fIfrPf`	`fIfrfP`
ANDB [*oprx16,xysp*]	[IDX2]	`E4 xb ee ff`	`fIPrPf`	`fIPrfP`

ANDCC Logical AND CCR with Mask ANDCC

Operation: (CCR) • (Mask) ⇒ CCR

Description: Performs bitwise logical AND between the content of a mask operand and the content of the CCR. The result is placed in the CCR. After the operation is performed, each bit of the CCR is the result of a logical AND with the corresponding bits of the mask. To clear CCR bits, clear the corresponding mask bits. CCR bits that correspond to ones in the mask are not changed by the ANDCC operation.

If the I mask bit is cleared, there is a 1-cycle delay before the system allows interrupt requests. This prevents interrupts from occurring between instructions in the sequences CLI, WAI and CLI, STOP (CLI is equivalent to ANDCC #$EF).

CCR Details:

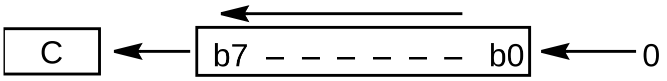
S	X	H	I	N	Z	V	C
⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓

Condition code bits are cleared if the corresponding bit was 0 before the operation or if the corresponding bit in the mask is 0.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ANDCC #*opr8i*	IMM	`10 ii`	`P`	`P`

ASL Arithmetic Shift Left Memory (same as LSL) ASL

Operation:



Description: Shifts all bits of memory location M one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of M.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

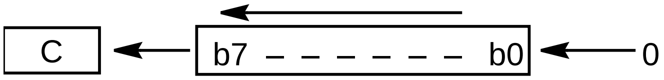
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M7
Set if the MSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	**M68HC12**
ASL *opr16a*	EXT	`78 hh ll`	`rPwO`	`rOPw`
ASL *oprx0_xysp*	IDX	`68 xb`	`rPw`	`rPw`
ASL *oprx9,xysp*	IDX1	`68 xb ff`	`rPwO`	`rPOw`
ASL *oprx16,xysp*	IDX2	`68 xb ee ff`	`frPwP`	`frPPw`
ASL [D,*xysp*]	[D,IDX]	`68 xb`	`fIfrPw`	`fIfrPw`
ASL [*oprx16,xysp*]	[IDX2]	`68 xb ee ff`	`fIPrPw`	`fIPrPw`

ASLA — Arithmetic Shift Left A (same as LSLA) — ASLA

Operation:



Description: Shifts all bits of accumulator A one bit position to the left. Bit 0 is loaded with a 0. TheC status bit is loaded from the most significant bit of A.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

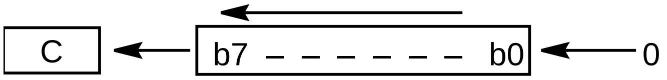
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A7
Set if the MSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ASLA	INH	48	O	O

ASLB Arithmetic Shift Left B (same as LSLB) ASLB

Operation:



Description: Shifts all bits of accumulator B one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of B.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

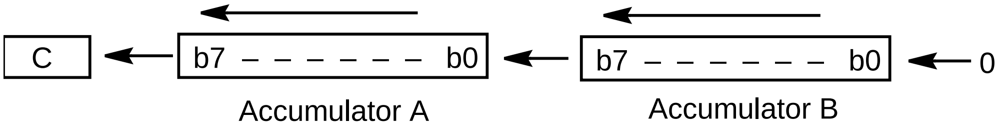
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B7
Set if the MSB of B was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	**M68HC12**
ASLB	INH	58	1	O

ASLD — Arithmetic Shift Left Double Accumulator (same as LSLD) — ASLD

Operation:



Description: Shifts all bits of double accumulator D one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of D.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

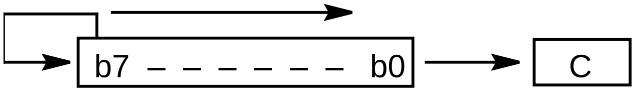
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: D15
Set if the MSB of D was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ASLD	INH	59	O	O

ASR Arithmetic Shift Right Memory ASR

Operation:



Description: Shifts all bits of memory location M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

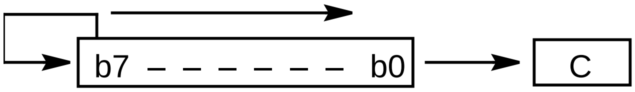
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ASR *opr16a*	EXT	`77 hh ll`	`rPwO`	`rOPw`
ASR *oprx0_xysp*	IDX	`67 xb`	`rPw`	`rPw`
ASR *oprx9,xysp*	IDX1	`67 xb ff`	`rPwO`	`rPOw`
ASR *oprx16,xysp*	IDX2	`67 xb ee ff`	`frPwP`	`frPPw`
ASR [D,*xysp*]	[D,IDX]	`67 xb`	`fIfrPw`	`fIfrPw`
ASR [*oprx16,xysp*]	[IDX2]	`67 xb ee ff`	`fIPrPw`	`fIPrPw`

ASRA　Arithmetic Shift Right A　ASRA

Operation:



Description: Shifts all bits of accumulator A one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

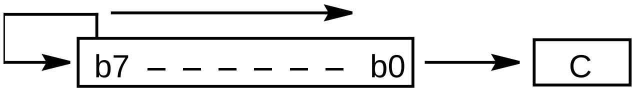
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A0
Set if the LSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ASRA	INH	47	O	O

ASRB Arithmetic Shift Right B ASRB

Operation:



Description: Shifts all bits of accumulator B one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B0
Set if the LSB of B was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ASRB	INH	57	O	O

BCC — Branch if Carry Cleared (Same as BHS) — BCC

Operation: If C = 0, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description: Tests the C status bit and branches if C = 0.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BCC *rel8*	REL	`24 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	$Z + (N \oplus V) = 0$	r≤m	BLE	2F	Signed
r≥m	BGE	2C	$N \oplus V = 0$	r<m	BLT	2D	Signed
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Signed
r≤m	BLE	2F	$Z + (N \oplus V) = 1$	r>m	BGT	2E	Signed
r<m	BLT	2D	$N \oplus V = 1$	r≥m	BGE	2C	Signed
r>m	BHI	22	$C + Z = 0$	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	$C = 0$	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Unsigned
r≤m	BLS	23	$C + Z = 1$	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	$C = 1$	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	$C = 1$	No Carry	BCC	24	Simple
Negative	BMI	2B	$N = 1$	Plus	BPL	2A	Simple
Overflow	BVS	29	$V = 1$	No Overflow	BVC	28	Simple
r=0	BEQ	27	$Z = 1$	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BCLR Clear Bits in Memory BCLR

Operation: $(M) \bullet (\overline{\text{Mask}}) \Rightarrow M$

Description: Clears bits in location M. To clear a bit, set the corresponding bit in the mask byte. Bits in M that correspond to 0s in the mask byte are not changed. Mask bytes can be located at PC + 2, PC + 3, or PC + 4, depending on addressing mode used.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode[1]	Object Code	Access Detail	
			HCS12	M68HC12
BCLR *opr8a, msk8*	DIR	`4D dd mm`	`rPwO`	`rPOw`
BCLR *opr16a, msk8*	EXT	`1D hh ll mm`	`rPwP`	`rPPw`
BCLR *oprx0_xysp, msk8*	IDX	`0D xb mm`	`rPwO`	`rPOw`
BCLR *oprx9,xysp, msk8*	IDX1	`0D xb ff mm`	`rPwP`	`rPwP`
BCLR *oprx16,xysp, msk8*	IDX2	`0D xb ee ff mm`	`frPwPO`	`frPwOP`

[1] Indirect forms of indexed addressing cannot be used with this instruction.

BCS Branch if Carry Set (Same as BLO) BCS

Operation: If C = 1, then (PC) + \$0002 + Rel ⇒ PC

Simple branch

Description: Tests the C status bit and branches if C = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BCS *rel8*	REL	`25 rr`	`PPP/P`(1)	`PPP/P`(1)

1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BEQ Branch if Equal BEQ

Operation: If $Z = 1$, then $(PC) + \$0002 + Rel \Rightarrow PC$

Simple branch

Description: Tests the Z status bit and branches if Z = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BEQ *rel8*	REL	`27 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BGE Branch if Greater than or Equal to Zero BGE

Operation: If N ⊕ V = 0, then (PC) + $0002 + Rel ⇒ PC

For signed two's complement values
if (Accumulator) ≥ (Memory), then branch

Description: BGE can be used to branch after comparing or subtracting signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BGE *rel8*	REL	`2C rr`	`PPP/P`(1)	`PPP/P`(1)

1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BGND Enter Background Debug Mode BGND

Description: BGND operates like a software interrupt, except that no registers are stacked. First, the current PC value is stored in internal CPU register TMP2. Next, the BDM ROM and background register block become active. The BDM ROM contains a substitute vector, mapped to the address of the software interrupt vector, which points to routines in the BDM ROM that control background operation. The substitute vector is fetched, and execution continues from the address that it points to. Finally, the CPU checks the location that TMP2 points to. If the value stored in that location is $00 (the BGND opcode), TMP2 is incremented, so that the instruction that follows the BGND instruction is the first instruction executed when normal program execution resumes.

For all other types of BDM entry, the CPU performs the same sequence of operations as for a BGND instruction, but the value stored in TMP2 already points to the instruction that would have executed next had BDM not become active. If active BDM is triggered just as a BGND instruction is about to execute, the BDM firmware does increment TMP2, but the change does not affect resumption of normal execution.

While BDM is active, the CPU executes debugging commands received via a special single-wire serial interface. BDM is terminated by the execution of specific debugging commands. Upon exit from BDM, the background/boot ROM and registers are disabled, the instruction queue is refilled starting with the return address pointed to by TMP2, and normal processing resumes.

BDM is normally disabled to avoid accidental entry. While BDM is disabled, BGND executes as described, but the firmware causes execution to return to the user program. Refer to **Section 8. Instruction Queue** for more information concerning BDM.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BGND	INH	`00`	`VfPPP`	`VfPPP`

BGT Branch if Greater than Zero BGT

Operation: If $Z + (N \oplus V) = 0$, then $(PC) + \$0002 + Rel \Rightarrow PC$

For signed two's complement values
if (Accumulator) > (Memory), then branch

Description: BGT can be used to branch after comparing or subtracting signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than the value in M. After CBA or SBA, the branch occurs if the value in B is greater than the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BGT *rel8*	REL	`2E rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	$Z + (N \oplus V) = 0$	r≤m	BLE	2F	Signed
r≥m	BGE	2C	$N \oplus V = 0$	r<m	BLT	2D	Signed
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Signed
r≤m	BLE	2F	$Z + (N \oplus V) = 1$	r>m	BGT	2E	Signed
r<m	BLT	2D	$N \oplus V = 1$	r≥m	BGE	2C	Signed
r>m	BHI	22	$C + Z = 0$	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	$C = 0$	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Unsigned
r≤m	BLS	23	$C + Z = 1$	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	$C = 1$	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	$C = 1$	No Carry	BCC	24	Simple
Negative	BMI	2B	$N = 1$	Plus	BPL	2A	Simple
Overflow	BVS	29	$V = 1$	No Overflow	BVC	28	Simple
r=0	BEQ	27	$Z = 1$	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BHI Branch if Higher BHI

Operation: If $C + Z = 0$, then $(PC) + \$0002 + Rel \Rightarrow PC$

For unsigned values, if (Accumulator) > (Memory), then branch

Description: BHI can be used to branch after comparing or subtracting unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than the value in M. After CBA or SBA, the branch occurs if the value in B is greater than the value in A. BHI should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BHI *rel8*	REL	`22 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	$Z + (N \oplus V) = 0$	r≤m	BLE	2F	Signed
r≥m	BGE	2C	$N \oplus V = 0$	r<m	BLT	2D	Signed
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Signed
r≤m	BLE	2F	$Z + (N \oplus V) = 1$	r>m	BGT	2E	Signed
r<m	BLT	2D	$N \oplus V = 1$	r≥m	BGE	2C	Signed
r>m	BHI	22	$C + Z = 0$	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	$C = 0$	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Unsigned
r≤m	BLS	23	$C + Z = 1$	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	$C = 1$	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	$C = 1$	No Carry	BCC	24	Simple
Negative	BMI	2B	$N = 1$	Plus	BPL	2A	Simple
Overflow	BVS	29	$V = 1$	No Overflow	BVC	28	Simple
r=0	BEQ	27	$Z = 1$	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BHS — Branch if Higher or Same (Same as BCC) — BHS

Operation: If C = 0, then (PC) + $0002 + Rel ⇒ PC

For unsigned values, if (Accumulator) ≥ (Memory), then branch

Description: BHS can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A. BHS should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BHS *rel8*	REL	`24 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BITA Bit Test A BITA

Operation: (A) • (M)

Description: Performs bitwise logical AND on the content of accumulator A and the content of memory location M and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the accumulator and the memory location. Neither the content of the accumulator nor the content of the memory location is affected.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BITA #*opr8i*	IMM	`85 ii`	`P`	`P`
BITA *opr8a*	DIR	`95 dd`	`rPf`	`rfP`
BITA *opr16a*	EXT	`B5 hh ll`	`rPO`	`rOP`
BITA *oprx0_xysp*	IDX	`A5 xb`	`rPf`	`rfP`
BITA *oprx9,xysp*	IDX1	`A5 xb ff`	`rPO`	`rPO`
BITA *oprx16,xysp*	IDX2	`A5 xb ee ff`	`frPP`	`frPP`
BITA [D,*xysp*]	[D,IDX]	`A5 xb`	`fIfrPf`	`fIfrfP`
BITA [*oprx16,xysp*]	[IDX2]	`A5 xb ee ff`	`fIPrPf`	`fIPrfP`

BITB Bit Test B BITB

Operation: (B) • (M)

Description: Performs bitwise logical AND on the content of accumulator B and the content of memory location M and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the accumulator and the memory location. Neither the content of the accumulator nor the content of the memory location is affected.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BITB #*opr8i*	IMM	`C5 ii`	`P`	`P`
BITB *opr8a*	DIR	`D5 dd`	`rPf`	`rfP`
BITB *opr16a*	EXT	`F5 hh ll`	`rPO`	`rOP`
BITB *oprx0_xysp*	IDX	`E5 xb`	`rPf`	`rfP`
BITB *oprx9,xysp*	IDX1	`E5 xb ff`	`rPO`	`rPO`
BITB *oprx16,xysp*	IDX2	`E5 xb ee ff`	`frPP`	`frPP`
BITB [D,*xysp*]	[D,IDX]	`E5 xb`	`fIfrPf`	`fIfrfP`
BITB [*oprx16,xysp*]	[IDX2]	`E5 xb ee ff`	`fIPrPf`	`fIPrfP`

BLE Branch if Less Than or Equal to Zero BLE

Operation: If $Z + (N \oplus V) = 1$, then $(PC) + \$0002 + Rel \Rightarrow PC$

For signed two’s complement numbers
if (Accumulator) ≤ (Memory), then branch

Description: BLE can be used to branch after subtracting or comparing signed two’s complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is less than or equal to the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BLE *rel8*	REL	`2F rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	$Z + (N \oplus V) = 0$	r≤m	BLE	2F	Signed
r≥m	BGE	2C	$N \oplus V = 0$	r<m	BLT	2D	Signed
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Signed
r≤m	BLE	2F	$Z + (N \oplus V) = 1$	r>m	BGT	2E	Signed
r<m	BLT	2D	$N \oplus V = 1$	r≥m	BGE	2C	Signed
r>m	BHI	22	$C + Z = 0$	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	$C = 0$	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	$Z = 1$	r≠m	BNE	26	Unsigned
r≤m	BLS	23	$C + Z = 1$	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	$C = 1$	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	$C = 1$	No Carry	BCC	24	Simple
Negative	BMI	2B	$N = 1$	Plus	BPL	2A	Simple
Overflow	BVS	29	$V = 1$	No Overflow	BVC	28	Simple
r=0	BEQ	27	$Z = 1$	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BLO **Branch if Lower (Same as BCS)** BLO

Operation: If C = 1, then (PC) + $0002 + Rel ⇒ PC

For unsigned values, if (Accumulator) < (Memory), then branch

Description: BLO can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A. BLO should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BLO *rel8*	REL	`25 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BLS Branch if Lower or Same BLS

Operation: If C + Z = 1, then (PC) + $0002 + Rel ⇒ PC

For unsigned values, if (Accumulator) ≤ (Memory), then branch

Description: If BLS is executed immediately after execution of CBA, CMPA, CMPB, CMPD, CPX, CPY, SBA, SUBA, SUBB, or SUBD, a branch occurs if and only if the unsigned binary number in the accumulator is less than or equal to the unsigned binary number in memory. Generally not useful after INC/DEC, LD/ST, and TST/CLR/COM because these instructions do not affect the C status bit.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BLS *rel8*	REL	`23 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BLT Branch if Less than Zero BLT

Operation: If $N \oplus V = 1$, then (PC) + \$0002 + Rel ⇒ PC

For signed two’s complement numbers
if (Accumulator) < (Memory), then branch

Description: BLT can be used to branch after subtracting or comparing signed two’s complement values. After CMPA, CMPB, CMPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BLT *rel8*	REL	`2D rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BMI Branch if Minus BMI

Operation: If N = 1, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description: Tests the N status bit and branches if N = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BMI *rel8*	REL	`2B rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BNE Branch if Not Equal to Zero BNE

Operation: If Z = 0, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description: Tests the Z status bit and branches if Z = 0.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BNE *rel8*	REL	`26 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BPL Branch if Plus BPL

Operation: If N = 0, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description: Tests the N status bit and branches if N = 0.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BPL *rel8*	REL	`2A rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BRA Branch Always BRA

Operation: (PC) + $0002 + Rel ⇒ PC

Description: Unconditional branch to an address calculated as shown in the expression. Rel is a relative offset stored as a two’s complement number in the second byte of the branch instruction.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the BRA branch condition is always satisfied, the branch is always taken, and the instruction queue must always be refilled.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BRA *rel8*	REL	`20 rr`	`PPP`	`PPP`

BRCLR Branch if Bits Cleared BRCLR

Operation: If (M) • (Mask) = 0, then branch

Description: Performs a bitwise logical AND of memory location M and the mask supplied with the instruction, then branches if and only if all bits with a value of 1 in the mask byte correspond to bits with a value of 0 in the tested byte. Mask operands can be located at PC + 1, PC + 2, or PC + 4, depending on addressing mode. The branch offset is referenced to the next address after the relative offset (rr) which is the last byte of the instruction object code.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode[1]	Object Code	Access Detail	
			HCS12	M68HC12
BRCLR *opr8a, msk8, rel8*	DIR	`4F dd mm rr`	`rPPP`	`rPPP`
BRCLR *opr16a, msk8, rel8*	EXT	`1F hh ll mm rr`	`rfPPP`	`rfPPP`
BRCLR *oprx0_xysp, msk8, rel8*	IDX	`0F xb mm rr`	`rPPP`	`rPPP`
BRCLR *oprx9,xysp, msk8, rel8*	IDX1	`0F xb ff mm rr`	`rfPPP`	`rffPPP`
BRCLR *oprx16,xysp, msk8, rel8*	IDX2	`0F xb ee ff mm rr`	`PrfPPP`	`frPffPPP`

[1] Indirect forms of indexed addressing cannot be used with this instruction.

BRN Branch Never BRN

Operation: (PC) + $0002 ⇒ PC

Description: Never branches. BRN is effectively a 2-byte NOP that requires one cycle to execute. BRN is included in the instruction set to provide a complement to the BRA instruction. The instruction is useful during program debug, to negate the effect of another branch instruction without disturbing the offset byte. A complement for BRA is also useful in compiler implementations.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the BRN branch condition is never satisfied, the branch is never taken, and only a single program fetch is needed to update the instruction queue.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BRN *rel8*	REL	`21 rr`	`P`	`P`

BRSET Branch if Bits Set BRSET

Operation: If ($\overline{\mathrm{M}}$) • (Mask) = 0, then branch

Description: Performs a bitwise logical AND of the inverse of memory location M and the mask supplied with the instruction, then branches if and only if all bits with a value of 1 in the mask byte correspond to bits with a value of one in the tested byte. Mask operands can be located at PC + 1, PC + 2, or PC + 4, depending on addressing mode. The branch offset is referenced to the next address after the relative offset (rr) which is the last byte of the instruction object code.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode[1]	Object Code	Access Detail	
			HCS12	M68HC12
BRSET *opr8a, msk8, rel8*	DIR	4E dd mm rr	rPPP	rPPP
BRSET *opr16a, msk8, rel8*	EXT	1E hh ll mm rr	rfPPP	rfPPP
BRSET *oprx0_xysp, msk8, rel8*	IDX	0E xb mm rr	rPPP	rPPP
BRSET *oprx9,xysp, msk8, rel8*	IDX1	0E xb ff mm rr	rfPPP	rffPPP
BRSET *oprx16,xysp, msk8, rel8*	IDX2	0E xb ee ff mm rr	PrfPPP	frPffPPP

1. Indirect forms of indexed addressing cannot be used with this instruction.

BSET Set Bit(s) in Memory BSET

Operation: (M) + (Mask) ⇒ M

Description: Sets bits in memory location M. To set a bit, set the corresponding bit in the mask byte. All other bits in M are unchanged. The mask byte can be located at PC + 2, PC + 3, or PC + 4, depending upon addressing mode.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode[1]	Object Code	Access Detail	
			HCS12	M68HC12
BSET *opr8a, msk8*	DIR	`4C dd mm`	`rPwO`	`rPOw`
BSET *opr16a, msk8*	EXT	`1C hh ll mm`	`rPwP`	`rPPw`
BSET *oprx0_xysp, msk8*	IDX	`0C xb mm`	`rPwO`	`rPOw`
BSET *oprx9,xysp, msk8*	IDX1	`0C xb ff mm`	`rPwP`	`rPwP`
BSET *oprx16,xysp, msk8*	IDX2	`0C xb ee ff mm`	`frPwPO`	`frPwOP`

[1] Indirect forms of indexed addressing cannot be used with this instruction.

BSR **Branch to Subroutine** BSR

Operation: (SP) – \$0002 ⇒ SP
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$
(PC) + Rel ⇒ PC

Description: Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction after the BSR as a return address.

Decrements the SP by two, to allow the two bytes of the return address to be stacked.

Stacks the return address (the SP points to the high-order byte of the return address).

Branches to a location determined by the branch offset.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BSR *rel8*	REL	`07 rr`	`SPPP`	`PPPS`

BVC Branch if Overflow Cleared BVC

Operation: If V = 0, then (PC) + \$0002 + Rel ⇒ PC

Simple branch

Description: Tests the V status bit and branches if V = 0.

BVC causes a branch when a previous operation on two's complement binary values does not cause an overflow. That is, when BVC follows a two's complement operation, a branch occurs when the result of the operation is valid.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
BVC *rel8*	REL	`28 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

BVS Branch if Overflow Set BVS

Operation: If V = 1, then (PC) + $0002 + Rel ⇒ PC

Simple branch

Description: Tests the V status bit and branches if V = 1.

BVS causes a branch when a previous operation on two's complement binary values causes an overflow. That is, when BVS follows a two's complement operation, a branch occurs when the result of the operation is invalid.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
BVS *rel8*	REL	`29 rr`	`PPP/P`[1]	`PPP/P`[1]

[1] PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

CALL Call Subroutine in Expanded Memory CALL

Operation:

$\text{(SP)} - \$0002 \Rightarrow \text{SP; } \text{RTN}_H : \text{RTN}_L \Rightarrow \text{M}_{(SP)} : \text{M}_{(SP+1)}$
$\text{(SP)} - \$0001 \Rightarrow \text{SP; (PPAGE)} \Rightarrow \text{M}_{(SP)}$
page ⇒ PPAGE; Subroutine Address ⇒ PC

Description: Sets up conditions to return to normal program flow, then transfers control to a subroutine in expanded memory. Uses the address of the instruction following the CALL as a return address. For code compatibility, CALL also executes correctly in devices that do not have expanded memory capability.

Decrements the SP by two, then stores the return address on the stack. The SP points to the high-order byte of the return address.

Decrements the SP by one, then stacks the current memory page value from the PPAGE register on the stack.

Writes a new page value supplied by the instruction to PPAGE and transfers control to the subroutine.

In indexed-indirect modes, the subroutine address and the PPAGE value are fetched from memory in the order M high byte, M low byte, and new PPAGE value.

Expanded-memory subroutines must be terminated by an RTC instruction, which restores the return address and PPAGE value from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CALL *opr16a, page*	EXT	`4A hh ll pg`	`gnSsPPP`	`gnfSsPPP`
CALL *oprx0_xysp, page*	IDX	`4B xb pg`	`gnSsPPP`	`gnfSsPPP`
CALL *oprx9,xysp, page*	IDX1	`4B xb ff pg`	`gnSsPPP`	`gnfSsPPP`
CALL *oprx16,xysp, page*	IDX2	`4B xb ee ff pg`	`fgnSsPPP`	`fgnfSsPPP`
CALL [D,*xysp*]	[D,IDX]	`4B xb`	`fIignSsPPP`	`fIignSsPPP`
CALL [*oprx16,xysp*]	[IDX2]	`4B xb ee ff`	`fIignSsPPP`	`fIignSsPPP`

CBA Compare Accumulators CBA

Operation: (A) – (B)

Description: Compares the content of accumulator A to the content of accumulator B and sets the condition codes, which may then be used for arithmetic and logical conditional branches. The contents of the accumulators are not changed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{B7} \bullet \overline{R7} + \overline{A7} \bullet B7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet B7 + B7 \bullet R7 + R7 \bullet \overline{A7}$
Set if there was a borrow from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CBA	INH	`18 17`	OO	OO

CLC Clear Carry CLC

Operation: 0 ⇒ C bit

Description: Clears the C status bit. This instruction is assembled as ANDCC #$FE. The ANDCC instruction can be used to clear any combination of bits in the CCR in one operation.

CLC can be used to set up the C bit prior to a shift or rotate instruction involving the C bit.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	0

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	**M68HC12**
CLC *translates to...* ANDCC #$FE	IMM	`10 FE`	`P`	`P`

CLI Clear Interrupt Mask CLI

Operation: 0 ⇒ I bit

Description: Clears the I mask bit. This instruction is assembled as ANDCC #$EF. The ANDCC instruction can be used to clear any combination of bits in the CCR in one operation.

When the I bit is cleared, interrupts are enabled. There is a 1-cycle (bus clock) delay in the clearing mechanism for the I bit so that, if interrupts were previously disabled, the next instruction after a CLI will always be executed, even if there was an interrupt pending prior to execution of the CLI instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	0	–	–	–	–

I: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CLI *translates to...* ANDCC #$EF	IMM	`10 EF`	`P`	`P`

CLR Clear Memory CLR

Operation: 0 ⇒ M

Description: All bits in memory location M are cleared to 0.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CLR *opr16a*	EXT	`79 hh ll`	`PwO`	`wOP`
CLR *oprx0_xysp*	IDX	`69 xb`	`Pw`	`Pw`
CLR *oprx9,xysp*	IDX1	`69 xb ff`	`PwO`	`PwO`
CLR *oprx16,xysp*	IDX2	`69 xb ee ff`	`PwP`	`PwP`
CLR [D,*xysp*]	[D,IDX]	`69 xb`	`PIfw`	`PIfPw`
CLR [*oprx16,xysp*]	[IDX2]	`69 xb ee ff`	`PIPw`	`PIPPw`

CLRA Clear A CLRA

Operation: 0 ⇒ A

Description: All bits in accumulator A are cleared to 0.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared
Z: 1; set
V: 0; cleared
C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CLRA	INH	87	O	O

CLRB Clear B CLRB

Operation: 0 ⇒ B

Description: All bits in accumulator B are cleared to 0.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared
Z: 1; set
V: 0; cleared
C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CLRB	INH	C7	O	O

CLV Clear Two’s Complement Overflow Bit CLV

Operation: 0 ⇒ V bit

Description: Clears the V status bit. This instruction is assembled as ANDCC #$FD. The ANDCC instruction can be used to clear any combination of bits in the CCR in one operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	0	–

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CLV *translates to...* ANDCC #$FD	IMM	`10 FD`	`P`	`P`

CMPA — Compare A — CMPA

Operation: (A) – (M)

Description: Compares the content of accumulator A to the content of memory location M and sets the condition codes, which may then be used for arithmetic and logical conditional branching. The contents of A and location M are not changed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if there was a borrow from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CMPA #*opr8i*	IMM	`81 ii`	`P`	`P`
CMPA *opr8a*	DIR	`91 dd`	`rPf`	`rfP`
CMPA *opr16a*	EXT	`B1 hh ll`	`rPO`	`rOP`
CMPA *oprx0_xysp*	IDX	`A1 xb`	`rPf`	`rfP`
CMPA *oprx9,xysp*	IDX1	`A1 xb ff`	`rPO`	`rPO`
CMPA *oprx16,xysp*	IDX2	`A1 xb ee ff`	`frPP`	`frPP`
CMPA [D,*xysp*]	[D,IDX]	`A1 xb`	`fIfrPf`	`fIfrfP`
CMPA [*oprx16*,*xysp*]	[IDX2]	`A1 xb ee ff`	`fIPrPf`	`fIPrfP`

CMPB **Compare B** CMPB

Operation: (B) – (M)

Description: Compares the content of accumulator B to the content of memory location M and sets the condition codes, which may then be used for arithmetic and logical conditional branching. The contents of B and location M are not changed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$
Set if there was a borrow from the MSB of the result; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CMPB *#opr8i*	IMM	`C1 ii`	`P`	`P`
CMPB *opr8a*	DIR	`D1 dd`	`rPf`	`rfP`
CMPB *opr16a*	EXT	`F1 hh ll`	`rPO`	`rOP`
CMPB *oprx0_xysp*	IDX	`E1 xb`	`rPf`	`rfP`
CMPB *oprx9,xysp*	IDX1	`E1 xb ff`	`rPO`	`rPO`
CMPB *oprx16,xysp*	IDX2	`E1 xb ee ff`	`frPP`	`frPP`
CMPB [D,*xysp*]	[D,IDX]	`E1 xb`	`fIfrPf`	`fIfrfP`
CMPB [*oprx16*,*xysp*]	[IDX2]	`E1 xb ee ff`	`fIPrPf`	`fIPrfP`

COM Complement Memory COM

Operation: $(\overline{M}) = \$FF - (M) \Rightarrow M$

Description: Replaces the content of memory location M with its one’s complement. Each bit of M is complemented. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two’s complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
COM *opr16a*	EXT	`71 hh ll`	`rPwO`	`rOPw`
COM *oprx0_xysp*	IDX	`61 xb`	`rPw`	`rPw`
COM *oprx9,xysp*	IDX1	`61 xb ff`	`rPwO`	`rPOw`
COM *oprx16,xysp*	IDX2	`61 xb ee ff`	`frPwP`	`frPPw`
COM [D,*xysp*]	[D,IDX]	`61 xb`	`fIfrPw`	`fIfrPw`
COM [*oprx16,xysp*]	[IDX2]	`61 xb ee ff`	`fIPrPw`	`fIPrPw`

COMA — Complement A — COMA

Operation: $(\overline{A}) = \$FF - (A) \Rightarrow A$

Description: Replaces the content of accumulator A with its one's complement. Each bit of A is complemented. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two's complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
COMA	INH	41	O	O

COMB Complement B COMB

Operation: $(\overline{B}) = \$FF - (B) \Rightarrow B$

Description: Replaces the content of accumulator B with its one’s complement. Each bit of B is complemented. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two’s complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
COMB	INH	51	O	O

CPD Compare Double Accumulator CPD

Operation: (A : B) – (M : M + 1)

Description: Compares the content of double accumulator D with a 16-bit value at the address specified and sets the condition codes accordingly. The compare is accomplished internally by a 16-bit subtract of (M : M + 1) from D without modifying either D or (M : M + 1).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the absolute value of the content of memory is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CPD #*opr16i*	IMM	`8C jj kk`	`PO`	`OP`
CPD *opr8a*	DIR	`9C dd`	`RPf`	`RfP`
CPD *opr16a*	EXT	`BC hh ll`	`RPO`	`ROP`
CPD *oprx0_xysp*	IDX	`AC xb`	`RPf`	`RfP`
CPD *oprx9,xysp*	IDX1	`AC xb ff`	`RPO`	`RPO`
CPD *oprx16,xysp*	IDX2	`AC xb ee ff`	`fRPP`	`fRPP`
CPD [D,*xysp*]	[D,IDX]	`AC xb`	`fIfRPf`	`fIfRfP`
CPD [*oprx16,xysp*]	[IDX2]	`AC xb ee ff`	`fIPRPf`	`fIPRfP`

CPS Compare Stack Pointer CPS

Operation: (SP) – (M : M + 1)

Description: Compares the content of the SP with a 16-bit value at the address specified, and sets the condition codes accordingly. The compare is accomplished internally by doing a 16-bit subtract of (M : M + 1) from the SP without modifying either the SP or (M : M + 1).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $S15 \bullet \overline{M15} \bullet \overline{R15} + \overline{S15} \bullet M15 \bullet R15$
Set if two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{S15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{S15}$
Set if the absolute value of the content of memory is larger than the absolute value of the SP; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
CPS #*opr16i*	IMM	`8F jj kk`	`PO`	`OP`
CPS *opr8a*	DIR	`9F dd`	`RPf`	`RfP`
CPS *opr16a*	EXT	`BF hh ll`	`RPO`	`ROP`
CPS *oprx0_xysp*	IDX	`AF xb`	`RPf`	`RfP`
CPS *oprx9,xysp*	IDX1	`AF xb ff`	`RPO`	`RPO`
CPS *oprx16,xysp*	IDX2	`AF xb ee ff`	`fRPP`	`fRPP`
CPS [D,*xysp*]	[D,IDX]	`AF xb`	`fIfRPf`	`fIfRfP`
CPS [*oprx16,xysp*]	[IDX2]	`AF xb ee ff`	`fIPRPf`	`fIPRfP`

CPX Compare Index Register X CPX

Operation: (X) – (M : M + 1)

Description: Compares the content of index register X with a 16-bit value at the address specified and sets the condition codes accordingly. The compare is accomplished internally by a 16-bit subtract of (M : M + 1) from index register X without modifying either index register X or (M : M + 1).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $X15 \bullet \overline{M15} \bullet \overline{R15} + \overline{X15} \bullet M15 \bullet R15$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{X15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{X15}$
Set if the absolute value of the content of memory is larger than the absolute value of the index register; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CPX #*opr16i*	IMM	`8E jj kk`	`PO`	`OP`
CPX *opr8a*	DIR	`9E dd`	`RPf`	`RfP`
CPX *opr16a*	EXT	`BE hh ll`	`RPO`	`ROP`
CPX *oprx0_xysp*	IDX	`AE xb`	`RPf`	`RfP`
CPX *oprx9,xysp*	IDX1	`AE xb ff`	`RPO`	`RPO`
CPX *oprx16,xysp*	IDX2	`AE xb ee ff`	`fRPP`	`fRPP`
CPX [D,*xysp*]	[D,IDX]	`AE xb`	`fIfRPf`	`fIfRfP`
CPX [*oprx16,xysp*]	[IDX2]	`AE xb ee ff`	`fIPRPf`	`fIPRfP`

CPY Compare Index Register Y CPY

Operation: (Y) – (M : M + 1)

Description: Compares the content of index register Y to a 16-bit value at the address specified and sets the condition codes accordingly. The compare is accomplished internally by a 16-bit subtract of (M : M + 1) from Y without modifying either Y or (M : M + 1).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $Y15 \bullet \overline{M15} \bullet \overline{R15} + \overline{Y15} \bullet M15 \bullet R15$
Set if two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{Y15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{Y15}$
Set if the absolute value of the content of memory is larger than the absolute value of the index register; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
CPY #*opr16i*	IMM	`8D jj kk`	`PO`	`OP`
CPY *opr8a*	DIR	`9D dd`	`RPf`	`RfP`
CPY *opr16a*	EXT	`BD hh ll`	`RPO`	`ROP`
CPY *oprx0_xysp*	IDX	`AD xb`	`RPf`	`RfP`
CPY *oprx9,xysp*	IDX1	`AD xb ff`	`RPO`	`RPO`
CPY *oprx16,xysp*	IDX2	`AD xb ee ff`	`fRPP`	`fRPP`
CPY [D,*xysp*]	[D,IDX]	`AD xb`	`fIfRPf`	`fIfRfP`
CPY [*oprx16,xysp*]	[IDX2]	`AD xb ee ff`	`fIPRPf`	`fIPRfP`

DAA Decimal Adjust A DAA

Description: DAA adjusts the content of accumulator A and the state of the C status bit to represent the correct binary-coded-decimal sum and the associated carry when a BCD calculation has been performed. To execute DAA, the content of accumulator A, the state of the C status bit, and the state of the H status bit must all be the result of performing an ABA, ADD, or ADC on BCD operands, with or without an initial carry.

The table shows DAA operation for all legal combinations of input operands. Columns 1 through 4 represent the results of ABA, ADC, or ADD operations on BCD operands. The correction factor in column 5 is added to the accumulator to restore the result of an operation on two BCD operands to a valid BCD value and to set or clear the C bit. All values are in hexadecimal.

1	2	3	4	5	6
Initial C Bit Value	**Value of A[7:4]**	**Initial H Bit Value**	**Value of A[3:0]**	**Correction Factor**	**Corrected C Bit Value**
0	0–9	0	0–9	00	0
0	0–8	0	A–F	06	0
0	0–9	1	0–3	06	0
0	A–F	0	0–9	60	1
0	9–F	0	A–F	66	1
0	A–F	1	0–3	66	1
1	0–2	0	0–9	60	1
1	0–2	0	A–F	66	1
1	0–3	1	0–3	66	1

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	?	Δ

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: Undefined
C: Represents BCD carry. See bit table

Source Form	Address Mode	Object Code	Access Detail HCS12	 M68HC12
DAA	INH	`18 07`	`OfO`	`OfO`

DBEQ Decrement and Branch if Equal to Zero DBEQ

Operation: (Counter) – 1 ⇒ Counter
If (Counter) = 0, then (PC) + $0003 + Rel ⇒ PC

Description: Subtract one from the specified counter register A, B, D, X, Y, or SP. If the counter register has reached zero, execute a branch to the specified relative destination. The DBEQ instruction is encoded into three bytes of machine code including the 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

IBEQ and TBEQ instructions are similar to DBEQ except that the counter is incremented or tested rather than being decremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
DBEQ *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBEQ.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	DBEQ A, *rel9*	`04 00 rr`	`04 10 rr`
B	001	DBEQ B, *rel9*	`04 01 rr`	`04 11 rr`
D	100	DBEQ D, *rel9*	`04 04 rr`	`04 14 rr`
X	101	DBEQ X, *rel9*	`04 05 rr`	`04 15 rr`
Y	110	DBEQ Y, *rel9*	`04 06 rr`	`04 16 rr`
SP	111	DBEQ SP, *rel9*	`04 07 rr`	`04 17 rr`

DBNE Decrement and Branch if Not Equal to Zero DBNE

Operation: (Counter) – 1 ⇒ Counter
If (Counter) not = 0, then (PC) + $0003 + Rel ⇒ PC

Description: Subtract one from the specified counter register A, B, D, X, Y, or SP. If the counter register has not been decremented to zero, execute a branch to the specified relative destination. The DBNE instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

IBNE and TBNE instructions are similar to DBNE except that the counter is incremented or tested rather than being decremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
DBNE *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	DBNE A, *rel9*	`04 20 rr`	`04 30 rr`
B	001	DBNE B, *rel9*	`04 21 rr`	`04 31 rr`
D	100	DBNE D, *rel9*	`04 24 rr`	`04 34 rr`
X	101	DBNE X, *rel9*	`04 25 rr`	`04 35 rr`
Y	110	DBNE Y, *rel9*	`04 26 rr`	`04 36 rr`
SP	111	DBNE SP, *rel9*	`04 27 rr`	`04 37 rr`

DEC Decrement Memory DEC

Operation: (M) – $01 ⇒ M

Description: Subtract one from the content of memory location M.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: Set if there was a two's complement overflow as a result of the operation; cleared otherwise. Two's complement overflow occurs if and only if (M) was $80 before the operation.

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
DEC *opr16a*	EXT	`73 hh ll`	`rPwO`	`rOPw`
DEC *oprx0_xysp*	IDX	`63 xb`	`rPw`	`rPw`
DEC *oprx9,xysp*	IDX1	`63 xb ff`	`rPwO`	`rPOw`
DEC *oprx16,xysp*	IDX2	`63 xb ee ff`	`frPwP`	`frPPw`
DEC [D,*xysp*]	[D,IDX]	`63 xb`	`fIfrPw`	`fIfrPw`
DEC [*oprx16,xysp*]	[IDX2]	`63 xb ee ff`	`fIPrPw`	`fIPrPw`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (DBEQ – 0) or not zero (DBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 would be 0:0 for DBNE.

DECA Decrement A DECA

Operation: (A) – \$01 ⇒ A

Description: Subtract one from the content of accumulator A.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: Set if there was a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (A) was \$80 before the operation.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
DECA	INH	43	O	O

DECB Decrement B DECB

Operation: (B) – \$01 ⇒ B

Description: Subtract one from the content of accumulator B.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: Set if there was a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (B) was \$80 before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
DECB	INH	53	O	O

DES Decrement Stack Pointer DES

Operation: (SP) – \$0001 ⇒ SP

Description: Subtract one from the SP. This instruction assembles to LEAS –1,SP. The LEAS instruction does not affect condition codes as DEX or DEY instructions do.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
DES *translates to...* LEAS –1,SP	IDX	`1B 9F`	`Pf`	`PP`[1]

[1] Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

DEX Decrement Index Register X DEX

Operation: (X) – \$0001 ⇒ X

Description: Subtract one from index register X. LEAX –1,X can produce the same result, but LEAX does not affect the Z bit. Although the LEAX instruction is more flexible, DEX requires only one byte of object code.

Only the Z bit is set or cleared according to the result of this operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is \$0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
DEX	INH	09	O	O

DEY Decrement Index Register Y DEY

Operation: (Y) – $0001 ⇒ Y

Description: Subtract one from index register Y. LEAY –1,Y can produce the same result, but LEAY does not affect the Z bit. Although the LEAY instruction is more flexible, DEY requires only one byte of object code.

Only the Z bit is set or cleared according to the result of this operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
DEY	INH	03	O	O

EDIV Extended Divide 32-Bit by 16-Bit (Unsigned) EDIV

Operation: (Y : D) ÷ (X) ⇒ Y; Remainder ⇒ D

Description: Divides a 32-bit unsigned dividend by a 16-bit divisor, producing a 16-bit unsigned quotient and an unsigned 16-bit remainder. All operands and results are located in CPU registers. If an attempt to divide by zero is made, C is set and the states of the N, Z, and V bits in the CCR are undefined. In case of an overflow or a divide by zero, the contents of the registers D and Y do not change.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	∆	∆	∆	∆

N: Set if MSB of result is set; cleared otherwise
Undefined after overflow or division by zero

Z: Set if result is $0000; cleared otherwise
Undefined after overflow or division by zero

V: Set if the result was > $FFFF; cleared otherwise Undefined after division by zero

C: Set if divisor was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EDIV	INH	11	ffffffffffO	ffffffffffO

EDIVS Extended Divide 32-Bit by 16-Bit (Signed) EDIVS

Operation: (Y : D) ÷ (X) ⇒ Y; Remainder ⇒ D

Description: Divides a signed 32-bit dividend by a 16-bit signed divisor, producing a signed 16-bit quotient and a signed 16-bit remainder. All operands and results are located in CPU registers. If an attempt to divide by zero is made, C is set and the states of the N, Z, and V bits in the CCR are undefined. In case of an overflow or a divide by zero, the contents of the registers D and Y do not change.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise
Undefined after overflow or division by zero

Z: Set if result is $0000; cleared otherwise
Undefined after overflow or division by zero

V: Set if the result was > $7FFF or < $8000; cleared otherwise
Undefined after division by zero

C: Set if divisor was $0000; cleared otherwise
Indicates division by zero

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EDIVS	INH	`18 14`	`Offfffffff fO`	`OffffffffffO`

EMACS Extended Multiply and Accumulate (Signed) 16-Bit by 16-Bit to 32-Bit EMACS

Operation: $(M_{(X)} : M_{(X+1)}) \times (M_{(Y)} : M_{(Y+1)}) + (M \sim M+3) \Rightarrow M \sim M+3$

Description: A 16-bit value is multiplied by a 16-bit value to produce a 32-bit intermediate result. This 32-bit intermediate result is then added to the content of a 32-bit accumulator in memory. EMACS is a signed integer operation. All operands and results are located in memory. When the EMACS instruction is executed, the first source operand is fetched from an address pointed to by X, and the second source operand is fetched from an address pointed to by index register Y. Before the instruction is executed, the X and Y index registers must contain values that point to the most significant bytes of the source operands. The most significant byte of the 32-bit result is specified by an extended address supplied with the instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00000000; cleared otherwise

V: $M31 \bullet I31 \bullet \overline{R31} + \overline{M31} \bullet \overline{I31} \bullet R31$
Set if result > $7FFFFFFF (+ overflow) or
< $80000000 (– underflow)
Indicates two’s complement overflow

C: $M15 \bullet I15 + I15 \bullet \overline{R15} + \overline{R15} \bullet M15$
Set if there was a carry from bit 15 of the result; cleared otherwise
Indicates a carry from low word to high word of the result occurred

Source Form[1]	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMACS *opr16a*	Special	`18 12 hh ll`	`ORROfffRRfWWP`	`ORROfffRRfWWP`

[1] *opr16a* is an extended address specification. Both X and Y point to source operands**.**

EMAXD — Place Larger of Two Unsigned 16-Bit Values in Accumulator D — EMAXD

Operation: MAX ((D), (M : M + 1)) ⇒ D

Description: Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger, and leaves the larger of the two values in D. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in D has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the largest value in a list of values.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = D – M : M + 1)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMAXD *oprx0_xysp*	IDX	18 1A xb	ORPf	ORfP
EMAXD *oprx9,xysp*	IDX1	18 1A xb ff	ORPO	ORPO
EMAXD *oprx16,xysp*	IDX2	18 1A xb ee ff	OfRPP	OfRPP
EMAXD [D,*xysp*]	[D,IDX]	18 1A xb	OfIfRPf	OfIfRfP
EMAXD [*oprx16,xysp*]	[IDX2]	18 1A xb ee ff	OfIPRPf	OfIPRfP

EMAXM — Place Larger of Two Unsigned 16-Bit Values in Memory — EMAXM

Operation: MAX ((D), (M : M + 1)) ⇒ M : M + 1

Description: Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger, and leaves the larger of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in D has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = D – M : M + 1)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMAXM *oprx0_xysp*	IDX	`18 1E xb`	`ORPW`	`ORPW`
EMAXM *oprx9,xysp*	IDX1	`18 1E xb ff`	`ORPWO`	`ORPWO`
EMAXM *oprx16,xysp*	IDX2	`18 1E xb ee ff`	`OfRPWP`	`OfRPWP`
EMAXM [D,*xysp*]	[D,IDX]	`18 1E xb`	`OfIfRPW`	`OfIfRPW`
EMAXM [*oprx16,xysp*]	[IDX2]	`18 1E xb ee ff`	`OfIPRPW`	`OfIPRPW`

EMIND Place Smaller of Two Unsigned 16-Bit Values in Accumulator D EMIND

Operation: MIN ((D), (M : M + 1)) ⇒ D

Description: Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger, and leaves the smaller of the two values in D. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in D has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the smallest value in a list of values.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = D – M : M + 1)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMIND *oprx0_xysp*	IDX	`18 1B xb`	`ORPf`	`ORfP`
EMIND *oprx9,xysp*	IDX1	`18 1B xb ff`	`ORPO`	`ORPO`
EMIND *oprx16,xysp*	IDX2	`18 1B xb ee ff`	`OfRPP`	`OfRPP`
EMIND [D,*xysp*]	[D,IDX]	`18 1B xb`	`OfIfRPf`	`OfIfRfP`
EMIND [*oprx16,xysp*]	[IDX2]	`18 1B xb ee ff`	`OfIPRPf`	`OfIPRfP`

EMINM Place Smaller of Two Unsigned 16-Bit Values in Memory EMINM

Operation: MIN ((D), (M : M + 1)) ⇒ M : M + 1

Description: Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger and leaves the smaller of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in D has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = D – M : M + 1)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMINM *oprx0_xysp*	IDX	18 1F xb	ORPW	ORPW
EMINM *oprx9,xysp*	IDX1	18 1F xb ff	ORPWO	ORPWO
EMINM *oprx16,xysp*	IDX2	18 1F xb ee ff	OfRPWP	OfRPWP
EMINM [D,*xysp*]	[D,IDX]	18 1F xb	OfIfRPW	OfIfRPW
EMINM [*oprx16,xysp*]	[IDX2]	18 1F xb ee ff	OfIPRPW	OfIPRPW

EMUL Extended Multiply 16-Bit by 16-Bit (Unsigned) EMUL

Operation: (D) × (Y) ⇒ Y : D

Description: An unsigned 16-bit value is multiplied by an unsigned 16-bit value to produce an unsigned 32-bit result. The first source operand must be loaded into 16-bit double accumulator D and the second source operand must be loaded into index register Y before executing the instruction. When the instruction is executed, the value in D is multiplied by the value in Y. The upper 16-bits of the 32-bit result are stored in Y and the low-order 16-bits of the result are stored in D.

The C status bit can be used to round the high-order 16 bits of the result.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

N: Set if the MSB of the result is set; cleared otherwise

Z: Set if result is $00000000; cleared otherwise

C: Set if bit 15 of the result is set; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMUL	INH	13	ffO	ffO

EMULS Extended Multiply 16-Bit by 16-Bit (Signed) EMULS

Operation: (D) × (Y) ⇒ Y : D

Description: A signed 16-bit value is multiplied by a signed 16-bit value to produce a signed 32-bit result. The first source operand must be loaded into 16-bit double accumulator D, and the second source operand must be loaded into index register Y before executing the instruction. When the instruction is executed, D is multiplied by the value Y. The 16 high-order bits of the 32-bit result are stored in Y and the 16 low-order bits of the result are stored in D.

The C status bit can be used to round the high-order 16 bits of the result.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

N: Set if the MSB of the result is set; cleared otherwise

Z: Set if result is $00000000; cleared otherwise

C: Set if bit 15 of the result is set; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EMULS	INH	18 13	OfO OffO[1]	OfO

[1] EMULS has an extra free cycle if it is followed by another PAGE TWO instruction.

EORA Exclusive OR A EORA

Operation: (A) ⊕ (M) ⇒ A

Description: Performs the logical exclusive OR between the content of accumulator A and the content of memory location M. The result is placed in A. Each bit of A after the operation is the logical exclusive OR of the corresponding bits of M and A before the operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EORA *#opr8i*	IMM	`88 ii`	`P`	`P`
EORA *opr8a*	DIR	`98 dd`	`rPf`	`rfP`
EORA *opr16a*	EXT	`B8 hh ll`	`rPO`	`rOP`
EORA *oprx0_xysp*	IDX	`A8 xb`	`rPf`	`rfP`
EORA *oprx9,xysp*	IDX1	`A8 xb ff`	`rPO`	`rPO`
EORA *oprx16,xysp*	IDX2	`A8 xb ee ff`	`frPP`	`frPP`
EORA [D,*xysp*]	[D,IDX]	`A8 xb`	`fIfrPf`	`fIfrfP`
EORA [*oprx16,xysp*]	[IDX2]	`A8 xb ee ff`	`fIPrPf`	`fIPrfP`

EORB Exclusive OR B EORB

Operation: (B) ⊕ (M) ⇒ B

Description: Performs the logical exclusive OR between the content of accumulator B and the content of memory location M. The result is placed in A. Each bit of A after the operation is the logical exclusive OR of the corresponding bits of M and B before the operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
EORB #*opr8i*	IMM	C8 ii	P	P
EORB *opr8a*	DIR	D8 dd	rPf	rfP
EORB *opr16a*	EXT	F8 hh ll	rPO	rOP
EORB *oprx0_xysp*	IDX	E8 xb	rPf	rfP
EORB *oprx9,xysp*	IDX1	E8 xb ff	rPO	rPO
EORB *oprx16,xysp*	IDX2	E8 xb ee ff	frPP	frPP
EORB [D,*xysp*]	[D,IDX]	E8 xb	fIfrPf	fIfrfP
EORB [*oprx16,xysp*]	[IDX2]	E8 xb ee ff	fIPrPf	fIPrfP

ETBL Extended Table Lookup and Interpolate ETBL

Operation: (M : M + 1) + [(B) × ((M + 2 : M + 3) – (M : M + 1))] ⇒ D

Description: ETBL linearly interpolates one of 256 result values that fall between each pair of data entries in a lookup table stored in memory. Data entries in the table represent the y values of endpoints of equally-spaced line segments. Table entries and the interpolated result are 16-bit values. The result is stored in the D accumulator.

Before executing ETBL, an index register points to the table entry corresponding to the x value (X1 that is closest to, but less than or equal to, the desired lookup point (XL, YL). This defines the left end of a line segment and the right end is defined by the next data entry in the table. Prior to execution, accumulator B holds a binary fraction (radix left of MSB) representing the ratio of (XL–X1) ÷ (X2–X1).

The 16-bit unrounded result is calculated using the following expression:

$$D = Y1 + [(B) \times (Y2 - Y1)]$$

Where:

(B) = (XL – X1) ÷ (X2 – X1)
Y1 = 16-bit data entry pointed to by <effective address>
Y2 = 16-bit data entry pointed to by <effective address> + 2

The intermediate value [(B) × (Y2 – Y1)] produces a 24-bit result with the radix point between bits 7 and 8. Any indexed addressing mode, except indirect modes or 9-bit and 16-bit offset modes, can be used to identify the first data point (X1,Y1). The second data point is the next table entry.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ[1]

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $0000; cleared otherwise
C: Set if result can be rounded up; cleared otherwise

[1] C-bit was undefined in original M68HC12

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ETBL *oprx0_xysp*	IDX	`18 3F xb`	`ORRffffffP`	`ORRffffffP`

EXG Exchange Register Contents EXG

Operation: See table

Description: Exchanges the contents of registers specified in the instruction as shown below. Note that the order in which exchanges between 8-bit and 16-bit registers are specified affects the high byte of the 16-bit registers differently. Exchanges of D with A or B are ambiguous. Cases involving TMP2 and TMP3 are reserved for Freescale use, so some assemblers may not permit their use, but it is possible to generate these cases by using DC.B or DC.W assembler directives.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Or:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

None affected, unless the CCR is the destination register. Condition codes take on the value of the corresponding source bits, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only in response to any reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
EXG *abcdxys,abcdxys*	INH	`B7 eb`	`P`	`P`

[1] Legal coding for `eb` is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit (bit 3 is a don't care). Values are in hexadecimal.

	8	9	A	B	C	D	E	F
0	A ⇔ A	B ⇔ A	CCR ⇔ A	TMP3$_L$ ⇒ A $00:A ⇒ TMP3	B ⇒ A A ⇒ B	X$_L$ ⇒ A $00:A ⇒ X	Y$_L$ ⇒ A $00:A ⇒ Y	SP$_L$ ⇒ A $00:A ⇒ SP
1	A ⇔ B	B ⇔ B	CCR ⇔ B	TMP3$_L$ ⇒ B $FF:B ⇒ TMP3	B ⇒ B $FF ⇒ A	X$_L$ ⇒ B $FF:B ⇒ X	Y$_L$ ⇒ B $FF:B ⇒ Y	SP$_L$ ⇒ B $FF:B ⇒ SP
2	A ⇔ CCR	B ⇔ CCR	CCR ⇔ CCR	TMP3$_L$ ⇒ CCR $FF:CCR ⇒ TMP3	B ⇒ CCR $FF:CCR ⇒ D	X$_L$ ⇒ CCR $FF:CCR ⇒ X	Y$_L$ ⇒ CCR $FF:CCR ⇒ Y	SP$_L$ ⇒ CCR $FF:CCR ⇒ SP
3	$00:A ⇒ TMP2 TMP2$_L$ ⇒ A	$00:B ⇒ TMP2 TMP2$_L$ ⇒ B	$00:CCR ⇒ TMP2 TMP2$_L$ ⇒ CCR	TMP3 ⇔ TMP2	D ⇔ TMP2	X ⇔ TMP2	Y ⇔ TMP2	SP ⇔ TMP2
4	$00:A ⇒ D	$00:B ⇒ D	$00:CCR ⇒ D B ⇒ CCR	TMP3 ⇔ D	D ⇔ D	X ⇔ D	Y ⇔ D	SP ⇔ D
5	$00:A ⇒ X X$_L$ ⇒ A	$00:B ⇒ X X$_L$ ⇒ B	$00:CCR ⇒ X X$_L$ ⇒ CCR	TMP3 ⇔ X	D ⇔ X	X ⇔ X	Y ⇔ X	SP ⇔ X
6	$00:A ⇒ Y Y$_L$ ⇒ A	$00:B ⇒ Y Y$_L$ ⇒ B	$00:CCR ⇒ Y Y$_L$ ⇒ CCR	TMP3 ⇔ Y	D ⇔ Y	X ⇔ Y	Y ⇔ Y	SP ⇔ Y
7	$00:A ⇒ SP SP$_L$ ⇒ A	$00:B ⇒ SP SP$_L$ ⇒ B	$00:CCR ⇒ SP SP$_L$ ⇒ CCR	TMP3 ⇔ SP	D ⇔ SP	X ⇔ SP	Y ⇔ SP	SP ⇔ SP

FDIV Fractional Divide FDIV

Operation: (D) ÷ (X) ⇒ X; Remainder ⇒ D

Description: Divides an unsigned 16-bit numerator in double accumulator D by an unsigned 16-bit denominator in index register X, producing an unsigned 16-bit quotient in X and an unsigned 16-bit remainder in D. If both the numerator and the denominator are assumed to have radix points in the same positions, the radix point of the quotient is to the left of bit 15. The numerator must be less than the denominator. In the case of overflow (denominator is less than or equal to the numerator) or division by zero, the quotient is set to $FFFF, and the remainder is indeterminate.

FDIV is equivalent to multiplying the numerator by 2^{16} and then performing 32 by 16-bit integer division. The result is interpreted as a binary-weighted fraction, which resulted from the division of a 16-bit integer by a larger 16-bit integer. A result of $0001 corresponds to 0.000015, and $FFFF corresponds to 0.9998. The remainder of an IDIV instruction can be resolved into a binary-weighted fraction by an FDIV instruction. The remainder of an FDIV instruction can be resolved into the next 16 bits of binary-weighted fraction by another FDIV instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	Δ	Δ

Z: Set if quotient is $0000; cleared otherwise

V: 1 if $X \leq D$
Set if the denominator was less than or equal to the numerator; cleared otherwise

C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet ... \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$
Set if denominator was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
FDIV	INH	18 11	OffffffffffO	OffffffffffO

IBEQ Increment and Branch if Equal to Zero IBEQ

Operation: (Counter) + 1 ⇒ Counter
If (Counter) = 0, then (PC) + \$0003 + Rel ⇒ PC

Description: Add one to the specified counter register A, B, D, X, Y, or SP. If the counter register has reached zero, branch to the specified relative destination. The IBEQ instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBEQ and TBEQ instructions are similar to IBEQ except that the counter is decremented or tested rather than being incremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
IBEQ *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

1. Encoding for `lb` is summarized in the following table. Bit 3 is not used (don’t care), bit 5 selects branch on zero (IBEQ – 0) or not zero (IBNE – 1) versions, and bit 0 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 1:0 for IBEQ.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	IBEQ A, *rel9*	`04 80 rr`	`04 90 rr`
B	001	IBEQ B, *rel9*	`04 81 rr`	`04 91 rr`
D	100	IBEQ D, *rel9*	`04 84 rr`	`04 94 rr`
X	101	IBEQ X, *rel9*	`04 85 rr`	`04 95 rr`
Y	110	IBEQ Y, *rel9*	`04 86 rr`	`04 96 rr`
SP	111	IBEQ SP, *rel9*	`04 87 rr`	`04 97 rr`

IBNE Increment and Branch if Not Equal to Zero IBNE

Operation: (Counter) + 1 ⇒ Counter
If (Counter) not = 0, then (PC) + $0003 + Rel ⇒ PC

Description: Add one to the specified counter register A, B, D, X, Y, or SP. If the counter register has not been incremented to zero, branch to the specified relative destination. The IBNE instruction is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBNE and TBNE instructions are similar to IBNE except that the counter is decremented or tested rather than being incremented. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
IBNE *abdxys, rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (IBEQ – 0) or not zero (IBNE – 1) versions, and bit 0 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 1:0 for IBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	IBNE A, *rel9*	`04 A0 rr`	`04 B0 rr`
B	001	IBNE B, *rel9*	`04 A1 rr`	`04 B1 rr`
D	100	IBNE D, *rel9*	`04 A4 rr`	`04 B4 rr`
X	101	IBNE X, *rel9*	`04 A5 rr`	`04 B5 rr`
Y	110	IBNE Y, *rel9*	`04 A6 rr`	`04 B6 rr`
SP	111	IBNE SP, *rel9*	`04 A7 rr`	`04 B7 rr`

IDIV Integer Divide IDIV

Operation: (D) ÷ (X) ⇒ X; Remainder ⇒ D

Description: Divides an unsigned 16-bit dividend in double accumulator D by an unsigned 16-bit divisor in index register X, producing an unsigned 16-bit quotient in X, and an unsigned 16-bit remainder in D. If both the divisor and the dividend are assumed to have radix points in the same positions, the radix point of the quotient is to the right of bit 0. In the case of division by zero, C is set, the quotient is set to $FFFF, and the remainder is indeterminate.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	0	Δ

Z: Set if quotient is $0000; cleared otherwise

V: 0; cleared

C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet \ldots \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$
Set if denominator was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
IDIV	INH	18 10	OffffffffffO	OffffffffffO

IDIVS Integer Divide (Signed) IDIVS

Operation: (D) ÷ (X) ⇒ X; Remainder ⇒ D

Description: Performs signed integer division of a signed 16-bit numerator in double accumulator D by a signed 16-bit denominator in index register X, producing a signed 16-bit quotient in X, and a signed 16-bit remainder in D. If division by zero is attempted, the values in D and X are not changed, C is set, and the values of the N, Z, and V status bits are undefined.

Other than division by zero, which is not legal and causes the C status bit to be set, the only overflow case is:

$$\frac{\$8000}{\$FFFF} = \frac{-32,768}{-1} = +32,768$$

But the highest positive value that can be represented in a 16-bit two’s complement number is 32,767 ($7FFF).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise
Undefined after overflow or division by zero

Z: Set if quotient is $0000; cleared otherwise
Undefined after overflow or division by zero

V: Set if the result was > $7FFF or < $8000; cleared otherwise
Undefined after division by zero

C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet ... \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$
Set if denominator was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
IDIVS	INH	18 15	OfffffffffO	OfffffffffO

INC Increment Memory INC

Operation: (M) + \$01 ⇒ M

Description: Add one to the content of memory location M.

The N, Z and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two’s complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: Set if there is a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (M) was \$7F before the operation.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
INC *opr16a*	EXT	`72 hh ll`	`rPwO`	`rOPw`
INC *oprx0_xysp*	IDX	`62 xb`	`rPw`	`rPw`
INC *oprx9,xysp*	IDX1	`62 xb ff`	`rPwO`	`rPOw`
INC *oprx16,xysp*	IDX2	`62 xb ee ff`	`frPwP`	`frPPw`
INC [D,*xysp*]	[D,IDX]	`62 xb`	`fIfrPw`	`fIfrPw`
INC [*oprx16,xysp*]	[IDX2]	`62 xb ee ff`	`fIPrPw`	`fIPrPw`

INCA Increment A INCA

Operation: (A) + $01 ⇒ A

Description: Add one to the content of accumulator A.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two’s complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: Set if there is a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (A) was $7F before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
INCA	INH	42	O	O

INCB Increment B INCB

Operation: (B) + $01 ⇒ B

Description: Add one to the content of accumulator B.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two’s complement values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: Set if there is a two’s complement overflow as a result of the operation; cleared otherwise. Two’s complement overflow occurs if and only if (B) was $7F before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
INCB	INH	52	O	O

INS Increment Stack Pointer INS

Operation: (SP) + $0001 ⇒ SP

Description: Add one to the SP. This instruction is assembled to LEAS 1,SP. The LEAS instruction does not affect condition codes as an INX or INY instruction would.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
INS *translates to...* LEAS 1,SP	IDX	`1B 81`	`Pf`	`PP`[1]

[1] Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

INX Increment Index Register X INX

Operation: (X) + $0001 ⇒ X

Description: Add one to index register X. LEAX 1,X can produce the same result but LEAX does not affect the Z status bit. Although the LEAX instruction is more flexible, INX requires only one byte of object code.

INX operation affects only the Z status bit.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
INX	INH	08	O	O

INY Increment Index Register Y INY

Operation: (Y) + $0001 ⇒ Y

Description: Add one to index register Y. LEAY 1,Y can produce the same result but LEAY does not affect the Z status bit. Although the LEAY instruction is more flexible, INY requires only one byte of object code.

INY operation affects only the Z status bit.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
INY	INH	02	O	O

JMP Jump JMP

Operation: Effective Address ⇒ PC

Description: Jumps to the instruction stored at the effective address. The effective address is obtained according to the rules for extended or indexed addressing.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
JMP *opr16a*	EXT	`06 hh ll`	`PPP`	`PPP`
JMP *oprx0_xysp*	IDX	`05 xb`	`PPP`	`PPP`
JMP *oprx9,xysp*	IDX1	`05 xb ff`	`PPP`	`PPP`
JMP *oprx16,xysp*	IDX2	`05 xb ee ff`	`fPPP`	`fPPP`
JMP [D,*xysp*]	[D,IDX]	`05 xb`	`fIfPPP`	`fIfPPP`
JMP [*oprx16,xysp*]	[IDX2]	`05 xb ee ff`	`fIfPPP`	`fIfPPP`

JSR Jump to Subroutine JSR

Operation: (SP) – \$0002 ⇒ SP
$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$
Subroutine Address ⇒ PC

Description: Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR as a return address.

Decrements the SP by two to allow the two bytes of the return address to be stacked.

Stacks the return address. The SP points to the high order byte of the return address.

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Jumps to the location determined by the effective address.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
JSR *opr8a*	DIR	`17 dd`	`SPPP`	`PPPS`
JSR *opr16a*	EXT	`16 hh ll`	`SPPP`	`PPPS`
JSR *oprx0_xysp*	IDX	`15 xb`	`PPPS`	`PPPS`
JSR *oprx9,xysp*	IDX1	`15 xb ff`	`PPPS`	`PPPS`
JSR *oprx16,xysp*	IDX2	`15 xb ee ff`	`fPPPS`	`fPPPS`
JSR [D,*xysp*]	[D,IDX]	`15 xb`	`fIfPPPS`	`fIfPPPS`
JSR [*oprx16,xysp*]	[IDX2]	`15 xb ee ff`	`fIfPPPS`	`fIfPPPS`

LBCC — Long Branch if Carry Cleared (Same as LBHS) — LBCC

Operation: If C = 0, then (PC) + $0004 + Rel ⇒ PC

Simple branch

Description: Tests the C status bit and branches if C = 0.

See 3.8 Relative Addressing Mode for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LBCC *rel16*	REL	`18 24 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBCS — Long Branch if Carry Set (Same as LBLO) — LBCS

Operation: If C = 1, then (PC) + \$0004 + Rel ⇒ PC

Simple branch

Description: Tests the C status bit and branches if C = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBCS *rel16*	REL	`18 25 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBEQ Long Branch if Equal LBEQ

Operation: If Z = 1, (PC) + \$0004 + Rel ⇒ PC

Simple branch

Description: Tests the Z status bit and branches if Z = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LBEQ *rel16*	REL	`18 27 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBGE Long Branch if Greater Than or Equal to Zero LBGE

Operation: If $N \oplus V = 0$, $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two’s complement numbers, if (Accumulator) ≥ Memory), then branch

Description: LBGE can be used to branch after subtracting or comparing signed two’s complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBGE *rel16*	REL	`18 2C qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBGT Long Branch if Greater Than Zero LBGT

Operation: If Z + (N ⊕ V) = 0, then (PC) + $0004 + Rel ⇒ PC

For signed two's complement numbers, If (Accumulator) > (Memory), then branch

Description: LBGT can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBGT *rel16*	REL	`18 2E qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBHI Long Branch if Higher LBHI

Operation: If $C + Z = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if (Accumulator) > (Memory), then branch

Description: LBHI can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than the value in M. After CBA or SBA, the branch occurs if the value in B is greater than the value in A. LBHI should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBHI *rel16*	REL	`18 22 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	$Z + (N \oplus V) = 0$	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	$N \oplus V = 0$	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	$Z = 1$	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	$Z + (N \oplus V) = 1$	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	$N \oplus V = 1$	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	$C + Z = 0$	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	$C = 0$	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	$Z = 1$	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	$C + Z = 1$	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	$C = 1$	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	$Z = 1$	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBHS Long Branch if Higher or Same (Same as LBCC) LBHS

Operation: If C = 0, then (PC) + $0004 + Rel ⇒ PC

For unsigned binary numbers, if (Accumulator) ≥ (Memory), then branch

Description: LBHS can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A. LBHS should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBHS *rel16*	REL	`18 24 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLE Long Branch if Less Than or Equal to Zero LBLE

Operation: If Z + (N ⊕ V) = 1, then (PC) + $0004 + Rel ⇒ PC

For signed two's complement numbers, if (Accumulator) ≤ (Memory), then branch.

Description: LBLE can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is less than or equal to the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBLE *rel16*	REL	`18 2F qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLO Long Branch if Lower (Same as LBCS) LBLO

Operation: If C = 1, then (PC) + $0004 + Rel ⇒ PC

For unsigned binary numbers, if (Accumulator) < (Memory), then branch

Description: LBLO can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A. LBLO should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBLO *rel16*	REL	`18 25 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLS Long Branch if Lower or Same LBLS

Operation: If $C + Z = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if (Accumulator) ≤ (Memory), then branch

Description: LBLS can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is less than or equal to the value in A. LBLS should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBLS *rel16*	REL	`18 23 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLT Long Branch if Less Than Zero LBLT

Operation: If N ⊕ V = 1, (PC) + \$0004 + Rel ⇒ PC

For signed two's complement numbers, if (Accumulator) < (Memory), then branch

Description: LBLT can be used to branch after subtracting or comparing signed two-s complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBLT *rel16*	REL	`18 2D qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBMI Long Branch if Minus LBMI

Operation: If N = 1, then (PC) + $0004 + Rel ⇒ PC

Simple branch

Description: Tests the N status bit and branches if N = 1.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LBMI *rel16*	REL	`18 2B qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBNE Long Branch if Not Equal to Zero LBNE

Operation: If Z = 0, then (PC) + \$0004 + Rel ⇒ PC

Simple branch

Description: Tests the Z status bit and branches if Z = 0.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LBNE *rel16*	REL	`18 26 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBPL Long Branch if Plus LBPL

Operation: If N = 0, then (PC) + $0004 + Rel ⇒ PC

Simple branch

Description: Tests the N status bit and branches if N = 0.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LBPL *rel16*	REL	`18 2A qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBRA Long Branch Always LBRA

Operation: (PC) + $0004 + Rel ⇒ PC

Description: Unconditional branch to an address calculated as shown in the expression. Rel is a relative offset stored as a two’s complement number in the second and third bytes of machine code corresponding to the long branch instruction.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the LBRA branch condition is always satisfied, the branch is always taken, and the instruction queue must always be refilled, so execution time is always the larger value.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBRA *rel16*	REL	`18 20 qq rr`	`OPPP`	`OPPP`

LBRN Long Branch Never LBRN

Operation: (PC) + $0004 ⇒ PC

Description: Never branches. LBRN is effectively a 4-byte NOP that requires three cycles to execute. LBRN is included in the instruction set to provide a complement to the LBRA instruction. The instruction is useful during program debug, to negate the effect of another branch instruction without disturbing the offset byte. A complement for LBRA is also useful in compiler implementations.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the LBRN branch condition is never satisfied, the branch is never taken, and the queue does not need to be refilled, so execution time is always the smaller value.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBRN *rel16*	REL	`18 21 qq rr`	`OPO`	`OPO`

LBVC Long Branch if Overflow Cleared LBVC

Operation: If V = 0, then (PC) + $0004 + Rel ⇒ PC

Simple branch

Description: Tests the V status bit and branches if V = 0.

LBVC causes a branch when a previous operation on two's complement binary values does not cause an overflow. That is, when LBVC follows a two's complement operation, a branch occurs when the result of the operation is valid.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBVC *rel16*	REL	`18 28 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBVS Long Branch if Overflow Set LBVS

Operation: If V = 1, then (PC) + $0004 + Rel ⇒ PC

Simple branch

Description: Tests the V status bit and branches if V = 1.

LBVS causes a branch when a previous operation on two's complement binary values causes an overflow. That is, when LBVS follows a two's complement operation, a branch occurs when the result of the operation is invalid.

See **3.8 Relative Addressing Mode** for details of branch execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LBVS *rel16*	REL	`18 29 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]

[1] OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
r>m	LBGT	18 2E	Z + (N ⊕ V) = 0	r≤m	LBLE	18 2F	Signed
r≥m	LBGE	18 2C	N ⊕ V = 0	r<m	LBLT	18 2D	Signed
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Signed
r≤m	LBLE	18 2F	Z + (N ⊕ V) = 1	r>m	LBGT	18 2E	Signed
r<m	LBLT	18 2D	N ⊕ V = 1	r≥m	LBGE	18 2C	Signed
r>m	LBHI	18 22	C + Z = 0	r≤m	LBLS	18 23	Unsigned
r≥m	LBHS/LBCC	18 24	C = 0	r<m	LBLO/LBCS	18 25	Unsigned
r=m	LBEQ	18 27	Z = 1	r≠m	LBNE	18 26	Unsigned
r≤m	LBLS	18 23	C + Z = 1	r>m	LBHI	18 22	Unsigned
r<m	LBLO/LBCS	18 25	C = 1	r≥m	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	C = 1	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	N = 1	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	V = 1	No Overflow	LBVC	18 28	Simple
r=0	LBEQ	18 27	Z = 1	r≠0	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LDAA Load Accumulator A LDAA

Operation: (M) ⇒ A

Description: Loads the content of memory location M into accumulator A. The condition codes are set according to the data.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	∆	∆	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LDAA #*opr8i*	IMM	`86 ii`	`P`	`P`
LDAA *opr8a*	DIR	`96 dd`	`rPf`	`rfP`
LDAA *opr16a*	EXT	`B6 hh ll`	`rPO`	`rOP`
LDAA *oprx0_xysp*	IDX	`A6 xb`	`rPf`	`rfP`
LDAA *oprx9,xysp*	IDX1	`A6 xb ff`	`rPO`	`rPO`
LDAA *oprx16,xysp*	IDX2	`A6 xb ee ff`	`frPP`	`frPP`
LDAA [D,*xysp*]	[D,IDX]	`A6 xb`	`fIfrPf`	`fIfrfP`
LDAA [*oprx16,xysp*]	[IDX2]	`A6 xb ee ff`	`fIPrPf`	`fIPrfP`

LDAB Load Accumulator B LDAB

Operation: (M) ⇒ B

Description: Loads the content of memory location M into accumulator B. The condition codes are set according to the data.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LDAB #*opr8i*	IMM	`C6 ii`	`P`	`P`
LDAB *opr8a*	DIR	`D6 dd`	`rPf`	`rfP`
LDAB *opr16a*	EXT	`F6 hh ll`	`rPO`	`rOP`
LDAB *oprx0_xysp*	IDX	`E6 xb`	`rPf`	`rfP`
LDAB *oprx9*,*xysp*	IDX1	`E6 xb ff`	`rPO`	`rPO`
LDAB *oprx16*,*xysp*	IDX2	`E6 xb ee ff`	`frPP`	`frPP`
LDAB [D,*xysp*]	[D,IDX]	`E6 xb`	`fIfrPf`	`fIfrfP`
LDAB [*oprx16*,*xysp*]	[IDX2]	`E6 xb ee ff`	`fIPrPf`	`fIPrfP`

LDD Load Double Accumulator LDD

Operation: (M : M+1) ⇒ A : B

Description: Loads the contents of memory locations M and M+1 into double accumulator D. The condition codes are set according to the data. The information from M is loaded into accumulator A, and the information from M+1 is loaded into accumulator B.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LDD #*opr16i*	IMM	`CC jj kk`	`PO`	`OP`
LDD *opr8a*	DIR	`DC dd`	`RPf`	`RfP`
LDD *opr16a*	EXT	`FC hh ll`	`RPO`	`ROP`
LDD *oprx0_xysp*	IDX	`EC xb`	`RPf`	`RfP`
LDD *oprx9,xysp*	IDX1	`EC xb ff`	`RPO`	`RPO`
LDD *oprx16,xysp*	IDX2	`EC xb ee ff`	`fRPP`	`fRPP`
LDD [D,*xysp*]	[D,IDX]	`EC xb`	`fIfRPf`	`fIfRfP`
LDD [*oprx16,xysp*]	[IDX2]	`EC xb ee ff`	`fIPRPf`	`fIPRfP`

LDS Load Stack Pointer LDS

Operation: (M : M+1) ⇒ SP

Description: Loads the most significant byte of the SP with the content of memory location M, and loads the least significant byte of the SP with the content of the next byte of memory at M+1.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LDS #*opr16i*	IMM	`CF jj kk`	`PO`	`OP`
LDS *opr8a*	DIR	`DF dd`	`RPf`	`RfP`
LDS *opr16a*	EXT	`FF hh ll`	`RPO`	`ROP`
LDS *oprx0_xysp*	IDX	`EF xb`	`RPf`	`RfP`
LDS *oprx9,xysp*	IDX1	`EF xb ff`	`RPO`	`RPO`
LDS *oprx16,xysp*	IDX2	`EF xb ee ff`	`fRPP`	`fRPP`
LDS [D,*xysp*]	[D,IDX]	`EF xb`	`fIfRPf`	`fIfRfP`
LDS [*oprx16,xysp*]	[IDX2]	`EF xb ee ff`	`fIPRPf`	`fIPRfP`

LDX Load Index Register X LDX

Operation: (M : M+1) ⇒ X

Description: Loads the most significant byte of index register X with the content of memory location M, and loads the least significant byte of X with the content of the next byte of memory at M+1.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LDX #*opr16i*	IMM	`CE jj kk`	`PO`	`OP`
LDX *opr8a*	DIR	`DE dd`	`RPf`	`RfP`
LDX *opr16a*	EXT	`FE hh ll`	`RPO`	`ROP`
LDX *oprx0_xysp*	IDX	`EE xb`	`RPf`	`RfP`
LDX *oprx9,xysp*	IDX1	`EE xb ff`	`RPO`	`RPO`
LDX *oprx16,xysp*	IDX2	`EE xb ee ff`	`fRPP`	`fRPP`
LDX [D,*xysp*]	[D,IDX]	`EE xb`	`fIfRPf`	`fIfRfP`
LDX [*oprx16,xysp*]	[IDX2]	`EE xb ee ff`	`fIPRPf`	`fIPRfP`

LDY Load Index Register Y LDY

Operation: (M : M+1) ⇒ Y

Description: Loads the most significant byte of index register Y with the content of memory location M, and loads the least significant byte of Y with the content of the next memory location at M+1.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LDY #*opr16i*	IMM	`CD jj kk`	`PO`	`OP`
LDY *opr8a*	DIR	`DD dd`	`RPf`	`RfP`
LDY *opr16a*	EXT	`FD hh ll`	`RPO`	`ROP`
LDY *oprx0_xysp*	IDX	`ED xb`	`RPf`	`RfP`
LDY *oprx9,xysp*	IDX1	`ED xb ff`	`RPO`	`RPO`
LDY *oprx16,xysp*	IDX2	`ED xb ee ff`	`fRPP`	`fRPP`
LDY [D,*xysp*]	[D,IDX]	`ED xb`	`fIfRPf`	`fIfRfP`
LDY [*oprx16,xysp*]	[IDX2]	`ED xb ee ff`	`fIPRPf`	`fIPRfP`

LEAS Load Stack Pointer with Effective Address LEAS

Operation: Effective Address ⇒ SP

Description: Loads the stack pointer with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See **3.9 Indexed Addressing Modes** for more details.

LEAS does not alter condition code bits. This allows stack modification without disturbing CCR bits changed by recent arithmetic operations.

Operation is a bit more complex when LEAS is used with auto-increment or auto-decrement operand specifications and the SP is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load index instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAS involves two different index registers and post-increment or post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAS 4,Y+. First S is loaded with the value of Y, then Y is incremented by 4.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LEAS *oprx0_xysp*	IDX	`1B xb`	`Pf`	`PP`[1]
LEAS *oprx9,xysp*	IDX1	`1B xb ff`	`PO`	`PO`
LEAS *oprx16,xysp*	IDX2	`1B xb ee ff`	`PP`	`PP`

[1] Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

LEAX Load X with Effective Address LEAX

Operation: Effective Address ⇒ X

Description: Loads index register X with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See **3.9 Indexed Addressing Modes** for more details.

Operation is a bit more complex when LEAX is used with auto-increment or auto-decrement operand specifications and index register X is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load indexed instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAX involves two different index registers and post-increment and post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAX 4,Y+. First X is loaded with the value of Y, then Y is incremented by 4.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LEAX *oprx0_xysp*	IDX	`1A xb`	`Pf`	`PP`[1]
LEAX *oprx9,xysp*	IDX1	`1A xb ff`	`PO`	`PO`
LEAX *oprx16,xysp*	IDX2	`1A xb ee ff`	`PP`	`PP`

[1] Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

LEAY Load Y with Effective Address LEAY

Operation: Effective Address ⇒ Y

Description: Loads index register Y with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See **3.9 Indexed Addressing Modes** for more details.

Operation is a bit more complex when LEAY is used with auto-increment or auto-decrement operand specifications and index register Y is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load indexed instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAY involves two different index registers and post-increment and post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAY 4,X+. First Y is loaded with the value of X, then X is incremented by 4.

CCR Details:

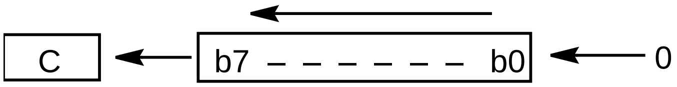
S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
LEAY *oprx0_xysp*	IDX	`19 xb`	`Pf`	`PP`[1]
LEAY *oprx9,xysp*	IDX1	`19 xb ff`	`PO`	`PO`
LEAY *oprx16,xysp*	IDX2	`19 xb ee ff`	`PP`	`PP`

1. Due to internal M68HC12 CPU requirements, the program word fetch is performed twice to the same address during this instruction.

LSL Logical Shift Left Memory (Same as ASL) LSL

Operation:



Description: Shifts all bits of the memory location M one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of M.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

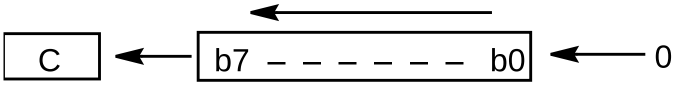
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M7
Set if the LSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSL *opr16a*	EXT	`78 hh ll`	`rPwO`	`rOPw`
LSL *oprx0_xysp*	IDX	`68 xb`	`rPw`	`rPw`
LSL *oprx9,xysp*	IDX1	`68 xb ff`	`rPwO`	`rPOw`
LSL *oprx16,xysp*	IDX2	`68 xb ee ff`	`frPwP`	`frPPw`
LSL [D,*xysp*]	[D,IDX]	`68 xb`	`fIfrPw`	`fIfrPw`
LSL [*oprx16,xysp*]	[IDX2]	`68 xb ee ff`	`fIPrPw`	`fIPrPw`

LSLA Logical Shift Left A (Same as ASLA) LSLA

Operation:



Description: Shifts all bits of accumulator A one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of A.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

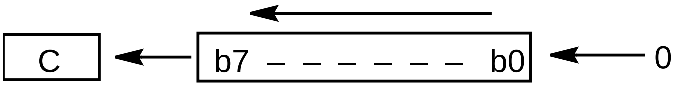
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A7
Set if the LSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSLA	INH	48	O	O

LSLB Logical Shift Left B (Same as ASLB) LSLB

Operation:



Description: Shifts all bits of accumulator B one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of B.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B7
Set if the LSB of B was set before the shift; cleared otherwise

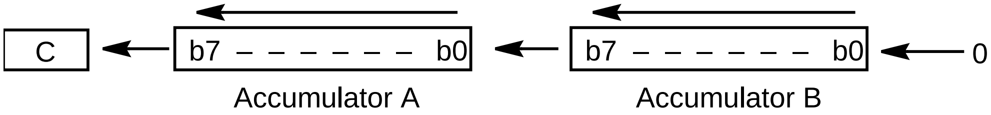
Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSLB	INH	58	O	O

LSLD

**Logical Shift Left Double
(Same as ASLD)**

LSLD

Operation:



Description: Shifts all bits of double accumulator D one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of accumulator A.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

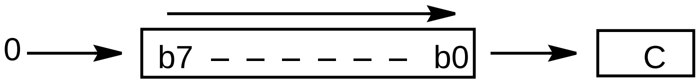
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: D15
Set if the MSB of D was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSLD	INH	59	O	O

LSR Logical Shift Right Memory LSR

Operation:



Description: Shifts all bits of memory location M one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of M.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is $00; cleared otherwise

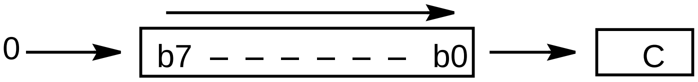
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSR *opr16a*	EXT	`74 hh ll`	`rPwO`	`rOPw`
LSR *oprx0_xysp*	IDX	`64 xb`	`rPw`	`rPw`
LSR *oprx9,xysp*	IDX1	`64 xb ff`	`rPwO`	`rPOw`
LSR *oprx16,xysp*	IDX2	`64 xb ee ff`	`frPwP`	`frPPw`
LSR [D,*xysp*]	[D,IDX]	`64 xb`	`fIfrPw`	`fIfrPw`
LSR [*oprx16,xysp*]	[IDX2]	`64 xb ee ff`	`fIPrPw`	`fIPrPw`

LSRA Logical Shift Right A LSRA

Operation:



Description: Shifts all bits of accumulator A one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of A.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$00; cleared otherwise

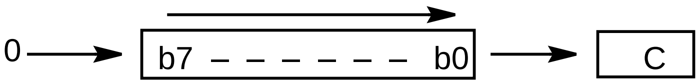
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A0
Set if the LSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSRA	INH	44	O	O

LSRB Logical Shift Right B LSRB

Operation:



Description: Shifts all bits of accumulator B one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of B.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$00; cleared otherwise

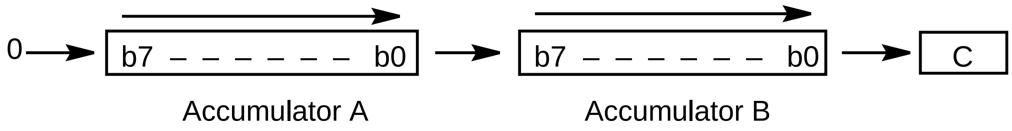
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B0
Set if the LSB of B was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSRB	INH	54	O	O

LSRD Logical Shift Right Double LSRD

Operation:



Description: Shifts all bits of double accumulator D one place to the right. D15 (MSB of A) is loaded with 0. The C status bit is loaded from D0 (LSB of B).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is $0000; cleared otherwise

V: D0
Set if, after the shift operation, C is set; cleared otherwise

C: D0
Set if the LSB of D was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
LSRD	INH	49	O	O

MAXA Place Larger of Two Unsigned 8-Bit Values in Accumulator A MAXA

Operation: MAX ((A), (M)) ⇒ A

Description: Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the larger of the two values in A. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in A has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the largest value in a list of values.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = A – M)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MAXA *oprx0_xysp*	IDX	`18 18 xb`	`OrPf`	`OrfP`
MAXA *oprx9,xysp*	IDX1	`18 18 xb ff`	`OrPO`	`OrPO`
MAXA *oprx16,xysp*	IDX2	`18 18 xb ee ff`	`OfrPP`	`OfrPP`
MAXA [D,*xysp*]	[D,IDX]	`18 18 xb`	`OfIfrPf`	`OfIfrfP`
MAXA [*oprx16,xysp*]	[IDX2]	`18 18 xb ee ff`	`OfIPrPf`	`OfIPrfP`

MAXM Place Larger of Two Unsigned 8-Bit Values in Memory MAXM

Operation: MAX ((A), (M)) ⇒ M

Description: Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the larger of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in accumulator A has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = A – M)

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
MAXM *oprx0_xysp*	IDX	`18 1C xb`	`OrPw`	`OrPw`
MAXM *oprx9,xysp*	IDX1	`18 1C xb ff`	`OrPwO`	`OrPwO`
MAXM *oprx16,xysp*	IDX2	`18 1C xb ee ff`	`OfrPwP`	`OfrPwP`
MAXM [D,*xysp*]	[D,IDX]	`18 1C xb`	`OfIfrPw`	`OfIfrPw`
MAXM [*oprx16,xysp*]	[IDX2]	`18 1C xb ee ff`	`OfIPrPw`	`OfIPrPw`

MEM Determine Grade of Membership (Fuzzy Logic) MEM

Operation: Grade of Membership ⇒ $M_{(Y)}$
(Y) + $0001 ⇒ Y
(X) + $0004 ⇒ X

Description: Before executing MEM, initialize A, X, and Y. Load A with the current crisp value of a system input variable. Load Y with the fuzzy input RAM location where the grade of membership is to be stored. Load X with the first address of a 4-byte data structure that describes a trapezoidal membership function. The data structure consists of:

- Point_1 — The x-axis starting point for the leading side (at M_X)
- Point_2 — The x-axis position of the rightmost point (at M_{X+1})
- Slope_1 — The slope of the leading side (at M_{X+2})
- Slope_2 — The slope of the trailing side (at M_{X+3}); the right side slopes up and to the left from point_2

A slope_1 or slope_2 value of $00 is a special case in which the membership function either starts with a grade of $FF at input = point_1, or ends with a grade of $FF at input = point_2 (infinite slope).

During execution, the value of A remains unchanged. X is incremented by four and Y is incremented by one.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	?	?

H, N, Z, V, and C may be altered by this instruction.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MEM	Special	`01`	`RRfOw`	`RRfOw`

MINA

Place Smaller of Two Unsigned 8-Bit Values in Accumulator A

MINA

Operation: MIN ((A), (M)) ⇒ A

Description: Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger, and leaves the smaller of the two values in accumulator A. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in accumulator A has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the smallest value in a list of values.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = A – M)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MINA *oprx0_xysp*	IDX	`18 19 xb`	`OrPf`	`OrfP`
MINA *oprx9,xysp*	IDX1	`18 19 xb ff`	`OrPO`	`OrPO`
MINA *oprx16,xysp*	IDX2	`18 19 xb ee ff`	`OfrPP`	`OfrPP`
MINA [D,*xysp*]	[D,IDX]	`18 19 xb`	`OfIfrPf`	`OfIfrfP`
MINA [*oprx16,xysp*]	[IDX2]	`18 19 xb ee ff`	`OfIPrPf`	`OfIPrfP`

MINM — Place Smaller of Two Unsigned 8-Bit Values in Memory — MINM

Operation: MIN ((A), (M)) ⇒ M

Description: Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the smaller of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in accumulator A has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction (R = A – M)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MINM *oprx0_xysp*	IDX	`18 1D xb`	`OrPw`	`OrPw`
MINM *oprx9,xysp*	IDX1	`18 1D xb ff`	`OrPwO`	`OrPwO`
MINM *oprx16,xysp*	IDX2	`18 1D xb ee ff`	`OfrPwP`	`OfrPwP`
MINM [D,*xysp*]	[D,IDX]	`18 1D xb`	`OfIfrPw`	`OfIfrPw`
MINM [*oprx16,xysp*]	[IDX2]	`18 1D xb ee ff`	`OfIPrPw`	`OfIPrPw`

MOVB — Move a Byte of Data from One Memory Location to Another — MOVB

Operation: $(M_1) \Rightarrow M_2$

Description: Moves the content of one memory location to another memory location. The content of the source memory location is not changed.

Move instructions use separate addressing modes to access the source and destination of a move. The following combinations of addressing modes are supported: IMM–EXT, IMM–IDX, EXT–EXT, EXT–IDX, IDX–EXT, and IDX–IDX. IDX operands allow indexed addressing mode specifications that fit in a single postbyte including 5-bit constant, accumulator offsets, and auto increment/decrement modes. Nine-bit and 16-bit constant offsets would require additional extension bytes and are not allowed. Indexed indirect modes (for example [D,r]) are also not allowed.

There are special considerations when using PC-relative addressing with move instructions. These are discussed in **3.10 Instructions Using Multiple Modes**.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form[1]	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MOVB #*opr8*, *opr16a*	IMM–EXT	`18 0B ii hh ll`	`OPwP`	`OPwP`
MOVB #*opr8i, oprx0_xysp*	IMM–IDX	`18 08 xb ii`	`OPwO`	`OPwO`
MOVB *opr16a, opr16a*	EXT–EXT	`18 0C hh ll hh ll`	`OrPwPO`	`OrPwPO`
MOVB *opr16a, oprx0_xysp*	EXT–IDX	`18 09 xb hh ll`	`OPrPw`	`OPrPw`
MOVB *oprx0_xysp, opr16a*	IDX–EXT	`18 0D xb hh ll`	`OrPwP`	`OrPwP`
MOVB *oprx0_xysp, oprx0_xysp*	IDX–IDX	`18 0A xb xb`	`OrPwO`	`OrPwO`

[1] The first operand in the source code statement specifies the source for the move.

MOVW — Move a Word of Data from One Memory Location to Another — MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$

Description: Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed.

Move instructions use separate addressing modes to access the source and destination of a move. The following combinations of addressing modes are supported: IMM–EXT, IMM–IDX, EXT–EXT, EXT–IDX, IDX–EXT, and IDX–IDX. IDX operands allow indexed addressing mode specifications that fit in a single postbyte including 5-bit constant, accumulator offsets, and auto increment/decrement modes. Nine-bit and 16-bit constant offsets would require additional extension bytes and are not allowed. Indexed indirect modes (for example [D,r]) are also not allowed.

There are special considerations when using PC-relative addressing with move instructions. These are discussed in **3.10 Instructions Using Multiple Modes**.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form[1]	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MOVW #*opr16i, opr16a*	IMM–EXT	`18 03 jj kk hh ll`	`OPWPO`	`OPWPO`
MOVW #*opr16i, oprx0_xysp*	IMM–IDX	`18 00 xb jj kk`	`OPPW`	`OPPW`
MOVW *opr16a, opr16a*	EXT–EXT	`18 04 hh ll hh ll`	`ORPWPO`	`ORPWPO`
MOVW *opr16a, oprx0_xysp*	EXT–IDX	`18 01 xb hh ll`	`OPRPW`	`OPRPW`
MOVW *oprx0_xysp, opr16a*	IDX–EXT	`18 05 xb hh ll`	`ORPWP`	`ORPWP`
MOVW *oprx0_xysp, oprx0_xysp*	IDX–IDX	`18 02 xb xb`	`ORPWO`	`ORPWO`

[1] The first operand in the source code statement specifies the source for the move.

MUL Multiply 8-Bit by 8-Bit (Unsigned) MUL

Operation: (A) × (B) ⇒ A : B

Description: Multiplies the 8-bit unsigned binary value in accumulator A by the 8-bit unsigned binary value in accumulator B and places the 16-bit unsigned result in double accumulator D. The carry flag allows rounding the most significant byte of the result through the sequence MUL, ADCA #0.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	Δ

C: R7
Set if bit 7 of the result (B bit 7) is set; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
MUL	INH	12	O	ffO

NEG Negate Memory NEG

Operation: $0 - (M) = (\overline{M}) + 1 \Rightarrow M$

Description: Replaces the content of memory location M with its two’s complement (the value $80 is left unchanged).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise.

Z: Set if result is $00; cleared otherwise.

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if there is a two’s complement overflow from the implied subtraction from zero; cleared otherwise. Two’s complement overflow occurs if and only if (M) = $80

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when (M) = $00.

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
NEG *opr16a*	EXT	`70 hh ll`	`rPwO`	`rOPw`
NEG *oprx0_xysp*	IDX	`60 xb`	`rPw`	`rPw`
NEG *oprx9,xysp*	IDX1	`60 xb ff`	`rPwO`	`rPOw`
NEG *oprx16,xysp*	IDX2	`60 xb ee ff`	`frPwP`	`frPPw`
NEG [D,*xysp*]	[D,IDX]	`60 xb`	`fIfrPw`	`fIfrPw`
NEG [*oprx16,xysp*]	[IDX2]	`60 xb ee ff`	`fIPrPw`	`fIPrPw`

NEGA Negate A NEGA

Operation: $0 - (A) = (\overline{A}) + 1 \Rightarrow A$

Description: Replaces the content of accumulator A with its two's complement (the value $80 is left unchanged).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise
Two's complement overflow occurs if and only if (A) = $80

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise
Set in all cases except when (A) = $00

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
NEGA	INH	40	O	O

NEGB Negate B NEGB

Operation: $0 - (B) = (\overline{B}) + 1 \Rightarrow B$

Description: Replaces the content of accumulator B with its two’s complement (the value $80 is left unchanged).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if there is a two’s complement overflow from the implied subtraction from zero; cleared otherwise
Two’s complement overflow occurs if and only if (B) = $80

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise
Set in all cases except when (B) = $00

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
NEGB	INH	50	O	O

NOP Null Operation NOP

Operation: No operation

Description: This single-byte instruction increments the PC and does nothing else. No other CPU registers are affected. NOP is typically used to produce a time delay, although some software disciplines discourage CPU frequency-based time delays. During debug, NOP instructions are sometimes used to temporarily replace other machine code instructions, thus disabling the replaced instruction(s).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
NOP	INH	A7	O	O

ORAA Inclusive OR A ORAA

Operation: (A) + (M) ⇒ A

Description: Performs bitwise logical inclusive OR between the content of accumulator A and the content of memory location M and places the result in A. Each bit of A after the operation is the logical inclusive OR of the corresponding bits of M and of A before the operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ORAA *#opr8i*	IMM	`8A ii`	`P`	`P`
ORAA *opr8a*	DIR	`9A dd`	`rPf`	`rfP`
ORAA *opr16a*	EXT	`BA hh ll`	`rPO`	`rOP`
ORAA *oprx0_xysp*	IDX	`AA xb`	`rPf`	`rfP`
ORAA *oprx9,xysp*	IDX1	`AA xb ff`	`rPO`	`rPO`
ORAA *oprx16,xysp*	IDX2	`AA xb ee ff`	`frPP`	`frPP`
ORAA [D,*xysp*]	[D,IDX]	`AA xb`	`fIfrPf`	`fIfrfP`
ORAA [*oprx16,xysp*]	[IDX2]	`AA xb ee ff`	`fIPrPf`	`fIPrfP`

ORAB Inclusive OR B ORAB

Operation: (B) + (M) ⇒ B

Description: Performs bitwise logical inclusive OR between the content of accumulator B and the content of memory location M. The result is placed in B. Each bit of B after the operation is the logical inclusive OR of the corresponding bits of M and of B before the operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
ORAB *#opr8i*	IMM	`CA ii`	`P`	`P`
ORAB *opr8a*	DIR	`DA dd`	`rPf`	`rfP`
ORAB *opr16a*	EXT	`FA hh ll`	`rPO`	`rOP`
ORAB *oprx0_xysp*	IDX	`EA xb`	`rPf`	`rfP`
ORAB *oprx9,xysp*	IDX1	`EA xb ff`	`rPO`	`rPO`
ORAB *oprx16,xysp*	IDX2	`EA xb ee ff`	`frPP`	`frPP`
ORAB [D,*xysp*]	[D,IDX]	`EA xb`	`fIfrPf`	`fIfrfP`
ORAB [*oprx16,xysp*]	[IDX2]	`EA xb ee ff`	`fIPrPf`	`fIPrfP`

ORCC Logical OR CCR with Mask ORCC

Operation: (CCR) + (M) ⇒ CCR

Description: Performs bitwise logical inclusive OR between the content of memory location M and the content of the CCR and places the result in the CCR. Each bit of the CCR after the operation is the logical OR of the corresponding bits of M and of CCR before the operation. To set one or more bits, set the corresponding bit of the mask equal to 1. Bits corresponding to 0s in the mask are not changed by the ORCC operation.

CCR Details:

S	X	H	I	N	Z	V	C
⇑	–	⇑	⇑	⇑	⇑	⇑	⇑

Condition code bits are set if the corresponding bit was 1 before the operation or if the corresponding bit in the instruction-provided mask is 1. The X interrupt mask cannot be set by any software instruction.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ORCC #*opr8i*	IMM	`14 ii`	`P`	`P`

PSHA Push A onto Stack PSHA

Operation: $(SP) - \$0001 \Rightarrow SP$
$(A) \Rightarrow M_{(SP)}$

Description: Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHA	INH	36	Os	Os

PSHB Push B onto Stack PSHB

Operation: (SP) – \$0001 ⇒ SP
(B) ⇒ $M_{(SP)}$

Description: Stacks the content of accumulator B. The stack pointer is decremented by one. The content of B is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHB	INH	37	Os	Os

PSHC Push CCR onto Stack PSHC

Operation: (SP) – \$0001 ⇒ SP
(CCR) ⇒ $M_{(SP)}$

Description: Stacks the content of the condition codes register. The stack pointer is decremented by one. The content of the CCR is then stored at the address to which the SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHC	INH	39	Os	Os

PSHD Push Double Accumulator onto Stack PSHD

Operation: $(SP) - \$0002 \Rightarrow SP$
$(A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description: Stacks the content of double accumulator D. The stack pointer is decremented by two, then the contents of accumulators A and B are stored at the location to which the SP points.

After PSHD executes, the SP points to the stacked value of accumulator A. This stacking order is the opposite of the order in which A and B are stacked when an interrupt is recognized. The interrupt stacking order is backward-compatible with the M6800, which had no 16-bit accumulator.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHD	INH	3B	OS	OS

PSHX Push Index Register X onto Stack PSHX

Operation: (SP) – \$0002 ⇒ SP
$(X_H : X_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description: Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stored at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHX	INH	34	OS	OS

PSHY Push Index Register Y onto Stack PSHY

Operation: $(SP) – \$0002 \Rightarrow SP$
$(Y_H : Y_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description: Stacks the content of index register Y. The stack pointer is decremented by two. The content of Y is then stored at the address to which the SP points. After PSHY executes, the SP points to the stacked value of the high-order half of Y.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PSHY	INH	35	OS	OS

PULA Pull A from Stack PULA

Operation: $(M_{(SP)}) \Rightarrow A$
$(SP) + \$0001 \Rightarrow SP$

Description: Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULA	INH	`32`	`ufO`	`ufO`

PULB Pull B from Stack PULB

Operation: $(M_{(SP)}) \Rightarrow B$
$(SP) + \$0001 \Rightarrow SP$

Description: Accumulator B is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULB	INH	33	ufO	ufO

PULC Pull Condition Code Register from Stack PULC

Operation: $(M_{(SP)}) \Rightarrow \text{CCR}$
$(\text{SP}) + \$0001 \Rightarrow \text{SP}$

Description: The condition code register is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULC	INH	38	ufO	ufO

PULD Pull Double Accumulator from Stack PULD

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B$
$(SP) + \$0002 \Rightarrow SP$

Description: Double accumulator D is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

The order in which A and B are pulled from the stack is the opposite of the order in which A and B are pulled when an RTI instruction is executed. The interrupt stacking order for A and B is backward-compatible with the M6800, which had no 16-bit accumulator.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULD	INH	3A	UfO	UfO

PULX Pull Index Register X from Stack PULX

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$
$(SP) + \$0002 \Rightarrow SP$

Description: Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULX	INH	30	UfO	UfO

PULY Pull Index Register Y from Stack PULY

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L$
$(SP) + \$0002 \Rightarrow SP$

Description: Index register Y is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
PULY	INH	31	UfO	UfO

REV Fuzzy Logic Rule Evaluation REV

Operation: MIN-MAX Rule Evaluation

Description: Performs an unweighted evaluation of a list of rules, using fuzzy input values to produce fuzzy outputs. REV can be interrupted, so it does not adversely affect interrupt latency.

The REV instruction uses an 8-bit offset from a base address stored in index register Y to determine the address of each fuzzy input and fuzzy output. For REV to execute correctly, each rule in the knowledge base must consist of a table of 8-bit antecedent offsets followed by a table of 8-bit consequent offsets. The value $FE marks boundaries between antecedents and consequents and between successive rules. The value $FF marks the end of the rule list. REV can evaluate any number of rules with any number of inputs and outputs.

Beginning with the address pointed to by the first rule antecedent, REV evaluates each successive fuzzy input value until it encounters an $FE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. Then, beginning with the address pointed to by the first rule consequent, the truth value is compared to each successive fuzzy output value until another $FE separator is encountered; if the truth value is greater than the current output value, it is written to the output. Operation is similar to that of a MAXM instruction. Rules are processed until an $FF terminator is encountered.

Before executing REV, perform these set up operations.

- X must point to the first 8-bit element in the rule list.
- Y must point to the base address for fuzzy inputs and fuzzy outputs.
- A must contain the value $FF, and the CCR V bit must = 0.
  (LDAA #$FF places the correct value in A and clears V.)
- Clear fuzzy outputs to 0s.

Index register X points to the element in the rule list that is being evaluated. X is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, X points to the next address after the $FF separator at the end of the rule list.

REV Fuzzy Logic Rule Evaluation (Continued) REV

Index register Y points to the base address for the fuzzy inputs and fuzzy outputs. The value in Y does not change during execution.

Accumulator A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A, and writes the smaller of the two to A. When all antecedents have been evaluated, A contains the smallest input value. This is the truth value used during consequent processing. Accumulator A must be initialized to $FF for the MIN function to evaluate the inputs of the first rule correctly. For subsequent rules, the value $FF is written to A when an $FE marker is encountered. At the end of execution, accumulator A holds the truth value for the last rule.

The V status bit signals whether antecedents (0) or consequents (1) are being processed. V must be initialized to 0 for processing to begin with the antecedents of the first rule. Once execution begins, the value of V is automatically changed as $FE separators are encountered. At the end of execution, V should equal 1, because the last element before the $FF end marker should be a rule consequent. If V is equal to 0 at the end of execution, the rule list is incorrect.

Fuzzy outputs must be cleared to $00 before processing begins in order for the MAX algorithm used during consequent processing to work correctly. Residual output values would cause incorrect comparison.

Refer to **Section 9. Fuzzy Logic Support** for details.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	?

V: 1; Normally set, unless rule structure is erroneous

H, N, Z, and C may be altered by this instruction

Source Form	Address Mode	Object Code	Access Detail[1]	
			HCS12	M68HC12

REV
(replace comma if interrupted)
	Special	`18 3A`	`Orf(t,tx)O`
			`ff + Orf(t,`
				`Orf(t,tx)O`
				`ff + Orf(t,`

1. The 3-cycle loop in parentheses is executed once for each element in the rule list. When an interrupt occurs, there is a 2-cycle exit sequence, a 4-cycle re-entry sequence, then execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

REVW Fuzzy Logic Rule Evaluation (Weighted) REVW

Operation: MIN-MAX Rule Evaluation with Optional Rule Weighting

Description: REVW performs either weighted or unweighted evaluation of a list of rules, using fuzzy inputs to produce fuzzy outputs. REVW can be interrupted, so it does not adversely affect interrupt latency.

For REVW to execute correctly, each rule in the knowledge base must consist of a table of 16-bit antecedent pointers followed by a table of 16-bit consequent pointers. The value $FFFE marks boundaries between antecedents and consequents, and between successive rules. The value $FFFF marks the end of the rule list. REVW can evaluate any number of rules with any number of inputs and outputs.

Setting the C status bit enables weighted evaluation. To use weighted evaluation, a table of 8-bit weighting factors, one per rule, must be stored in memory. Index register Y points to the weighting factors.

Beginning with the address pointed to by the first rule antecedent, REVW evaluates each successive fuzzy input value until it encounters an $FFFE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. Next, if weighted evaluation is enabled, a computation is performed, and the truth value is modified. Then, beginning with the address pointed to by the first rule consequent, the truth value is compared to each successive fuzzy output value until another $FFFE separator is encountered; if the truth value is greater than the current output value, it is written to the output. Operation is similar to that of a MAXM instruction. Rules are processed until an $FFFF terminator is encountered.

Perform these set up operations before execution:

- X must point to the first 16-bit element in the rule list.
- A must contain the value $FF, and the CCR V bit must = 0 (LDAA #$FF places the correct value in A and clears V).
- Clear fuzzy outputs to 0s.
- Set or clear the CCR C bit. When weighted evaluation is enabled, Y must point to the first item in a table of 8-bit weighting factors.

REVW Fuzzy Logic Rule Evaluation (Weighted) (Continued) REVW

Index register X points to the element in the rule list that is being evaluated. X is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, X points to the address after the $FFFF separator at the end of the rule list.

Index register Y points to the weighting factor being used. Y is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, Y points to the last weighting factor used. When weighting is not used (C = 0), Y is not changed.

Accumulator A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A and writes the smaller of the two to A. When all antecedents have been evaluated, A contains the smallest input value. For unweighted evaluation, this is the truth value used during consequent processing. For weighted evaluation, the value in A is multiplied by the quantity (Rule Weight + 1) and the upper eight bits of the result replace the content of A. Accumulator A must be initialized to $FF for the MIN function to evaluate the inputs of the first rule correctly. For subsequent rules, the value $FF is automatically written to A when an $FFFE marker is encountered. At the end of execution, accumulator A holds the truth value for the last rule.

The V status bit signals whether antecedents (0) or consequents (1) are being processed. V must be initialized to 0 for processing to begin with the antecedents of the first rule. Once execution begins, the value of V is automatically changed as $FFFE separators are encountered. At the end of execution, V should equal 1, because the last element before the $FF end marker should be a rule consequent. If V is equal to 0 at the end of execution, the rule list is incorrect.

Fuzzy outputs must be cleared to $00 before processing begins in order for the MAX algorithm used during consequent processing to work correctly. Residual output values would cause incorrect comparison.

Refer to **Section 9. Fuzzy Logic Support** for details.

REVW Fuzzy Logic Rule Evaluation (Weighted) (Concluded) REVW

CCR Details:

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	!

V: 1; Normally set, unless rule structure is erroneous

C: Selects weighted (1) or unweighted (0) rule evaluation

H, N, Z, and C may be altered by this instruction

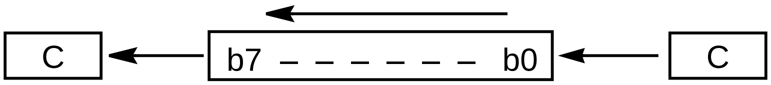
Source Form	Address Mode	Object Code	Access Detail[1]	
			HCS12	M68HC12

REVW
(add 2 at end of ins if wts)
(replace comma if interrupted)
	Special	`18 3B`	`ORf(t,Tx)O`
			`(r,RfRf)`
			`ffff + ORf(t,`
				`ORf(tTx)O`
				`(r,RfRf)`
				`ffff + ORf(t,`

[1] The 3-cycle loop in parentheses expands to five cycles for separators when weighting is enabled. The loop is executed once for each element in the rule list. When an interrupt occurs, there is a 2-cycle exit sequence, a 4-cycle re-entry sequence, then execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

ROL Rotate Left Memory ROL

Operation:



Description: Shifts all bits of memory location M one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of M. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, ROL HIGH could be used where LOW, MID and HIGH refer to the low-order, middle and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

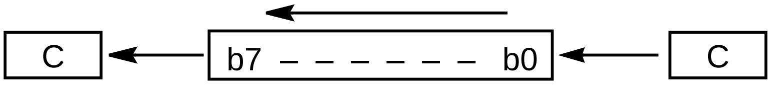
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M7
Set if the MSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ROL *opr16a*	EXT	`75 hh ll`	`rPwO`	`rOPw`
ROL *oprx0_xysp*	IDX	`65 xb`	`rPw`	`rPw`
ROL *oprx9,xysp*	IDX1	`65 xb ff`	`rPwO`	`rPOw`
ROL *oprx16,xysp*	IDX2	`65 xb ee ff`	`frPwP`	`frPPw`
ROL [D,*xysp*]	[D,IDX]	`65 xb`	`fIfrPw`	`fIfrPw`
ROL [*oprx16,xysp*]	[IDX2]	`65 xb ee ff`	`fIPrPw`	`fIPrPw`

ROLA Rotate Left A ROLA

Operation:



Description: Shifts all bits of accumulator A one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of A. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, and ROL HIGH could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

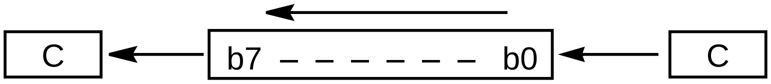
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A7
Set if the MSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ROLA	INH	45	O	O

ROLB Rotate Left B ROLB

Operation:



Description: Shifts all bits of accumulator B one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of B. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, and ROL HIGH could be used where LOW, MID, and HIGH refer to the low-order, middle and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

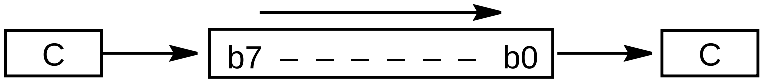
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B7
Set if the MSB of B was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ROLB	INH	55	O	O

ROR Rotate Right Memory ROR

Operation:



Description: Shifts all bits of memory location M one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of M. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

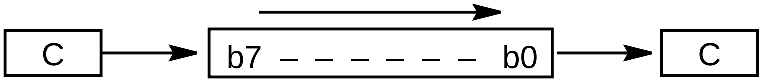
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
ROR *opr16a*	EXT	`76 hh ll`	`rPwO`	`rOPw`
ROR *oprx0_xysp*	IDX	`66 xb`	`rPw`	`rPw`
ROR *oprx9,xysp*	IDX1	`66 xb ff`	`rPwO`	`rPOw`
ROR *oprx16,xysp*	IDX2	`66 xb ee ff`	`frPwP`	`frPPw`
ROR [D,*xysp*]	[D,IDX]	`66 xb`	`fIfrPw`	`fIfrPw`
ROR [*oprx16,xysp*]	[IDX2]	`66 xb ee ff`	`fIPrPw`	`fIPrPw`

RORA Rotate Right A RORA

Operation:



Description: Shifts all bits of accumulator A one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of A. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

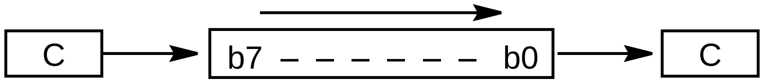
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A0
Set if the LSB of A was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RORA	INH	46	O	O

RORB Rotate Right B RORB

Operation:



Description: Shifts all bits of accumulator B one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of B. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle and high-order bytes of the 24-bit value, respectively.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B0
Set if the LSB of B was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RORB	INH	56	O	O

RTC Return from Call RTC

Operation:

$(M_{(SP)}) \Rightarrow$ PPAGE
$(SP) + \$0001 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$
$(SP) + \$0002 \Rightarrow SP$

Description: Terminates subroutines in expanded memory invoked by the CALL instruction. Returns execution flow from the subroutine to the calling program. The program overlay page (PPAGE) register and the return address are restored from the stack; program execution continues at the restored address. For code compatibility purposes, CALL and RTC also execute correctly in devices that do not have expanded memory capability.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RTC	INH	`0A`	`uUnfPPP`	`uUnPPP`

RTI Return from Interrupt RTI

Operation:

$(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) – \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

Description: Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

If another interrupt is pending when RTI has finished restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched. This operation is functionally identical to the same operation in the M68HC11, where registers actually are re-stacked, but is faster.

CCR Details:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RTI (with interrupt pending)	INH	0B	uUUUUPPP uUUUUfVfPPP	uUUUUPPP uUUUUVfPPP

RTS Return from Subroutine RTS

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$; $(SP) + \$0002 \Rightarrow SP$

Description: Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
RTS	INH	`3D`	`UfPPP`	`UfPPP`

SBA Subtract Accumulators SBA

Operation: $(A) - (B) \Rightarrow A$

Description: Subtracts the content of accumulator B from the content of accumulator A and places the result in A. The content of B is not affected. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{B7} \bullet \overline{R7} + \overline{A7} \bullet B7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet B7 + B7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the absolute value of B is larger than the absolute value of A; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SBA	INH	`18 16`	OO	OO

SBCA Subtract with Carry from A SBCA

Operation: (A) – (M) – C ⇒ A

Description: Subtracts the content of memory location M and the value of the C status bit from the content of accumulator A. The result is placed in A. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SBCA #*opr8i*	IMM	82 ii	P	P
SBCA *opr8a*	DIR	92 dd	rPf	rfP
SBCA *opr16a*	EXT	B2 hh ll	rPO	rOP
SBCA *oprx0_xysp*	IDX	A2 xb	rPf	rfP
SBCA *oprx9,xysp*	IDX1	A2 xb ff	rPO	rPO
SBCA *oprx16,xysp*	IDX2	A2 xb ee ff	frPP	frPP
SBCA [D,*xysp*]	[D,IDX]	A2 xb	fIfrPf	fIfrfP
SBCA [*oprx16,xysp*]	[IDX2]	A2 xb ee ff	fIPrPf	fIPrfP

SBCB Subtract with Carry from B SBCB

Operation: (B) – (M) – C ⇒ B

Description: Subtracts the content of memory location M and the value of the C status bit from the content of accumulator B. The result is placed in B. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$
Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SBCB *#opr8i*	IMM	`C2 ii`	`P`	`P`
SBCB *opr8a*	DIR	`D2 dd`	`rPf`	`rfP`
SBCB *opr16a*	EXT	`F2 hh ll`	`rPO`	`rOP`
SBCB *oprx0_xysp*	IDX	`E2 xb`	`rPf`	`rfP`
SBCB *oprx9*,*xysp*	IDX1	`E2 xb ff`	`rPO`	`rPO`
SBCB *oprx16*,*xysp*	IDX2	`E2 xb ee ff`	`frPP`	`frPP`
SBCB [D,*xysp*]	[D,IDX]	`E2 xb`	`fIfrPf`	`fIfrfP`
SBCB [*oprx16*,*xysp*]	[IDX2]	`E2 xb ee ff`	`fIPrPf`	`fIPrfP`

SEC Set Carry SEC

Operation: 1 ⇒ C bit

Description: Sets the C status bit. This instruction is assembled as ORCC #$01. The ORCC instruction can be used to set any combination of bits in the CCR in one operation.

SEC can be used to set up the C bit prior to a shift or rotate instruction involving the C bit.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	1

C: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SEC *translates to...* ORCC #$01	IMM	`14 01`	`P`	`P`

SEI Set Interrupt Mask SEI

Operation: 1 ⇒ I bit

Description: Sets the I mask bit. This instruction is assembled as ORCC #$10. The ORCC instruction can be used to set any combination of bits in the CCR in one operation. When the I bit is set, all maskable interrupts are inhibited, and the CPU will recognize only non-maskable interrupt sources or an SWI.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SEI *translates to...* ORCC #$10	IMM	14 10	P	P

SEV Set Two’s Complement Overflow Bit SEV

Operation: 1 ⇒ V bit

Description: Sets the V status bit. This instruction is assembled as ORCC #$02. The ORCC instruction can be used to set any combination of bits in the CCR in one operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	1	–

V: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SEV *translates to...* ORCC #$02	IMM	`14 02`	`P`	`P`

SEX Sign Extend into 16-Bit Register SEX

Operation: If r1 bit 7 = 0, then \$00 : (r1) ⇒ r2
If r1 bit 7 = 1, then \$FF : (r1) ⇒ r2

Description: This instruction is an alternate mnemonic for the TFR r1,r2 instruction, where r1 is an 8-bit register and r2 is a 16-bit register. The result in r2 is the 16-bit sign extended representation of the original two's complement number in r1. The content of r1 is unchanged in all cases except that of SEX A,D (D is A : B).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail HCS12	Access Detail M68HC12
SEX *abc,dxys*	INH	`B7 eb`	`P`	`P`

[1] Legal coding for `eb` is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit (MSB is a don't care). Values are in hexadecimal.

	0	1	2
3	sex:A ⇒ TMP2	sex:B ⇒ TMP2	sex:CCR ⇒ TMP2
4	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR ⇒ D SEX CCR,D
5	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR ⇒ X SEX CCR,X
6	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR ⇒ Y SEX CCR,Y
7	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR ⇒ SP SEX CCR,SP

STAA Store Accumulator A STAA

Operation: (A) ⇒ M

Description: Stores the content of accumulator A in memory location M. The content of A is unchanged.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
STAA *opr8a*	DIR	5A dd	Pw	Pw
STAA *opr16a*	EXT	7A hh ll	PwO	wOP
STAA *oprx0_xysp*	IDX	6A xb	Pw	Pw
STAA *oprx9,xysp*	IDX1	6A xb ff	PwO	PwO
STAA *oprx16,xysp*	IDX2	6A xb ee ff	PwP	PwP
STAA [D,*xysp*]	[D,IDX]	6A xb	PIfw	PIfPw
STAA [*oprx16,xysp*]	[IDX2]	6A xb ee ff	PIPw	PIPPw

STAB Store Accumulator B STAB

Operation: (B) ⇒ M

Description: Stores the content of accumulator B in memory location M. The content of B is unchanged.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
STAB *opr8a*	DIR	5B dd	Pw	Pw
STAB *opr16a*	EXT	7B hh ll	PwO	wOP
STAB *oprx0_xysp*	IDX	6B xb	Pw	Pw
STAB *oprx9*,*xysp*	IDX1	6B xb ff	PwO	PwO
STAB *oprx16*,*xysp*	IDX2	6B xb ee ff	PwP	PwP
STAB [D,*xysp*]	[D,IDX]	6B xb	PIfw	PIfPw
STAB [*oprx16,xysp*]	[IDX2]	6B xb ee ff	PIPw	PIPPw

STD Store Double Accumulator STD

Operation: (A : B) ⇒ M : M + 1

Description: Stores the content of double accumulator D in memory location M : M + 1. The content of D is unchanged.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
STD *opr8a*	DIR	`5C dd`	`PW`	`PW`
STD *opr16a*	EXT	`7C hh ll`	`PWO`	`WOP`
STD *oprx0_xysp*	IDX	`6C xb`	`PW`	`PW`
STD *oprx9,xysp*	IDX1	`6C xb ff`	`PWO`	`PWO`
STD *oprx16,xysp*	IDX2	`6C xb ee ff`	`PWP`	`PWP`
STD [D,*xysp*]	[D,IDX]	`6C xb`	`PIfW`	`PIfPW`
STD [*oprx16,xysp*]	[IDX2]	`6C xb ee ff`	`PIPW`	`PIPPW`

STOP Stop Processing STOP

Operation:

$(SP) – \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
Stop All Clocks

Description:

When the S control bit is set, STOP is disabled and operates like a 2-cycle NOP instruction. When the S bit is cleared, STOP stacks CPU context, stops all system clocks, and puts the device in standby mode.

Standby operation minimizes system power consumption. The contents of registers and the states of I/O pins remain unchanged.

Asserting the $\overline{\text{RESET}}$, $\overline{\text{XIRQ}}$, or $\overline{\text{IRQ}}$ signals ends standby mode. Stacking on entry to STOP allows the CPU to recover quickly when an interrupt is used, provided a stable clock is applied to the device. If the system uses a clock reference crystal that also stops during low-power mode, crystal startup delay lengthens recovery time.

If $\overline{\text{XIRQ}}$ is asserted while the X mask bit = 0 ($\overline{\text{XIRQ}}$ interrupts enabled), execution resumes with a vector fetch for the $\overline{\text{XIRQ}}$ interrupt. If the X mask bit = 1 ($\overline{\text{XIRQ}}$ interrupts disabled), a 2-cycle recovery sequence including an O cycle is used to adjust the instruction queue and the stack-pointer, and execution continues with the next instruction after STOP.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
STOP (entering STOP)	INH	`18 3E`	`OOSSSSsf`	`OOSSSfSs`
(exiting STOP)			`fVfPPP`	`fVfPPP`
(continue)			`ff`	`fO`
(if STOP disabled)			`OO`	`OO`

STS Store Stack Pointer STS

Operation: $(SP_H : SP_L) \Rightarrow M : M + 1$

Description: Stores the content of the stack pointer in memory. The most significant byte of the SP is stored at the specified address, and the least significant byte of the SP is stored at the next higher byte address (the specified address plus one).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
STS *opr8a*	DIR	`5F dd`	`PW`	`PW`
STS *opr16a*	EXT	`7F hh ll`	`PWO`	`WOP`
STS *oprx0_xysp*	IDX	`6F xb`	`PW`	`PW`
STS *oprx9,xysp*	IDX1	`6F xb ff`	`PWO`	`PWO`
STS *oprx16,xysp*	IDX2	`6F xb ee ff`	`PWP`	`PWP`
STS [D,*xysp*]	[D,IDX]	`6F xb`	`PIfW`	`PIfPW`
STS [*oprx16,xysp*]	[IDX2]	`6F xb ee ff`	`PIPW`	`PIPPW`

STX Store Index Register X STX

Operation: $(X_H : X_L) \Rightarrow M : M + 1$

Description: Stores the content of index register X in memory. The most significant byte of X is stored at the specified address, and the least significant byte of X is stored at the next higher byte address (the specified address plus one).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
STX *opr8a*	DIR	`5E dd`	`PW`	`PW`
STX *opr16a*	EXT	`7E hh ll`	`PWO`	`WOP`
STX *oprx0_xysp*	IDX	`6E xb`	`PW`	`PW`
STX *oprx9,xysp*	IDX1	`6E xb ff`	`PWO`	`PWO`
STX *oprx16,xysp*	IDX2	`6E xb ee ff`	`PWP`	`PWP`
STX [D,*xysp*]	[D,IDX]	`6E xb`	`PIfW`	`PIfPW`
STX [*oprx16,xysp*]	[IDX2]	`6E xb ee ff`	`PIPW`	`PIPPW`

STY Store Index Register Y STY

Operation: $(Y_H : Y_L) \Rightarrow M : M + 1$

Description: Stores the content of index register Y in memory. The most significant byte of Y is stored at the specified address, and the least significant byte of Y is stored at the next higher byte address (the specified address plus one).

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
STY *opr8a*	DIR	`5D dd`	`PW`	`PW`
STY *opr16a*	EXT	`7D hh ll`	`PWO`	`WOP`
STY *oprx0_xysp*	IDX	`6D xb`	`PW`	`PW`
STY *oprx9,xysp*	IDX1	`6D xb ff`	`PWO`	`PWO`
STY *oprx16,xysp*	IDX2	`6D xb ee ff`	`PWP`	`PWP`
STY [D,*xysp*]	[D,IDX]	`6D xb`	`PIfW`	`PIfPW`
STY [*oprx16,xysp*]	[IDX2]	`6D xb ee ff`	`PIPW`	`PIPPW`

SUBA Subtract A SUBA

Operation: (A) – (M) ⇒ A

Description: Subtracts the content of memory location M from the content of accumulator A, and places the result in A. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SUBA #*opr8i*	IMM	`80 ii`	`P`	`P`
SUBA *opr8a*	DIR	`90 dd`	`rPf`	`rfP`
SUBA *opr16a*	EXT	`B0 hh ll`	`rPO`	`rOP`
SUBA *oprx0_xysp*	IDX	`A0 xb`	`rPf`	`rfP`
SUBA *oprx9,xysp*	IDX1	`A0 xb ff`	`rPO`	`rPO`
SUBA *oprx16,xysp*	IDX2	`A0 xb ee ff`	`frPP`	`frPP`
SUBA [D,*xysp*]	[D,IDX]	`A0 xb`	`fIfrPf`	`fIfrfP`
SUBA [*oprx16,xysp*]	[IDX2]	`A0 xb ee ff`	`fIPrPf`	`fIPrfP`

SUBB Subtract B SUBB

Operation: (B) – (M) ⇒ B

Description: Subtracts the content of memory location M from the content of accumulator B and places the result in B. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$
Set if a two’s complement overflow resulted from the operation; cleared otherwise

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SUBB *#opr8i*	IMM	C0 ii	P	P
SUBB *opr8a*	DIR	D0 dd	rPf	rfP
SUBB *opr16a*	EXT	F0 hh ll	rPO	rOP
SUBB *oprx0_xysp*	IDX	E0 xb	rPf	rfP
SUBB *oprx9,xysp*	IDX1	E0 xb ff	rPO	rPO
SUBB *oprx16,xysp*	IDX2	E0 xb ee ff	frPP	frPP
SUBB [D,*xysp*]	[D,IDX]	E0 xb	fIfrPf	fIfrfP
SUBB [*oprx16,xysp*]	[IDX2]	E0 xb ee ff	fIPrPf	fIPrfP

SUBD Subtract Double Accumulator SUBD

Operation: (A : B) – (M : M + 1) ⇒ A : B

Description: Subtracts the content of memory location M : M + 1 from the content of double accumulator D and places the result in D. For subtraction instructions, the C status bit represents a borrow.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SUBD #*opr16i*	IMM	`83 jj kk`	`PO`	`OP`
SUBD *opr8a*	DIR	`93 dd`	`RPf`	`RfP`
SUBD *opr16a*	EXT	`B3 hh ll`	`RPO`	`ROP`
SUBD *oprx0_xysp*	IDX	`A3 xb`	`RPf`	`RfP`
SUBD *oprx9,xyssp*	IDX1	`A3 xb ff`	`RPO`	`RPO`
SUBD *oprx16,xysp*	IDX2	`A3 xb ee ff`	`fRPP`	`fRPP`
SUBD [D,*xysp*]	[D,IDX]	`A3 xb`	`fIfRPf`	`fIfRfP`
SUBD [*oprx16,xysp*]	[IDX2]	`A3 xb ee ff`	`fIPRPf`	`fIPRfP`

SWI Software Interrupt SWI

Operation:

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
$1 \Rightarrow I$
$(\text{SWI Vector}) \Rightarrow PC$

Description: Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to **Section 7. Exception Processing** for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
SWI	INH	3F	VSPSSPSsP[1]	VSPSSPSsP[1]

[1] The CPU also uses the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation of the sequence (VfPPP) is used for resets.

TAB Transfer from Accumulator A to Accumulator B TAB

Operation: (A) ⇒ B

Description: Moves the content of accumulator A to accumulator B. The former content of B is lost; the content of A is not affected. Unlike the general transfer instruction TFR A,B which does not affect condition codes, the TAB instruction affects the N, Z, and V status bits for compatibility with M68HC11.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
TAB	INH	18 0E	OO	OO

TAP Transfer from Accumulator A to Condition Code Register TAP

Operation: (A) ⇒ CCR

Description: Transfers the logic states of bits [7:0] of accumulator A to the corresponding bit positions of the CCR. The content of A remains unchanged. The X mask bit can be cleared as a result of a TAP, but cannot be set if it was cleared prior to execution of the TAP. If the I bit is cleared, there is a 1-cycle delay before the system allows interrupt requests. This prevents interrupts from occurring between instructions in the sequences CLI, WAI and CLI, STOP.

This instruction is accomplished with the TFR A,CCR instruction. For compatibility with the M68HC11, the mnemonic TAP is translated by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

Condition codes take on the value of the corresponding bit of accumulator A, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TAP *translates to...* TFR A,CCR	INH	`B7 02`	`P`	`P`

TBA Transfer from Accumulator B to Accumulator A TBA

Operation: (B) ⇒ A

Description: Moves the content of accumulator B to accumulator A. The former content of A is lost; the content of B is not affected. Unlike the general transfer instruction TFR B,A, which does not affect condition codes, the TBA instruction affects N, Z, and V for compatibility with M68HC11.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TBA	INH	`18 0F`	OO	OO

TBEQ Test and Branch if Equal to Zero TBEQ

Operation: If (Counter) = 0, then (PC) + $0003 + Rel ⇒ PC

Description: Tests the specified counter register A, B, D, X, Y, or SP. If the counter register is zero, branches to the specified relative destination. TBEQ is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBEQ and IBEQ instructions are similar to TBEQ, except that the counter is decremented or incremented rather than simply being tested. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
TBEQ *abdxys,rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

[1] Encoding for `lb` is summarized in the following table. Bit 3 is not used (don’t care), bit 5 selects branch on zero (TBEQ – 0) or not zero (TBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 0:1 for TBEQ.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	TBEQ A, *rel9*	`04 40 rr`	`04 50 rr`
B	001	TBEQ B, *rel9*	`04 41 rr`	`04 51 rr`
D	100	TBEQ D, *rel9*	`04 44 rr`	`04 54 rr`
X	101	TBEQ X, *rel9*	`04 45 rr`	`04 55 rr`
Y	110	TBEQ Y, *rel9*	`04 46 rr`	`04 56 rr`
SP	111	TBEQ SP, *rel9*	`04 47 rr`	`04 57 rr`

TBL Table Lookup and Interpolate TBL

Operation: $(M) + [(B) \times ((M+1) - (M))] \Rightarrow A$

Description: Linearly interpolates one of 256 result values that fall between each pair of data entries in a lookup table stored in memory. Data entries in the table represent the Y values of endpoints of equally spaced line segments. Table entries and the interpolated result are 8-bit values. The result is stored in accumulator A.

Before executing TBL, an index register points to the table entry corresponding to the X value (X1) that is closest to, but less than or equal to, the desired lookup point (XL, YL). This defines the left end of a line segment and the right end is defined by the next data entry in the table. Prior to execution, accumulator B holds a binary fraction (radix point to left of MSB), representing the ratio (XL–X1) ÷ (X2–X1).

The 8-bit unrounded result is calculated using the following expression:

$$A = Y1 + [(B) \times (Y2 - Y1)]$$

Where

(B) = (XL – X1) ÷ (X2 – X1)
Y1 = 8-bit data entry pointed to by <effective address>
Y2 = 8-bit data entry pointed to by <effective address> + 1

The intermediate value [(B) × (Y2 – Y1)] produces a 16-bit result with the radix point between bits 7 and 8. Any indexed addressing mode referenced to X, Y, SP, or PC, except indirect modes or 9-bit and 16-bit offset modes, can be used to identify the first data point (X1,Y1). The second data point is the next table entry.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ[1]

[1] C-bit was undefined in original M68HC12.

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
C: Set if result can be rounded up; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TBL *oprx0_xysp*	IDX	`18 3D xb`	`ORfffP`	`OrrffffP`

TBNE Test and Branch if Not Equal to Zero TBNE

Operation: If (Counter) ≠ 0, then (PC) + \$0003 + Rel ⇒ PC

Description: Tests the specified counter register A, B, D, X, Y, or SP. If the counter register is not zero, branches to the specified relative destination. TBNE is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBNE and IBNE instructions are similar to TBNE, except that the counter is decremented or incremented rather than simply being tested. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code(1)	Access Detail HCS12	Access Detail M68HC12
TBNE *abdxys,rel9*	REL	`04 lb rr`	`PPP/PPO`	`PPP`

1. Encoding for `lb` is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (TBEQ – 0) or not zero (TBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 0:1 for TBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	TBNE A, *rel9*	`04 60 rr`	`04 70 rr`
B	001	TBNE B, *rel9*	`04 61 rr`	`04 71 rr`
D	100	TBNE D, *rel9*	`04 64 rr`	`04 74 rr`
X	101	TBNE X, *rel9*	`04 65 rr`	`04 75 rr`
Y	110	TBNE Y, *rel9*	`04 66 rr`	`04 76 rr`
SP	111	TBNE SP, *rel9*	`04 67 rr`	`04 77 rr`

TFR — Transfer Register Content to Another Register — TFR

Operation: See table.

Description: Transfers the content of a source register to a destination register specified in the instruction. The order in which transfers between 8-bit and 16-bit registers are specified affects the high byte of the 16-bit registers differently. Cases involving TMP2 and TMP3 are reserved for Freescale use, so some assemblers may not permit their use. It is possible to generate these cases by using DC.B or DC.W assembler directives.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Or:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

None affected, unless the CCR is the destination register. Condition codes take on the value of the corresponding source bits, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Source Form	Address Mode	Object Code[1]	Access Detail	
			HCS12	M68HC12
TFR *abcdxys,abcdxys*	INH	`B7 eb`	`P`	`P`

[1] Legal coding for `eb` is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit (MSB is a don't-care). Values are in hexadecimal.

	0	1	2	3	4	5	6	7
0	A ⇒ A	B ⇒ A	CCR ⇒ A	TMP3_L ⇒ A	B ⇒ A	X_L ⇒ A	Y_L ⇒ A	SP_L ⇒ A
1	A ⇒ B	B ⇒ B	CCR ⇒ B	TMP3_L ⇒ B	B ⇒ B	X_L ⇒ B	Y_L ⇒ B	SP_L ⇒ B
2	A ⇒ CCR	B ⇒ CCR	CCR ⇒ CCR	TMP3_L ⇒ CCR	B ⇒ CCR	X_L ⇒ CCR	Y_L ⇒ CCR	SP_L ⇒ CCR
3	sex:A ⇒ TMP2	sex:B ⇒ TMP2	sex:CCR ⇒ TMP2	TMP3 ⇒ TMP2	D ⇒ TMP2	X ⇒ TMP2	Y ⇒ TMP2	SP ⇒ TMP2
4	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR ⇒ D SEX CCR,D	TMP3 ⇒ D	D ⇒ D	X ⇒ D	Y ⇒ D	SP ⇒ D
5	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR ⇒ X SEX CCR,X	TMP3 ⇒ X	D ⇒ X	X ⇒ X	Y ⇒ X	SP ⇒ X
6	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR ⇒ Y SEX CCR,Y	TMP3 ⇒ Y	D ⇒ Y	X ⇒ Y	Y ⇒ Y	SP ⇒ Y
7	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR ⇒ SP SEX CCR,SP	TMP3 ⇒ SP	D ⇒ SP	X ⇒ SP	Y ⇒ SP	SP ⇒ SP

TPA Transfer from Condition Code Register to Accumulator A TPA

Operation: (CCR) ⇒ A

Description: Transfers the content of the condition code register to corresponding bit positions of accumulator A. The CCR remains unchanged.

This mnemonic is implemented by the TFR CCR,A instruction. For compatibility with the M68HC11, the mnemonic TPA is translated into the TFR CCR,A instruction by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TPA *translates to...* TFR CCR,A	INH	`B7 20`	`P`	`P`

TRAP Unimplemented Opcode Trap TRAP

Operation:

$(SP) – \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
$1 \Rightarrow I$
$(Trap\ Vector) \Rightarrow PC$

Description: Traps unimplemented opcodes. There are opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the unimplemented opcodes on page 2, an opcode trap interrupt occurs. Unimplemented opcode traps are essentially interrupts that share the $FFF8:$FFF9 interrupt vector.

TRAP uses the next address after the unimplemented opcode as a return address. It stacks the return address, index registers Y and X, accumulators B and A, and the CCR, automatically decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the trap vector, and instruction execution resumes at that location. This instruction is not maskable by the I bit. Refer to **Section 7. Exception Processing** for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: 1; set

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TRAP *trapnum*	INH	`$18 tn`[1]	`OVSPSSPSsP`	`OfVSPSSPSsP`

[1] The value tn represents an unimplemented page 2 opcode in either of the two ranges $30 to $39 or $40 to $FF.

TST Test Memory TST

Operation: (M) – $00

Description: Subtracts $00 from the content of memory location M and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying M.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TST *opr16a*	EXT	`F7 hh ll`	`rPO`	`rOP`
TST *oprx0_xysp*	IDX	`E7 xb`	`rPf`	`rfP`
TST *oprx9,xysp*	IDX1	`E7 xb ff`	`rPO`	`rPO`
TST *oprx16,xysp*	IDX2	`E7 xb ee ff`	`frPP`	`frPP`
TST [D,*xysp*]	[D,IDX]	`E7 xb`	`fIfrPf`	`fIfrfP`
TST [*oprx16,xysp*]	[IDX2]	`E7 xb ee ff`	`fIPrPf`	`fIPrfP`

TSTA Test A TSTA

Operation: (A) – $00

Description: Subtracts $00 from the content of accumulator A and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying A.

The TSTA instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TSTA. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $00; cleared otherwise
V: 0; cleared
C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TSTA	INH	97	O	O

TSTB Test B TSTB

Operation: (B) – $00

Description: Subtracts $00 from the content of accumulator B and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying B.

The TSTB instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TSTB. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is $00; cleared otherwise

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TSTB	INH	D7	O	O

TSX Transfer from Stack Pointer to Index Register X TSX

Operation: (SP) ⇒ X

Description: This is an alternate mnemonic to transfer the stack pointer value to index register X. The content of the SP remains unchanged. After a TSX instruction, X points at the last value that was stored on the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TSX *translates to...* TFR SP,X	INH	`B7 75`	`P`	`P`

TSY Transfer from Stack Pointer to Index Register Y TSY

Operation: (SP) ⇒ Y

Description: This is an alternate mnemonic to transfer the stack pointer value to index register Y. The content of the SP remains unchanged. After a TSY instruction, Y points at the last value that was stored on the stack.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TSY *translates to...* TFR SP,Y	INH	`B7 76`	`P`	`P`

TXS Transfer from Index Register X to Stack Pointer TXS

Operation: (X) ⇒ SP

Description: This is an alternate mnemonic to transfer index register X value to the stack pointer. The content of X is unchanged.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
TXS *translates to...* TFR X,SP	INH	B7 57	P	P

TYS — Transfer from Index Register Y to Stack Pointer — TYS

Operation: (Y) ⇒ SP

Description: This is an alternate mnemonic to transfer index register Y value to the stack pointer. The content of Y is unchanged.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
TYS *translates to...* TFR Y,SP	INH	`B7 67`	`P`	`P`

WAI Wait for Interrupt WAI

Operation:

$(SP) – \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
$(SP) – \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
Stop CPU Clocks

Description: Puts the CPU into a wait state. Uses the address of the instruction following WAI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked.

The CPU then enters a wait state for an integer number of bus clock cycles. During the wait state, CPU clocks are stopped, but other MCU clocks can continue to run. The CPU leaves the wait state when it senses an interrupt that has not been masked.

Upon leaving the wait state, the CPU sets the appropriate interrupt mask bit(s), fetches the vector corresponding to the interrupt sensed, and instruction execution continues at the location the vector points to.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
WAI (before interrupt)	INH	3E	OSSSSsf	OSSSfSsf
WAI (when interrupt comes)			fVfPPP	VfPPP

Although the WAI instruction itself does not alter the condition codes, the interrupt that causes the CPU to resume processing also causes the I mask bit (and the X mask bit, if the interrupt was $\overline{\text{XIRQ}}$) to be set as the interrupt vector is fetched.

WAV Weighted Average WAV

Operation: Do until B = 0, leave SOP in Y : D, SOW in X

Partial Product = (M pointed to by X) × (M pointed to by Y)
Sum-of-Products (24-bit SOP) = Previous SOP + Partial Product
Sum-of-Weights (16-bit SOW) = Previous SOW + (M pointed to by Y)
(X) + \$0001 ⇒ X; (Y) + \$0001 ⇒ Y
(B) – \$01 ⇒ B

Description: Performs weighted average calculations on values stored in memory. Uses indexed (X) addressing mode to reference one source operand list, and indexed (Y) addressing mode to reference a second source operand list. Accumulator B is used as a counter to control the number of elements to be included in the weighted average.

For each pair of data points, a 24-bit sum of products (SOP) and a 16-bit sum of weights (SOW) is accumulated in temporary registers. When B reaches zero (no more data pairs), the SOP is placed in Y : D. The SOW is placed in X. To arrive at the final weighted average, divide the content of Y : D by X by executing an EDIV after the WAV.

This instruction can be interrupted. If an interrupt occurs during WAV execution, the intermediate results (six bytes) are stacked in the order $\text{SOW}_{[15:0]}$, $\text{SOP}_{[15:0]}$, $\$00\text{:SOP}_{[23:16]}$ before the interrupt is processed. The wavr pseudo-instruction is used to resume execution after an interrupt. The mechanism is re-entrant. New WAV instructions can be started and interrupted while a previous WAV instruction is interrupted.

This instruction is often used in fuzzy logic rule evaluation. Refer to **Section 9. Fuzzy Logic Support** for more information.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	?	–	?	1	?	?

Z: 1; set
H, N, V and C may be altered by this instruction

Source Form	Address Mode	Object Code	Access Detail[1] HCS12	M68HC12
WAV	Special	18 3C	`Of(frr,ffff)O`	`Off(frr,fffff)O`
			(replace comma if interrupted)	
			`SSS + UUUrr`	`SSSf + UUUrr`

[1] The replace comma sequence in parentheses represents the loop for one iteration of SOP and SOW accumulation.

XGDX — Exchange Double Accumulator and Index Register X — XGDX

Operation: (D) ⇔ (X)

Description: Exchanges the content of double accumulator D and the content of index register X. For compatibility with the M68HC11, the XGDX instruction is translated into an EXG D,X instruction by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12	M68HC12
XGDX *translates to...* EXG D,X	INH	`B7 C5`	`P`	`P`

XGDY Exchange Double Accumulator and Index Register Y XGDY

Operation: (D) ⇔ (Y)

Description: Exchanges the content of double accumulator D and the content of index register Y. For compatibility with the M68HC11, the XGDY instruction is translated into an EXG D,Y instruction by the assembler.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12	Access Detail M68HC12
XGDY *translates to...* EXG D,Y	INH	`B7 C6`	`P`	`P`

Section 7. Exception Processing

7.1 Introduction

Exceptions are events that require processing outside the normal flow of instruction execution. This section describes exceptions and the way each is handled.

7.2 Types of Exceptions

Central processor unit (CPU12) exceptions include:

- Resets
  - Power-on reset and $\overline{\text{RESET}}$ pin
  - Clock monitor reset
  - COP watchdog reset
- An unimplemented opcode trap
- A software interrupt instruction (SWI)
- Non-maskable (X-bit) interrupts
- Maskable (I-bit) interrupts

Each exception has an associated 16-bit vector, which points to the memory location where the routine that handles the exception is located. As shown in **Table 7-1**, vectors are stored in the upper bytes of the standard 64-Kbyte address map.

The six highest vector addresses are used for resets and unmaskable interrupt sources. The remaining vectors are used for maskable interrupts. All vectors must be programmed to point to the address of the appropriate service routine.

Table 7-1. CPU12 Exception Vector Map[1]

Vector Address	Source
\$FFFE–\$FFFF	System reset
\$FFFC–\$FFFD	Clock monitor reset
\$FFFA–\$FFFB	COP reset
\$FFF8–\$FFF9	Unimplemented opcode trap
\$FFF6–\$FFF7	Software interrupt instruction (SWI)
\$FFF4–\$FFF5	$\overline{\text{XIRQ}}$ signal
\$FFF2–\$FFF3	$\overline{\text{IRQ}}$ signal
\$FF00–\$FFF1	Device-specific interrupt sources (HCS12)
\$FFC0–\$FFF1	Device-specific interrupt sources (M68HC12)

[1] See Device User Guide and Interrupt Block Guide for further details

The CPU12 can handle up to 128 exception vectors, but the number actually used varies from device to device, and some vectors are reserved for Freescale use. Refer to Device User Guide for more information.

Exceptions can be classified by the effect of the X and I interrupt mask bits on recognition of a pending request.

- Resets, the unimplemented opcode trap, and the SWI instruction are not affected by the X and I mask bits.
- Interrupt service requests from the $\overline{\text{XIRQ}}$ pin are inhibited when X = 1, but are not affected by the I bit.
- All other interrupts are inhibited when I = 1.

7.3 Exception Priority

A hardware priority hierarchy determines which reset or interrupt is serviced first when simultaneous requests are made. Six sources are not maskable. The remaining sources are maskable, and the device integration module typically can change the relative priorities of maskable interrupts. Refer to **7.5 Interrupts** for more detail concerning interrupt priority and servicing.

The priorities of the unmaskable sources are:

1. $\overline{\text{RESET}}$ pin or power-on reset (POR)
2. Clock monitor reset
3. Computer operating properly (COP) watchdog reset
4. Non-maskable interrupt request ($\overline{\text{XIRQ}}$) signal
5. Unimplemented opcode trap
6. Software interrupt instruction (SWI)

External reset and POR share the highest exception-processing priority, followed by clock monitor reset, and then the on-chip watchdog reset.

The $\overline{\text{XIRQ}}$ interrupt is pseudo-non-maskable. After reset, the X bit in the CCR is set, which inhibits all interrupt service requests from the $\overline{\text{XIRQ}}$ pin until the X bit is cleared. The X bit can be cleared by a program instruction, but program instructions cannot change X from 0 to 1. Once the X bit is cleared, interrupt service requests made via the $\overline{\text{XIRQ}}$ pin become non-maskable.

The unimplemented page 2 opcode trap (TRAP) and the SWI are special cases. In one sense, these two exceptions have very low priority, because any enabled interrupt source that is pending prior to the time exception processing begins will take precedence. However, once the CPU begins processing a TRAP or SWI, neither can be interrupted. Also, since these are mutually exclusive instructions, they have no relative priority.

All remaining interrupts are subject to masking via the I bit in the CCR. Most HCS12 microcontroller units (MCU) have an external $\overline{\text{IRQ}}$ pin, which is assigned the highest I-bit interrupt priority and an internal periodic real-time interrupt generator, which has the next highest priority. The other maskable sources have default priorities that follow the address order of the interrupt vectors — the higher the address, the higher the priority of the interrupt. Other maskable interrupts are associated with on-chip peripherals such as timers or serial ports. On the HCS12, logic in the device integration module can give one I-masked source priority over other I-masked sources. Refer to the documentation for the specific HCS12 derivative for more information.

7.4 Resets

M68HC12 devices perform resets with a combination of hardware and software. Integration module circuitry determines the type of reset that has occurred, performs basic system configuration, then passes control to the CPU12. The CPU fetches a vector determined by the type of reset that has occurred, jumps to the address pointed to by the vector, and begins to execute code at that address.

The are four possible sources of reset are:

- Power-on reset (POR)
- External reset ($\overline{\text{RESET}}$ pin)
- COP reset
- Clock monitor reset

Power-on reset (POR) and external reset share the same reset vector. The computer operating properly (COP) reset and the clock monitor reset each have a vector.

7.4.1 Power-On Reset

The HCS12 incorporate circuitry to detect a positive transition in the V_{DD} supply and initialize the device during cold starts, generally by asserting the reset signal internally. The signal is typically released after a delay that allows the device clock generator to stabilize.

7.4.2 External Reset

The MCU distinguishes between internal and external resets by sensing how quickly the signal on the $\overline{\text{RESET}}$ pin rises to logic level 1 after it has been asserted. When the MCU senses any of the four reset conditions, internal circuitry drives the $\overline{\text{RESET}}$ signal low for N clock cycles, then releases. M clock cycles later, the MCU samples the state of the signal applied to the $\overline{\text{RESET}}$ pin. If the signal is still low, an external reset has occurred. If the signal is high, reset is assumed to have been initiated internally by either the COP system or the clock monitor.

7.4.3 COP Reset

The MCU includes a computer operating properly (COP) system to help protect against software failures. When the COP is enabled, software must write a particular code sequence to a specific address to keep a watchdog timer from timing out. If software fails to execute the sequence properly, a reset occurs.

7.4.4 Clock Monitor Reset

The clock monitor circuit uses an internal RC circuit to determine whether clock frequency is above a predetermined limit. If clock frequency falls below the limit when the clock monitor is enabled, a reset occurs.

7.5 Interrupts

Each HCS12 device can recognize a number of interrupt sources. Each source has a vector in the vector table. The $\overline{\text{XIRQ}}$ signal, the unimplemented opcode trap, and the SWI instruction are non-maskable, and have a fixed priority. The remaining interrupt sources can be masked by the I bit. In most devices, the external interrupt request pin is assigned the highest maskable interrupt priority, and the internal periodic real-time interrupt generator has the next highest priority. Other maskable interrupts are associated with on-chip peripherals such as timers or serial ports. These maskable sources have default priorities that follow the address order of the interrupt vectors. The higher the vector address, the higher the priority of the interrupt. On The HCS12, a device integration module incorporates logic that can give any one maskable source priority over other maskable sources.

7.5.1 Non-Maskable Interrupt Request ($\overline{\text{XIRQ}}$)

The $\overline{\text{XIRQ}}$ input is an updated version of the non-maskable interrupt ($\overline{\text{NMI}}$) input of earlier MCUs. The $\overline{\text{XIRQ}}$ function is disabled during system reset and upon entering the interrupt service routine for an $\overline{\text{XIRQ}}$ interrupt.

During reset, both the I bit and the X bit in the CCR are set. This disables maskable interrupts and interrupt service requests made by asserting the $\overline{\text{XIRQ}}$ signal. After minimum system initialization, software can clear the X bit using an instruction such as ANDCC #$BF. Software cannot set the X bit from 0 to 1 once it has been cleared, and interrupt requests made via the $\overline{\text{XIRQ}}$ pin become non-maskable. When a non-maskable interrupt is recognized, both the X and I bits are set after context is saved. The X bit is not affected by maskable interrupts. Execution of an return-from-interrupt (RTI) instruction at the end of the interrupt service routine normally restores the X and I bits to the pre-interrupt request state.

7.5.2 Maskable Interrupts

Maskable interrupt sources include on-chip peripheral systems and external interrupt service requests. Interrupts from these sources are recognized when the global interrupt mask bit (I) in the CCR is cleared. The default state of the I bit out of reset is 1, but it can be written at any time.

The interrupt module manages maskable interrupt priorities. Typically, an on-chip interrupt source is subject to masking by associated bits in control registers in addition to global masking by the I bit in the CCR. Sources generally must be enabled by writing one or more bits in associated control registers. There may be other interrupt-related control bits and flags, and there may be specific register read-write sequences associated with interrupt service. Refer to individual on-chip peripheral descriptions for details.

7.5.3 Interrupt Recognition

Once enabled, an interrupt request can be recognized at any time after the I mask bit is cleared. When an interrupt service request is recognized, the CPU responds at the completion of the instruction being executed. Interrupt latency varies according to the number of cycles required to complete the current instruction. Because the fuzzy logic rule evaluation (REV), fuzzy logic rule evaluation weighted (REVW), and weighted average (WAV) instructions can take many cycles to complete, they are designed so that they can be interrupted. Instruction execution resumes when interrupt execution is complete. When the CPU begins to service an interrupt, the instruction queue is refilled, a return address is calculated, and then the return address and the contents of the CPU registers are stacked as shown in **Table 7-2**.

Table 7-2. Stacking Order on Entry to Interrupts

Memory Location	CPU Registers
SP + 7	RTN_H : RTN_L
SP + 5	Y_H : Y_L
SP + 3	X_H : X_L
SP + 1	B : A
SP	CCR

After the CCR is stacked, the I bit (and the X bit, if an $\overline{\text{XIRQ}}$ interrupt service request caused the interrupt) is set to prevent other interrupts from disrupting the interrupt service routine. Execution continues at the address pointed to by the vector for the highest-priority interrupt that was pending at the beginning of the interrupt sequence. At the end of the interrupt service routine, an RTI instruction restores context from the stacked registers, and normal program execution resumes.

7.5.4 External Interrupts

External interrupt service requests are made by asserting an active-low signal connected to the $\overline{\text{IRQ}}$ pin. Typically, control bits affect how the signal is detected and recognized.

The I bit serves as the $\overline{\text{IRQ}}$ interrupt enable flag. When an $\overline{\text{IRQ}}$ interrupt is recognized, the I bit is set to inhibit interrupts during the interrupt service routine. Before other maskable interrupt requests can be recognized, the I bit must be cleared. This is generally done by an RTI instruction at the end of the service routine.

7.5.5 Return-from-Interrupt Instruction (RTI)

RTI is used to terminate interrupt service routines. RTI is an 8-cycle instruction when no other interrupt is pending and 11 cycles (10 cycles in M68HC12) when another interrupt is pending. In either case, the first five cycles are used to restore (pull) the CCR, B:A, X, Y, and the return address from the stack. If no other interrupt is pending at this point, three program words are fetched to refill the instruction queue from the area of the return address and processing proceeds from there.

If another interrupt is pending after registers are restored, a new vector is fetched, and the stack pointer is adjusted to point at the CCR value that was just recovered (SP = SP – 9). This makes it appear that the registers have been stacked again. After the SP is adjusted, three program words are fetched to refill the instruction queue, starting at the address the vector points to. Processing then continues with execution of the instruction that is now at the head of the queue.

7.6 Unimplemented Opcode Trap

The CPU12 has opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the 202 unused opcodes on page 2, an unimplemented opcode trap occurs. The 202 unimplemented opcodes are essentially interrupts that share a common interrupt vector, $FFF8:$FFF9.

The CPU12 uses the next address after an unimplemented page 2 opcode as a return address. This differs from the M68HC11 illegal opcode interrupt, which uses the address of an illegal opcode as the return address. In the

CPU12, the stacked return address can be used to calculate the address of the unimplemented opcode for software-controlled traps.

7.7 Software Interrupt Instruction (SWI)

Execution of the SWI instruction causes an interrupt without an interrupt service request. SWI is not inhibited by the global mask bits in the CCR, and execution of SWI sets the I mask bit. Once an SWI interrupt begins, maskable interrupts are inhibited until the I bit in the CCR is cleared. This typically occurs when an RTI instruction at the end of the SWI service routine restores context.

7.8 Exception Processing Flow

The first cycle in the exception processing flow for all CPU12 exceptions is the same, regardless of the source of the exception. Between the first and second cycles of execution, the CPU chooses one of three alternative paths. The first path is for resets, the second path is for pending X or I interrupts, and the third path is used for software interrupts (SWI) and trapping unimplemented opcodes. The last two paths are virtually identical, differing only in the details of calculating the return address. Refer to **Figure 7-1** for the following discussion.

7.8.1 Vector Fetch

The first cycle of all exception processing, regardless of the cause, is a vector fetch. The vector points to the address where exception processing will continue. Exception vectors are stored in a table located at the top of the memory map ($FFxx). The CPU cannot use the fetched vector until the third cycle of the exception processing sequence.

During the vector fetch cycle, the CPU issues a signal that tells the interrupt module to drive the vector address of the highest priority, pending exception onto the system address bus (the CPU does not provide this address).

After the vector fetch, the CPU selects one of the three alternate execution paths, depending upon the cause of the exception.

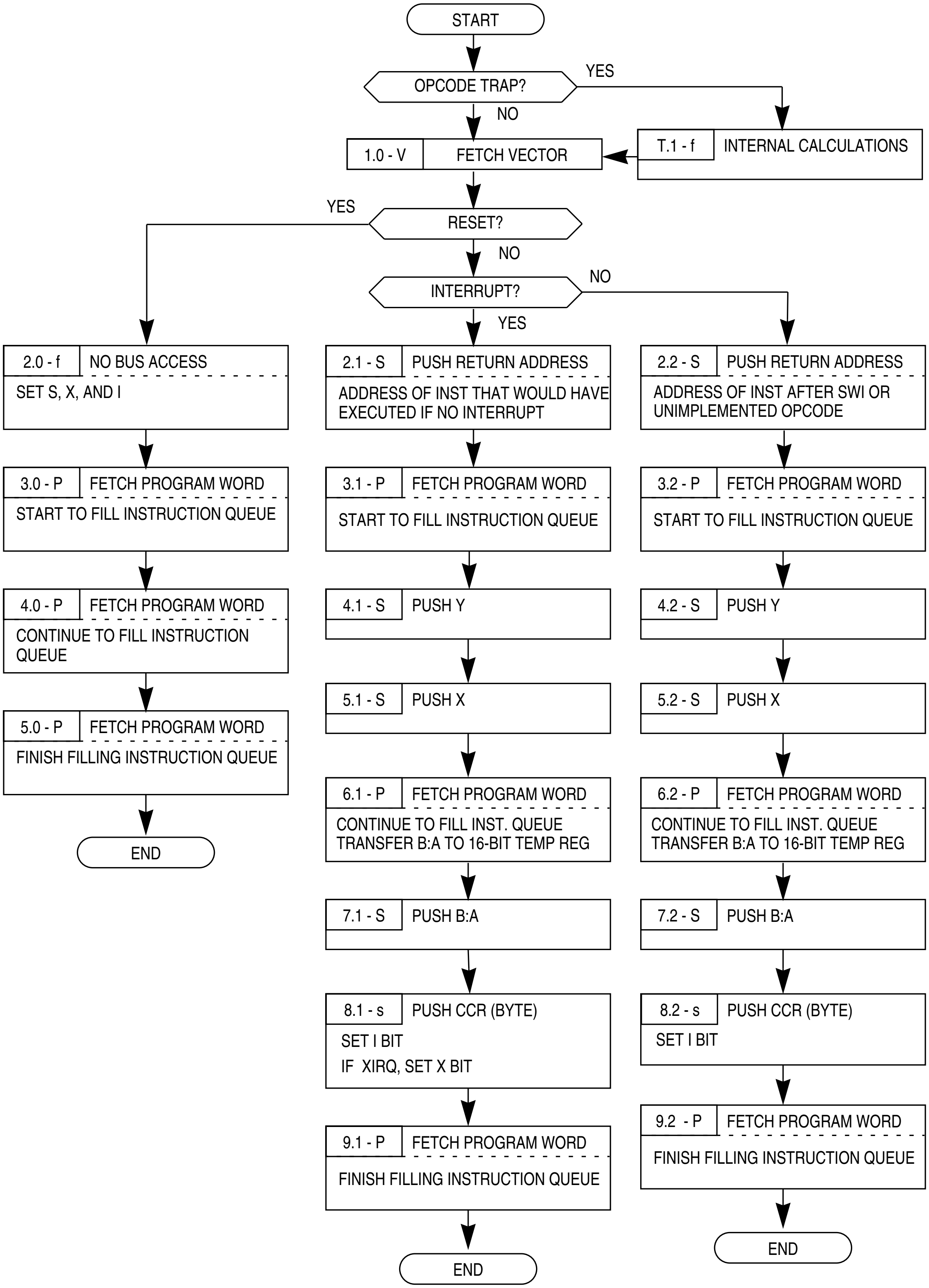


Figure 7-1. Exception Processing Flow Diagram

7.8.2 Reset Exception Processing

If reset caused the exception, processing continues to cycle 2.0. This cycle sets the S, X, and I bits in the CCR. Cycles 3.0 through 5.0 are program word fetches that refill the instruction queue. Fetches start at the address pointed to by the reset vector. When the fetches are completed, exception processing ends, and the CPU starts executing the instruction at the head of the instruction queue.

7.8.3 Interrupt and Unimplemented Opcode Trap Exception Processing

If an exception was not caused by a reset, a return address is calculated.

- Cycles 2.1and 2.2 are both S cycles (stack a 16-bit word), but the CPU12 performs different return address calculations for each type of exception.
  - When an X- or I-related interrupt causes the exception, the return address points to the next instruction that would have been executed had processing not been interrupted.
  - When an exception is caused by an SWI opcode or by an unimplemented opcode (see **7.6 Unimplemented Opcode Trap**), the return address points to the next address after the opcode.
- Once calculated, the return address is pushed onto the stack.
- Cycles 3.1 through 9.1 are identical to cycles 3.2 through 9.2 for the rest of the sequence, except for optional setting of the X mask bit performed in cycle 8.1 (see below).
- Cycle 3.1/3.2 is the first of three program word fetches that refill the instruction queue.
- Cycle 4.1/4.2 pushes Y onto the stack.
- Cycle 5.1/5.2 pushes X onto the stack.
- Cycle 6.1/6.2 is the second of three program word fetches that refill the instruction queue. During this cycle, the contents of the A and B accumulators are concatenated into a 16-bit word in the order B:A. This makes register order in the stack frame the same as that of the M68HC11, M6801, and the M6800.
- Cycle 7.1/7.2 pushes the 16-bit word containing B:A onto the stack.

- Cycle 8.1/8.2 pushes the 8-bit CCR onto the stack, then updates the mask bits.
  - When an $\overline{\text{XIRQ}}$ interrupt causes an exception, both X and I are set, which inhibits further interrupts during exception processing.
  - When any other interrupt causes an exception, the I bit is set, but the X bit is not changed.
- Cycle 9.1/9.2 is the third of three program word fetches that refill the instruction queue. It is the last cycle of exception processing. After this cycle the CPU starts executing the first cycle of the instruction at the head of the instruction queue.

Section 8. Instruction Queue

8.1 Introduction

This section describes development and debug support features related to the central processor unit (CPU12). Topics include:

- Single-wire background debug interface
- Hardware breakpoint system
- Instruction queue operation and reconstruction
- Instruction tagging

1 = Valid Data

TRACE — Trace Flag

Indicates when tracing is enabled. Firmware in the BDM ROM sets TRACE in response to a TRACE1 command and TRACE is cleared upon completion of the TRACE1 command. Do not attempt to write TRACE directly with WRITE_BD_BYTE commands.

0 = Tracing not enabled
1 = TRACE1 command in progress

8.2 External Reconstruction of the Queue

The CPU12 uses an instruction queue to buffer program information and increase instruction throughput. The HCS12 implements the queue somewhat differently from the original M68HC12. The HCS12 queue consists of three 16-bit stages while the M68HC12 queue consists of two 16-bit stages, plus a 16-bit holding latch. Program information is always fetched in aligned 16-bit words. At least three bytes of program information are available to the CPU when instruction execution begins. The holding latch in the M68HC12 is used when a word of program information arrives before the queue can advance.

Because of the queue, program information is fetched a few cycles before it is used by the CPU. Internally, the microcontroller unit (MCU) only needs to buffer the fetched data. But, in order to monitor cycle-by-cycle CPU activity externally, it is necessary to capture data and address to discern what is happening in the instruction queue.

Two external pins, IPIPE1 and IPIPE0, provide time-multiplexed information about data movement in the queue and instruction execution. The instruction queue and cycle-by-cycle activity can be reconstructed in real time or from trace history captured by a logic analyzer. However, neither scheme can be used to stop the CPU12 at a specific instruction. By the time an operation is visible outside the MCU, the instruction has already begun execution. A separate instruction tagging mechanism is provided for this purpose. A tag follows the information in the queue as the queue is advanced. During debugging, the CPU enters active background debug mode when a tagged instruction reaches the head of the queue, rather than executing the tagged instruction. For more information about tagging, refer to **8.6 Instruction Tagging**.

8.3 Instruction Queue Status Signals

The IPIPE1 and IPIPE0 signals carry time-multiplexed information about data movement and instruction execution during normal CPU operation. The signals are available on two multifunctional device pins. During reset, the pins are used as mode-select input signals MODA and MODB.

To reconstruct the queue, the information carried by the status signals must be captured externally. In general, data movement and execution start information are considered to be distinct 2-bit values, with the low-order bit on IPIPE0 and the high-order bit on IPIPE1.

8.3.1 HCS12 Timing Detail

In the HCS12, data-movement information is available when E clock is high or on falling edges of the E clock; execution-start information is available when E clock is low or on rising edges of the E clock, as shown in **Figure 8-1**. Data-movement information refers to data on the bus. Execution-start information refers to the bus cycle that starts with that E-low time and continues through the following E-high time. **Table 8-1** summarizes the information encoded on the IPIPE1 and IPIPE0 pins.

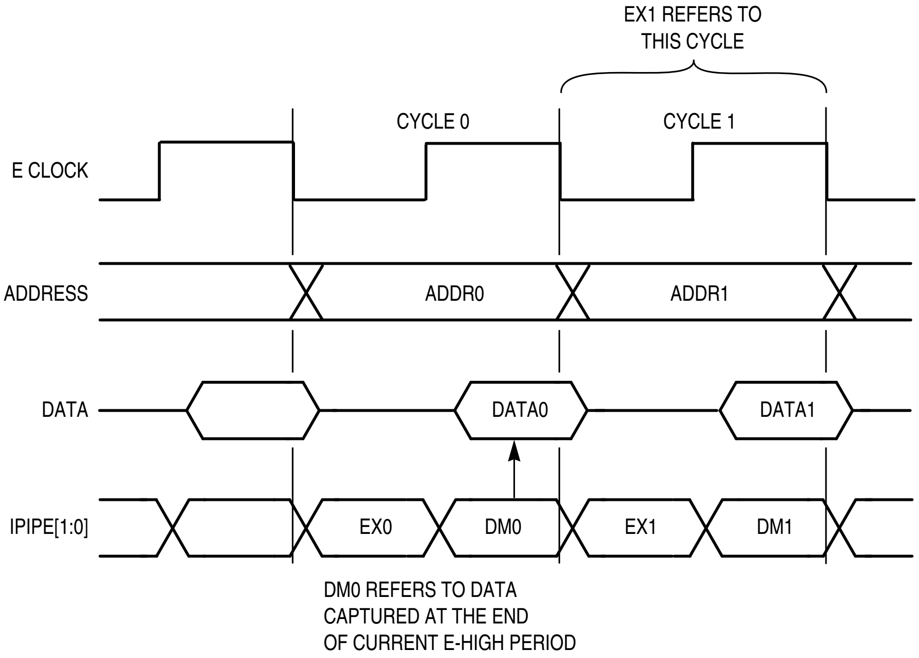


Figure 8-1. Queue Status Signal Timing (HCS12)

8.3.2 M68HC12 Timing Detail

In the M68HC12, data movement information is available on rising edges of the E clock; execution start information is available on falling edges of the E clock, as shown in **Figure 8-2**. Data movement information refers to data on the bus at the previous falling edge of E. Execution information refers to the bus cycle from the current falling edge to the next falling edge of E. **Table 8-1** summarizes the information encoded on the IPIPE1 and IPIPE0 pins.

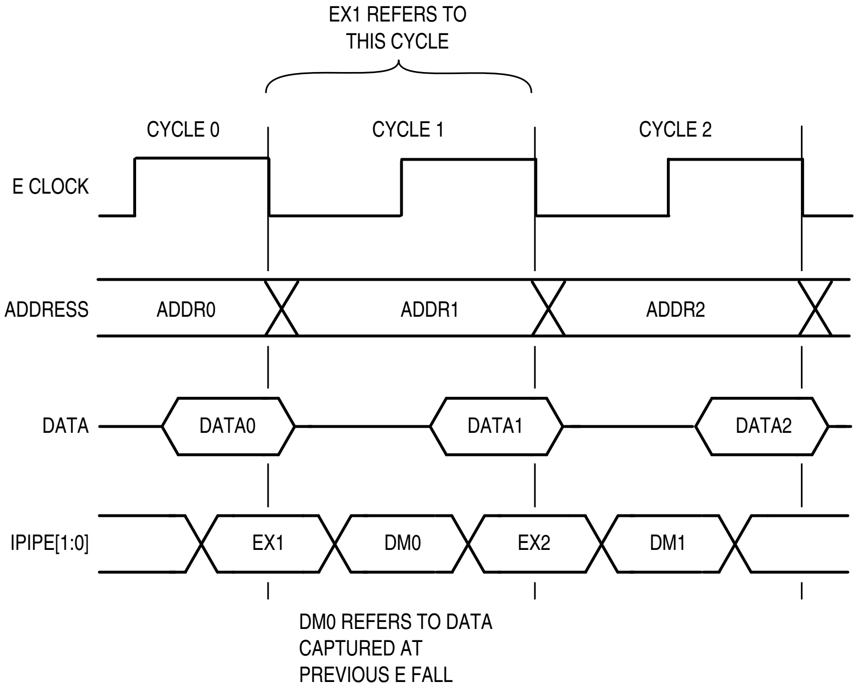


Figure 8-2. Queue Status Signal Timing (M68HC12)

Table 8-1. IPIPE1 and IPIPE0 Decoding (HCS12 and M68HC12)

	Mnemonic	**Meaning**
Data Movement	**Capture at E Fall in HCS12 (E Rise in M68HC12)**	
0:0	—	No movement
0:1	LAT[1]	Latch data from bus
1:0	ALD	Advance queue and load from bus
1:1	ALL[1]	Advance queue and load from latch
Execution Start	**Capture at E Rise in HCS12 (E Fall in M68HC12)**	
0:0	—	No start
0:1	INT	Start interrupt sequence
1:0	SEV	Start even instruction
1:1	SOD	Start odd instruction

1. The HCS12 implementation does not include a holding latch, so these data movement codes are used only in the original M68HC12.

8.3.3 Null (Code 0:0)

The 0:0 data movement state indicates that there was no data movement in the instruction queue; the 0:0 execution start state indicates continuation of an instruction or interrupt sequence (no new instruction or interrupt start).

8.3.4 LAT — Latch Data from Bus (Code 0:1)

This code is not used in the HCS12. In the M68HC12, fetched program information has arrived, but the queue is not ready to advance. The information is latched into a buffer. Later, when the queue does advance, stage 1 is refilled from the buffer or from the data bus if the buffer is empty. In some instruction sequences, there can be several latch cycles before the queue advances. In these cases, the buffer is filled on the first latch event and additional latch requests are ignored.

8.3.5 ALD — Advance and Load from Data Bus (Code 1:0)

The instruction queue is advanced by one word and stage one is refilled with a word of program information from the data bus. The CPU requested the information two bus cycles earlier but, due to access delays, the information was not available until the E cycle referred to by the ALD code.

8.3.6 ALL — Advance and Load from Latch (Code 1:1)

This code is not used in the HCS12. In the M68HC12, the 2-stage instruction queue is advanced by one word and stage one is refilled with a word of program information from the buffer. The information was latched from the data bus at the falling edge of a previous E cycle because the instruction queue was not ready to advance when it arrived.

8.3.7 INT — Interrupt Sequence Start (Code 0:1)

The E cycle associated with this code is the first cycle of an interrupt sequence. Normally, this cycle is a read of the interrupt vector. However, in systems that have interrupt vectors in external memory and an 8-bit data bus, this cycle reads the upper byte of the 16-bit interrupt vector.

8.3.8 SEV — Start Instruction on Even Address (Code 1:0)

The E cycle associated with this code is the first cycle of the instruction in the even (high order) half of the word at the head of the instruction queue. The queue treats the $18 prebyte for instructions on page 2 of the opcode map as a special 1-byte, 1-cycle instruction, except that interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

8.3.9 SOD — Start Instruction on Odd Address (Code 1:1)

The E cycle associated with this code is the first cycle of the instruction in the odd (low order) half of the word at the head of the instruction queue. The queue treats the $18 prebyte for instructions on page 2 of the opcode map as a special 1-byte, 1-cycle instruction, except that interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

8.4 Queue Reconstruction (for HCS12)

The raw signals required for queue reconstruction are the address bus (ADDR), the data bus (DATA), the system clock (E), and the queue status signals (IPIPE1 and IPIPE2). An ALD data movement implies a read; therefore, it is not necessary to capture the $R/\overline{W}$ signal. An E clock cycle begins at a falling edge of E. Addresses and execution status must be captured at the rising E edge in the middle of the cycle. Data and data-movement status must be captured at the falling edge of E at the end of the cycle. These captures can then be organized into records with one record per E clock cycle.

Implementation details depend on the type of MCU and the mode of operation. For instance, the data bus can be eight bits or 16 bits wide, and nonmultiplexed or multiplexed. In all cases, the externally reconstructed queue must use 16-bit words. Demultiplexing and assembly of 8-bit data into 16-bit words is done before program information enters the real queue, so it must also be done for the external reconstruction.

An example:

> Systems with an 8-bit data bus and a program stored in external memory require two cycles for each program word fetch. MCU bus-control logic freezes the CPU clocks long enough to do two 8-bit accesses rather than

a single 16-bit access, so the CPU sees only 16-bit words of program information. To recover the 16-bit program words externally, latch the data bus state at the falling edge of E when ADDR0 = 0, and gate the outputs of the latch onto DATA[15:8] when an ALD cycle occurs. Since the 8-bit data bus is connected to DATA[7:0], the 16-bit word on the data lines corresponds to the ALD during the last half of the second 8-bit fetch, which is always to an odd address. IPIPE[1:0] status signals indicate 0:0 for the second half of the E cycle corresponding to the first 8-bit fetch.

Some MCUs have address lines to support memory expansion beyond the standard 64-Kbyte address space. When memory expansion is used, expanded addresses must also be captured and maintained.

8.4.1 Queue Reconstruction Registers (for HCS12)

Queue reconstruction requires the following registers, which can be implemented as software variables when previously captured trace data is used, or as hardware latches in real time.

8.4.1.1 fetch_add Register

This register buffers the fetch address.

8.4.1.2 st1_add, st1_dat Registers

These registers contain address and data for the first stage of the reconstructed instruction queue.

8.4.1.3 st2_add, st2_dat Registers

These registers contain address and data for the middle stage of the reconstructed instruction queue.

8.4.1.4 st3_add, st3_dat Registers

These registers contain address and data for the final stage of the reconstructed instruction queue. When the IPIPE[1:0] signals indicate the execution status, the address and opcode can be found in these registers.

8.4.2 Reconstruction Algorithm (for HCS12)

This section describes how to use IPIPE[1:0] signals and queue reconstruction registers to reconstruct the queue.

Typically, the first few cycles of raw capture data are not useful because it takes several cycles before an instruction propagates to the head of the queue. During these first raw cycles, the only meaningful information available is data movement signals. Information on the external address and data buses during this setup time is still captured and propagated through the reconstructed queue, but the information reflects the actions of instructions that were fetched before data collection started.

In the special case of a reset, there is a five-cycle sequence (`VfPPP`) during which the reset vector is fetched and the instruction queue is filled, before execution of the first instruction begins. Due to the timing of the switchover of the IPIPE[1:0] pins from their alternate function as mode-select inputs, the status information on these two pins may be erroneous during the first cycle or two after the release of reset. This is not a problem because the status is correct in time for queue reconstruction logic to correctly replicate the queue.

On an advance-and-load-from-data-bus (ALD) cycle, the information in the instruction queue must advance by one stage. Whatever was in stage three of the queue simply disappears. The previous contents of stage two go to stage three, the previous contents of stage one go to stage two, and the contents of fetch_add and data from the current cycle go to stage one.

Figure 8-3 shows the reset sequence and illustrates the relationship between instruction cycle codes (`VfPPP`) and pipe status signals. One cycle of the data bus is shown to indicate the relationship between the ALD data movement code and the data value it refers to. The SEV execution start code indicates that the reset vector pointed to an even address in this example.

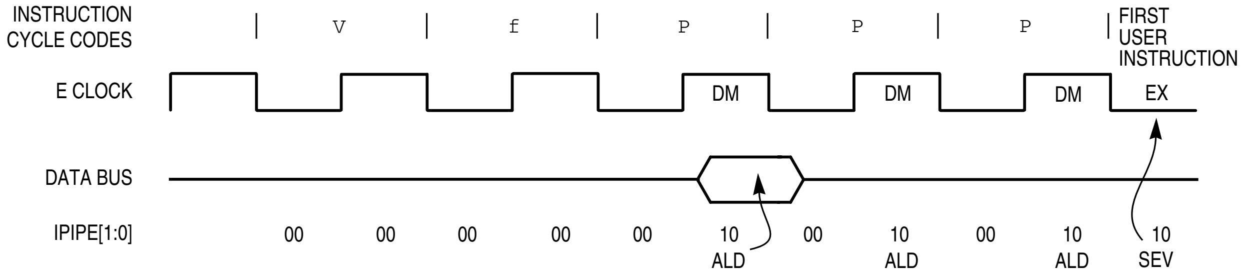


Figure 8-3. Reset Sequence for HCS12

8.5 Queue Reconstruction (for M68HC12)

The raw signals required for queue reconstruction are the address bus (ADDR), the data bus (DATA), the system clock (E), and the queue status signals (IPIPE1 and IPIPE0). An E-clock cycle begins after an E fall. Addresses and data movement status must be captured at the E rise in the middle of the cycle. Data and execution start status must be captured at the E fall at the end of the cycle. These captures can then be organized into records with one record per E clock cycle.

Implementation details depend upon the type of device and the mode of operation. For instance, the data bus can be eight bits or 16 bits wide, and non-multiplexed or multiplexed. In all cases, the externally reconstructed queue must use 16-bit words. Demultiplexing and assembly of 8-bit data into 16-bit words is done before program information enters the real queue, so it must also be done for the external reconstruction.

An example:

> Systems with an 8-bit data bus and a program stored in external memory require two cycles for each program word fetch. MCU bus control logic freezes the CPU clocks long enough to do two 8-bit accesses rather than a single 16-bit access, so the CPU sees only 16-bit words of program information. To recover the 16-bit program words externally, latch the data bus state at the falling edge of E when ADDR0 = 0, and gate the outputs of the latch onto DATA[15:8] when a LAT or ALD cycle occurs. Since the 8-bit data bus is connected to DATA[7:0], the 16-bit word on the data lines corresponds to the ALD or LAT status indication at the E rise after the second 8-bit fetch, which is always to an odd address. IPIPE1 and IPIPE0 status signals indicate 0:0 at the beginning (E fall) and middle (E rise) of the first 8-bit fetch.

Some M68HC12 devices have address lines to support memory expansion beyond the standard 64-Kbyte address space. When memory expansion is used, expanded addresses must also be captured and maintained.

8.5.1 Queue Reconstruction Registers (for M68HC12)

Queue reconstruction requires these registers, which can be implemented as software variables when previously captured trace data is used or as hardware latches in real time.

8.5.1.1 in_add, in_dat Registers

These registers contain the address and data from the previous external bus cycle. Depending on how records are read and processed from the raw capture information, it may be possible to simply read this information from the raw capture data file when needed.

8.5.1.2 fetch_add, fetch_dat Registers

These registers buffer address and data for information that was fetched before the queue was ready to advance.

8.5.1.3 st1_add, st1_dat Registers

These registers contain address and data for the first stage of the reconstructed instruction queue.

8.5.1.4 st2_add, st2_dat Registers

These registers contain address and data for the final stage of the reconstructed instruction queue. When the IPIPE1 and IPIPE0 signals indicate that an instruction is starting to execute, the address and opcode can be found in these registers.

8.5.2 Reconstruction Algorithm (for M68HC12)

This subsection describes in detail how to use IPIPE1 and IPIPE0 signals and queue reconstruction registers to reconstruct the queue. An “is_full” flag is used to indicate when the fetch_add and fetch_dat buffer registers contain information. The use of the flag is explained more fully in subsequent paragraphs.

Typically, the first few cycles of raw capture data are not useful because it takes several cycles before an instruction propagates to the head of the queue. During these first raw cycles, the only meaningful information available are data movement signals. Information on the external address and data buses during this setup time reflects the actions of instructions that were fetched before data collection started.

In the special case of a reset, there is a 5-cycle sequence (`VfPPP`) during which the reset vector is fetched and the instruction queue is filled, before execution of the first instruction begins. Due to the timing of the switchover of the IPIPE1 and IPIPE0 pins from their alternate function as mode select inputs, the status information on these two pins may be erroneous during the first cycle or two after the release of reset. This is not a problem because the status is correct in time for queue reconstruction logic to correctly replicate the queue.

Before starting to reconstruct the queue, clear the is_full flag to indicate that there is no meaningful information in the fetch_add and fetch_dat buffers. Further movement of information in the instruction queue is based on the decoded status on the IPIPE1 and IPIPE0 signals at the rising edges of E.

8.5.2.1 LAT Decoding

On a latch cycle (LAT), check the is_full flag. If and only if is_full = 0, transfer the address and data from the previous bus cycle (in_add and in_dat) into the fetch_add and fetch_dat registers, respectively. Then, set the is_full flag. The usual reason for a latch request instead of an advance request is that the previous instruction ended with a single aligned byte of program information in the last stage of the instruction queue. Since the odd half of this word still holds the opcode for the next instruction, the queue cannot advance on this cycle. However, the cycle to fetch the next word of program information has already started and the data is on its way.

8.5.2.2 ALD Decoding

On an advance-and-load-from-data-bus (ALD) cycle, the information in the instruction queue must advance by one stage. Whatever was in stage 2 of the queue is simply thrown away. The previous contents of stage 1 are moved to stage 2, and the address and data from the previous cycle (in_add and in_dat) are transferred into stage 1 of the instruction queue. Finally, clear the is_full flag to indicate the buffer latch is ready for new data. Usually, there would be no useful information in the fetch buffer when an ALD cycle was encountered, but in the case of a change-of-flow, any data that was there needs to be flushed out (by clearing the is_full flag).

8.5.2.3 ALL Decoding

On an advance-and-load-from-latch (ALL) cycle, the information in the instruction queue must advance by one stage. Whatever was in stage 2 of the queue is simply thrown away. The previous contents of stage 1 are moved to stage 2, and the contents of the fetch buffer latch are transferred into stage 1 of the instruction queue. One or more cycles preceding the ALL cycle will have been a LAT cycle. After updating the instruction queue, clear the is_full flag to indicate the fetch buffer is ready for new information.

Figure 8-4 shows the reset sequence and illustrates the relationship between instruction cycle codes (`VfPPP`) and pipe status signals. One cycle of the data bus is shown to indicate the relationship between the ALD data movement code and the data value it refers to. The SEV execution start code indicates that the reset vector pointed to an even address in this example.

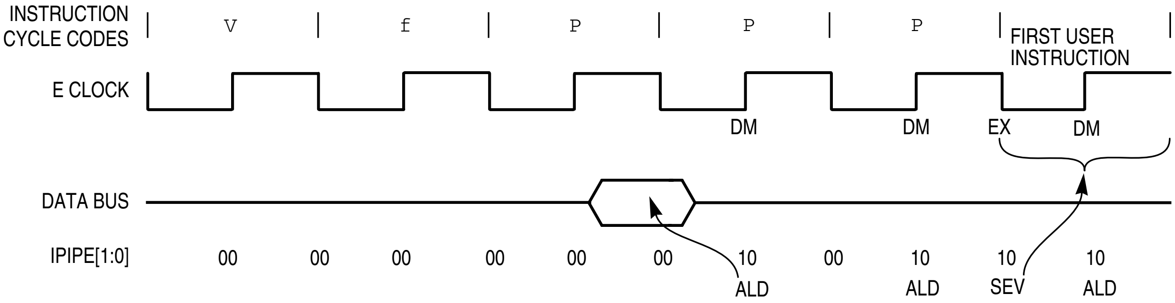


Figure 8-4. Reset Sequence for M68HC12

8.6 Instruction Tagging

The instruction queue and cycle-by-cycle CPU activity can be reconstructed in real time or from trace history that was captured by a logic analyzer. However, the reconstructed queue cannot be used to stop the CPU at a specific instruction, because execution has already begun by the time an operation is visible outside the MCU. A separate instruction tagging mechanism is provided for this purpose.

Executing the BDM TAGGO command configures two MCU pins for tagging. The TAGLO signal shares a pin with the LSTRB signal, and the TAGHI signal shares the BKGD pin. Tagging information is latched on the falling edge of ECLK, as shown in **Figure 8-5**.

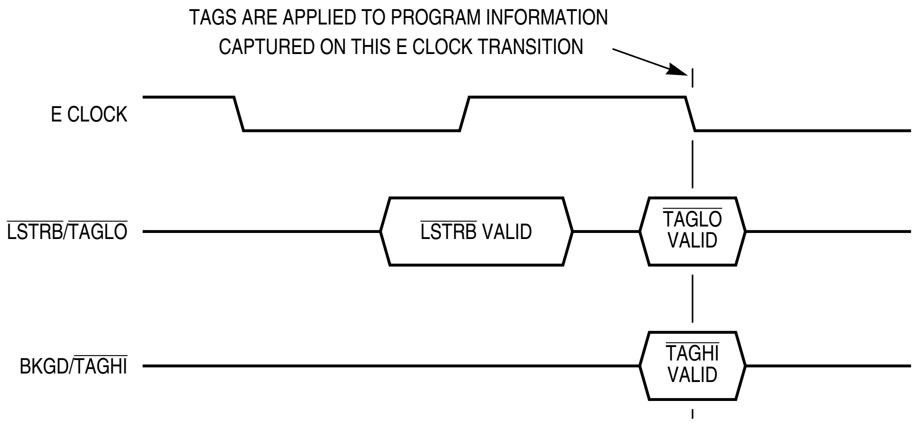

TAGS ARE APPLIED TO PROGRAM INFORMATION
CAPTURED ON THIS E CLOCK TRANSITION
E CLOCK
LSTRB/TAGLO
LSTRB VALID
TAGLO VALID
BKGD/TAGHI
TAGHI VALID

Figure 8-5. Tag Input Timing

Table 8-2 shows the functions of the two independent tagging pins. The presence of logic level 0 on either pin at the fall of ECLK tags (marks) the associated byte of program information as it is read into the instruction queue. Tagging is allowed in all modes. Tagging is disabled when BDM becomes active.

Table 8-2. Tag Pin Function

TAGHI	TAGLO	Tag
1	1	No tag
1	0	Low byte
0	1	High byte
0	0	Both bytes

In HCS12 and M68HC12 derivatives that have hardware breakpoint capability, the breakpoint control logic and BDM control logic use the same internal signals for instruction tagging. The CPU does not differentiate between the two kinds of tags.

The tag follows program information as it advances through the queue. When a tagged instruction reaches the head of the queue, the CPU enters active background debug mode rather than executing the instruction.

Section 9. Fuzzy Logic Support

9.1 Introduction

The instruction set of the central processor unit (CPU12) is the first instruction set to specifically address the needs of fuzzy logic. This section describes the use of fuzzy logic in control systems, discusses the CPU12 fuzzy logic instructions, and provides examples of fuzzy logic programs.

The CPU12 includes four instructions that perform specific fuzzy logic tasks. In addition, several other instructions are especially useful in fuzzy logic programs. The overall C-friendliness of the instruction set also aids development of efficient fuzzy logic programs.

This section explains the basic fuzzy logic algorithm for which the four fuzzy logic instructions are intended. Each of the fuzzy logic instructions are then explained in detail. Finally, other custom fuzzy logic algorithms are discussed, with emphasis on use of other CPU12 instructions.

The four fuzzy logic instructions are:

- MEM (determine grade of membership), which evaluates trapezoidal membership functions
- REV (fuzzy logic rule evaluation) and REVW (fuzzy logic rule evaluation weighted), which perform unweighted or weighted MIN-MAX rule evaluation
- WAV (weighted average), which performs weighted average defuzzification on singleton output membership functions.

Other instructions that are useful for custom fuzzy logic programs include:

- MINA (place smaller of two unsigned 8-bit values in accumulator A)
- EMIND (place smaller of two unsigned 16-bit values in accumulator D)
- MAXM (place larger of two unsigned 8-bit values in memory)
- EMAXM (place larger of two unsigned 16-bit values in memory)
- TBL (table lookup and interpolate)
- ETBL (extended table lookup and interpolate)
- EMACS (extended multiply and accumulate signed 16-bit by 16-bit to 32-bit)

For higher resolution fuzzy programs, the fast extended precision math instructions in the CPU12 are also beneficial. Flexible indexed addressing modes help simplify access to fuzzy logic data structures stored as lists or tabular data structures in memory.

The actual logic additions required to implement fuzzy logic support in the CPU12 are quite small, so there is no appreciable increase in cost for the typical user. A fuzzy inference kernel for the CPU12 requires one-fifth as much code space and executes almost 50 times faster than a comparable kernel implemented on a typical midrange microcontroller.

9.2 Fuzzy Logic Basics

This is an overview of basic fuzzy logic concepts. It can serve as a general introduction to the subject, but that is not the main purpose. There are a number of fuzzy logic programming strategies. This discussion concentrates on the methods implemented in the CPU12 fuzzy logic instructions. The primary goal is to provide a background for a detailed explanation of the CPU12 fuzzy logic instructions.

In general, fuzzy logic provides for set definitions that have fuzzy boundaries rather than the crisp boundaries of Aristotelian logic. These sets can overlap so that, for a specific input value, one or more sets associated with linguistic labels may be true to a degree at the same time. As the input varies from the range of one set into the range of an adjacent set, the first set becomes progressively less true while the second set becomes progressively more true.

Fuzzy logic has membership functions which emulate human concepts like “temperature is warm”; that is, conditions are perceived to have gradual boundaries. This concept seems to be a key element of the human ability to solve certain types of complex problems that have eluded traditional control methods.

Fuzzy sets provide a means of using linguistic expressions like "temperature is warm" in rules which can then be evaluated with a high degree of numerical precision and repeatability. This directly contradicts the common misperception that fuzzy logic produces approximate results — a specific set of input conditions always produces the same result, just as a conventional control system does.

A microcontroller-based fuzzy logic control system has two parts:

- A fuzzy inference kernel which is executed periodically to determine system outputs based on current system inputs
- A knowledge base which contains membership functions and rules

Figure 9-1 is a block diagram of this kind of fuzzy logic system.

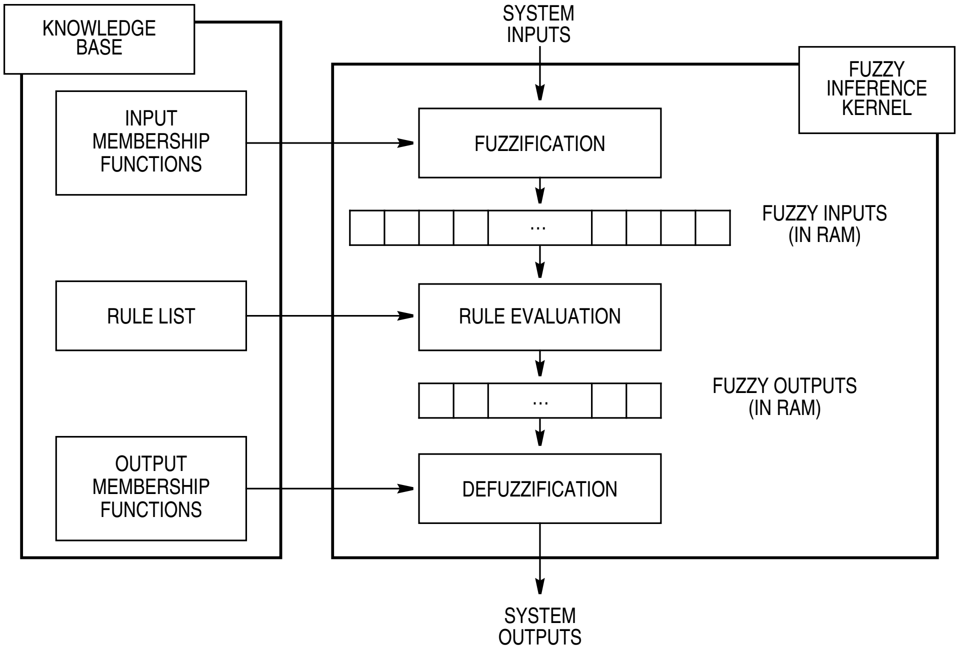


Figure 9-1. Block Diagram of a Fuzzy Logic System

The knowledge base can be developed by an application expert without any microcontroller programming experience. Membership functions are simply expressions of the expert's understanding of the linguistic terms that describe the system to be controlled. Rules are ordinary language statements that describe the actions a human expert would take to solve the application problem.

Rules and membership functions can be reduced to relatively simple data structures (the knowledge base) stored in non-volatile memory. A fuzzy inference kernel can be written by a programmer who does not know how the application system works. The only thing the programmer needs to do with knowledge base information is store it in the memory locations used by the kernel.

One execution pass through the fuzzy inference kernel generates system output signals in response to current input conditions. The kernel is executed as often as needed to maintain control. If the kernel is executed more often than needed, processor bandwidth and power are wasted; delaying too long between passes can cause the system to get too far out of control. Choosing a periodic rate for a fuzzy control system is the same as it would be for a conventional control system.

9.2.1 Fuzzification (MEM)

During the fuzzification step, the current system input values are compared against stored input membership functions to determine the degree to which each label of each system input is true. This is accomplished by finding the y-value for the current input value on a trapezoidal membership function for each label of each system input. The MEM instruction in the CPU12 performs this calculation for one label of one system input. To perform the complete fuzzification task for a system, several MEM instructions must be executed, usually in a program loop structure.

Figure 9-2 shows a system of three input membership functions, one for each label of the system input. The x-axis of all three membership functions represents the range of possible values of the system input. The vertical line through all three membership functions represents a specific system input value. The y-axis represents degree of truth and varies from completely false ($00 or 0 percent) to completely true ($FF or 100 percent). The y-value where the vertical line intersects each of the membership functions, is the degree to which the current input value matches the associated label for this system input. For example, the expression "temperature is warm" is 25 percent true ($40). The value $40 is stored to a random-access memory (RAM) location and is called a fuzzy input (in this case, the fuzzy input for “temperature is warm”). There is a RAM location for each fuzzy input (for each label of each system input).

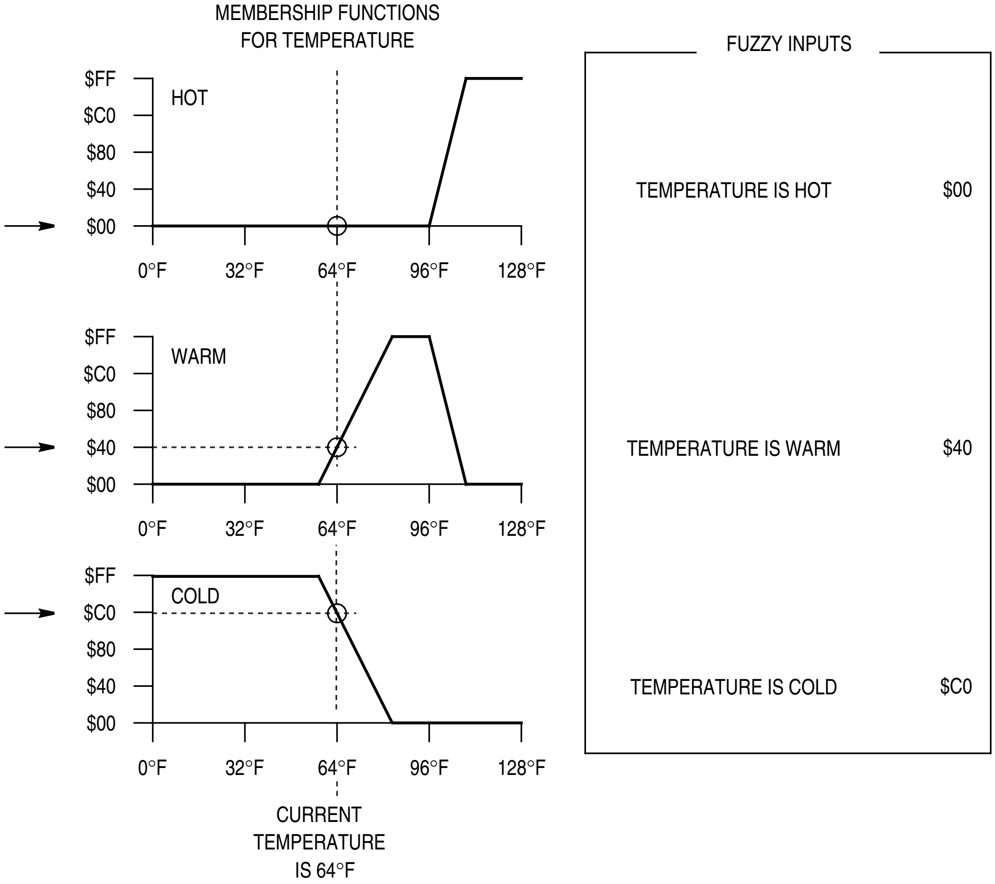


Figure 9-2. Fuzzification Using Membership Functions

When the fuzzification step begins, the current value of the system input is in an accumulator of the CPU12, one index register points to the first membership function definition in the knowledge base, and a second index register points to the first fuzzy input in RAM. As each fuzzy input is calculated by executing a MEM instruction, the result is stored to the fuzzy input and both pointers are updated automatically to point to the locations associated with the next fuzzy input. The MEM instruction takes care of everything except counting the number of labels per system input and loading the current value of any subsequent system inputs.

The end result of the fuzzification step is a table of fuzzy inputs representing current system conditions.

9.2.2 Rule Evaluation (REV and REVW)

Rule evaluation is the central element of a fuzzy logic inference program. This step processes a list of rules from the knowledge base using current fuzzy input values from RAM to produce a list of fuzzy outputs in RAM. These fuzzy outputs can be thought of as raw suggestions for what the system output should be in response to the current input conditions. Before the results can be applied, the fuzzy outputs must be further processed, or defuzzified, to produce a single output value that represents the combined effect of all of the fuzzy outputs.

The CPU12 offers two variations of rule evaluation instructions. The REV instruction provides for unweighted rules (all rules are considered to be equally important). The REVW instruction is similar but allows each rule to have a separate weighting factor which is stored in a separate parallel data structure in the knowledge base. In addition to the weights, the two rule evaluation instructions also differ in the way rules are encoded into the knowledge base.

An understanding of the structure and syntax of rules is needed to understand how a microcontroller performs the rule evaluation task. An example of a typical rule is:

> If temperature is warm and pressure is high, then heat is
> (should be) off.

At first glance, it seems that encoding this rule in a compact form understandable to the microcontroller would be difficult, but it is actually simple to reduce the rule to a small list of memory pointers. The antecedent portion of the rule is a statement of input conditions and the consequent portion of the rule is a statement of output actions.

The antecedent portion of a rule is made up of one or more (in this case two) antecedents connected by a fuzzy *and* operator. Each antecedent expression consists of the name of a system input, followed by *is*, followed by a label name. The label must be defined by a membership function in the knowledge base. Each antecedent expression corresponds to one of the fuzzy inputs in RAM. Since *and* is the only operator allowed to connect antecedent expressions, there is no need to include these in the encoded rule. The antecedents can be encoded as a simple list of pointers to (or addresses of) the fuzzy inputs to which they refer.

The consequent portion of a rule is made up of one or more (in this case one) consequents. Each consequent expression consists of the name of a system output, followed by *is*, followed by a label name. Each consequent expression corresponds to a specific fuzzy output in RAM. Consequents for a rule can be encoded as a simple list of pointers to (or addresses of) the fuzzy outputs to which they refer.

The complete rules are stored in the knowledge base as a list of pointers or addresses of fuzzy inputs and fuzzy outputs. For the rule evaluation logic to work, there must be some means of knowing which pointers refer to fuzzy inputs and which refer to fuzzy outputs. There also must be a way to know when the last rule in the system has been reached.

- One method of organization is to have a fixed number of rules with a specific number of antecedents and consequents.
- A second method, employed in Freescale Freeware M68HC11 kernels, is to mark the end of the rule list with a reserved value, and use a bit in the pointers to distinguish antecedents from consequents.
- A third method of organization, used in the CPU12, is to mark the end of the rule list with a reserved value, and separate antecedents and consequents with another reserved value. This permits any number of rules, and allows each rule to have any number of antecedents and consequents, subject to the limits imposed by availability of system memory.

Each rule is evaluated sequentially, but the rules as a group are treated as if they were all evaluated simultaneously. Two mathematical operations take place during rule evaluation. The fuzzy *and* operator corresponds to the mathematical minimum operation and the fuzzy *or* operation corresponds to the mathematical maximum operation. The fuzzy *and* is used to connect antecedents within a rule. The fuzzy *or* is implied between successive rules. Before evaluating any rules, all fuzzy outputs are set to zero (meaning not true at all). As each rule is evaluated, the smallest (minimum) antecedent is taken to be the overall truth of the rule. This rule truth value is applied to each consequent of the rule (by storing this value to the corresponding fuzzy output) unless the fuzzy output is already larger (maximum). If two rules affect the same fuzzy output, the rule that is most true governs the value in the fuzzy output because the rules are connected by an implied fuzzy *or*.

In the case of rule weighting, the truth value for a rule is determined as usual by finding the smallest rule antecedent. Before applying this truth value to the consequents for the rule, the value is multiplied by a fraction from zero (rule disabled) to one (rule fully enabled). The resulting modified truth value is then applied to the fuzzy outputs.

The end result of the rule evaluation step is a table of suggested or “raw” fuzzy outputs in RAM. These values were obtained by plugging current conditions (fuzzy input values) into the system rules in the knowledge base. The raw results cannot be supplied directly to the system outputs because they may be ambiguous. For instance, one raw output can indicate that the system output should be medium with a degree of truth of 50 percent while, at the same time, another indicates that the system output should be low with a degree of truth of 25 percent. The defuzzification step resolves these ambiguities.

9.2.3 Defuzzification (WAV)

The final step in the fuzzy logic program combines the raw fuzzy outputs into a composite system output. Unlike the trapezoidal shapes used for inputs, the CPU12 typically uses singletons for output membership functions. As with the inputs, the x-axis represents the range of possible values for a system output. Singleton membership functions consist of the x-axis position for a label of the system output. Fuzzy outputs correspond to the y-axis height of the corresponding output membership function.

The WAV instruction calculates the numerator and denominator sums for weighted average of the fuzzy outputs according to the formula:

$$\text{System Output} = \frac{\sum_{i=1}^{n} S_i F_i}{\sum_{i=1}^{n} F_i}$$

Where n is the number of labels of a system output, S_i are the singleton positions from the knowledge base, and F_i are fuzzy outputs from RAM. For a common fuzzy logic program on the CPU12, n is eight or less (though this instruction can handle any value to 255) and S_i and F_i are 8-bit values. The final divide is performed with a separate EDIV instruction placed immediately after the WAV instruction.

Before executing WAV, an accumulator must be loaded with the number of iterations (n), one index register must be pointed at the list of singleton positions in the knowledge base, and a second index register must be pointed at the list of fuzzy outputs in RAM. If the system has more than one system output, the WAV instruction is executed once for each system output.

9.3 Example Inference Kernel

Figure 9-3 is a complete fuzzy inference kernel written in CPU12 assembly language. Numbers in square brackets are cycle counts for an HCS12 device. The kernel uses two system inputs with seven labels each and one system output with seven labels. The program assembles to 57 bytes. It executes in about 20 µs at an 25-MHz bus rate. The basic structure can easily be extended to a general-purpose system with a larger number of inputs and outputs.

```
   *
01 [2]    FUZZIFY      LDX   #INPUT_MFS      ;Point at MF definitions
02 [2]                 LDY   #FUZ_INS        ;Point at fuzzy input table
03 [3]                 LDAA  CURRENT_INS     ;Get first input value
04 [1]                 LDAB  #7              ;7 labels per input
05 [5]    GRAD_LOOP    MEM                   ;Evaluate one MF
06 [3]                 DBNE  B,GRAD_LOOP     ;For 7 labels of 1 input
07 [3]                 LDAA  CURRENT_INS+1   ;Get second input value
08 [1]                 LDAB  #7              ;7 labels per input
09 [5]    GRAD_LOOP1   MEM                   ;Evaluate one MF
10 [3]                 DBNE  B,GRAD_LOOP1    ;For 7 labels of 1 input

11 [1]                 LDAB  #7              ;Loop count
12 [2]    RULE_EVAL    CLR   1,Y+            ;Clr a fuzzy out & inc ptr
13 [3]                 DBNE  b,RULE_EVAL     ;Loop to clr all fuzzy outs
14 [2]                 LDX   #RULE_START     ;Point at first rule element
15 [2]                 LDY   #FUZ_INS        ;Point at fuzzy ins and outs
16 [1]                 LDAA  #$FF            ;Init A (and clears V-bit)
17 [3n+4]              REV                   ;Process rule list

18 [2]    DEFUZ        LDY   #FUZ_OUT        ;Point at fuzzy outputs
19 [2]                 LDX   #SGLTN_POS      ;Point at singleton positions
20 [1]                 LDAB  #7              ;7 fuzzy outs per COG output
21 [7b+4]              WAV                   ;Calculate sums for wtd av
22 [11]                EDIV                  ;Final divide for wtd av
23 [1]                 TFR   Y,D             ;Move result to A:B
24 [3]                 STAB  COG_OUT         ;Store system output
   *
   ***** End
```

Figure 9-3. Fuzzy Inference Engine

Lines 1 to 3 set up pointers and load the system input value into the A accumulator.

Line 4 sets the loop count for the loop in lines 5 and 6.

Lines 5 and 6 make up the fuzzification loop for seven labels of one system input. The MEM instruction finds the y-value on a trapezoidal membership function for the current input value, for one label of the current input, and then stores the result to the corresponding fuzzy input. Pointers in X and Y are automatically updated by four and one so they point at the next membership function and fuzzy input respectively.

Line 7 loads the current value of the next system input. Pointers in X and Y already point to the right places as a result of the automatic update function of the MEM instruction in line 5.

Line 8 reloads a loop count.

Lines 9 and 10 form a loop to fuzzify the seven labels of the second system input. When the program drops to line 11, the Y index register is pointing at the next location after the last fuzzy input, which is the first fuzzy output in this system.

Line 11 sets the loop count to clear seven fuzzy outputs.

Lines 12 and 13 form a loop to clear all fuzzy outputs before rule evaluation starts.

Line 14 initializes the X index register to point at the first element in the rule list for the REV instruction.

Line 15 initializes the Y index register to point at the fuzzy inputs and outputs in the system. The rule list (for REV) consists of 8-bit offsets from this base address to particular fuzzy inputs or fuzzy outputs. The special value $FE is interpreted by REV as a marker between rule antecedents and consequents.

Line 16 initializes the A accumulator to the highest 8-bit value in preparation for finding the smallest fuzzy input referenced by a rule antecedent. The LDAA #$FF instruction also clears the V-bit in the CPU12’s condition code register so the REV instruction knows it is processing antecedents. During rule list processing, the V bit is toggled each time an $FE is detected in the list. The V bit indicates whether REV is processing antecedents or consequents.

Line 17 is the REV instruction, a self-contained loop to process successive elements in the rule list until an $FF character is found. For a system of 17 rules with two antecedents and one consequent each, the REV instruction takes 259 cycles, but it is interruptible so it does not cause a long interrupt latency.

Lines 18 through 20 set up pointers and an iteration count for the WAV instruction.

Line 21 is the beginning of defuzzification. The WAV instruction calculates a sum-of-products and a sum-of-weights.

Line 22 completes defuzzification. The EDIV instruction performs a 32-bit by 16-bit divide on the intermediate results from WAV to get the weighted average.

Line 23 moves the EDIV result into the double accumulator.

Line 24 stores the low 8-bits of the defuzzification result.

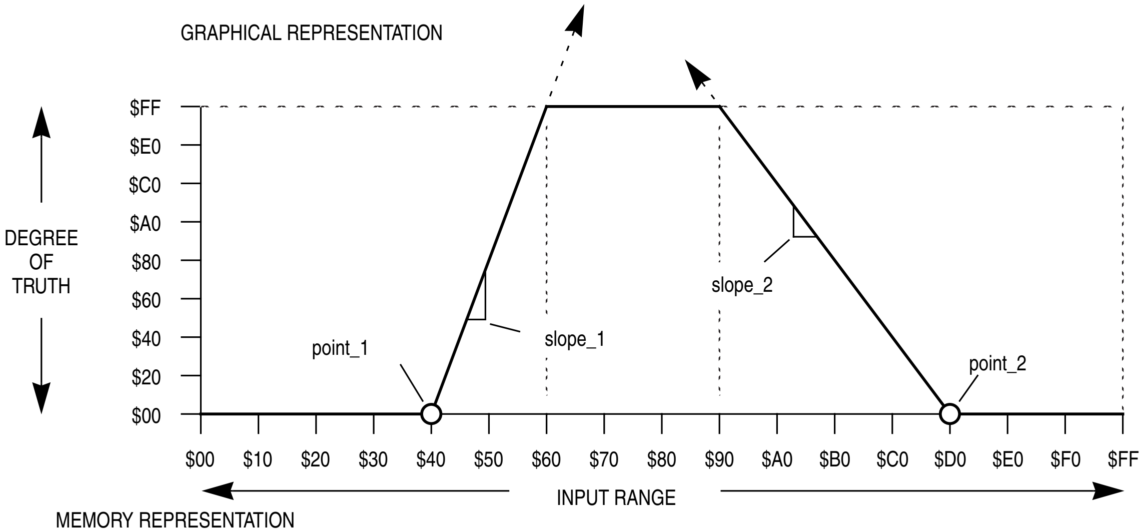
This example inference program shows how easy it is to incorporate fuzzy logic into general applications using the CPU12. Code space and execution time are no longer serious factors in the decision to use fuzzy logic. The next section begins a much more detailed look at the fuzzy logic instructions of the CPU12.

9.4 MEM Instruction Details

This section provides a more detailed explanation of the membership function evaluation instruction (MEM), including details about abnormal special cases for improperly defined membership functions.

9.4.1 Membership Function Definitions

Figure 9-4 shows how a normal membership function is specified in the CPU12. Typically, a software tool is used to input membership functions graphically, and the tool generates data structures for the target processor and software kernel. Alternatively, points and slopes for the membership functions can be determined and stored in memory with define-constant assembler directives.



MEMORY REPRESENTATION

ADDR	$40	X-POSITION OF point_1
ADDR+1	$D0	X-POSITION OF point_2
ADDR+2	$08	slope_1 ($FF/(X-POS OF SATURATION – point_1))
ADDR+3	$04	slope_2 ($FF/(point_2 – X-POS OF SATURATION))

Figure 9-4. Defining a Normal Membership Function

An internal CPU algorithm calculates the y-value where the current input intersects a membership function. This algorithm assumes the membership function obeys some common-sense rules. If the membership function definition is improper, the results may be unusual. See **9.4.2 Abnormal Membership Function Definitions** for a discussion of these cases.

These rules apply to normal membership functions.

- $00 ≤ point1 < $FF
- $00 < point2 ≤ $FF
- point1 < point2
- The sloping sides of the trapezoid meet at or above $FF.

Each system input such as temperature has several labels such as cold, cool, normal, warm, and hot. Each label of each system input must have a membership function to describe its meaning in an unambiguous numerical way. Typically, there are three to seven labels per system input, but there is no practical restriction on this number as far as the fuzzification step is concerned.

9.4.2 Abnormal Membership Function Definitions

In the CPU12, it is possible (and proper) to define “crisp” membership functions. A crisp membership function has one or both sides vertical (infinite slope). Since the slope value $00 is not used otherwise, it is assigned to mean infinite slope to the MEM instruction in the CPU12.

Although a good fuzzy development tool will not allow the user to specify an improper membership function, it is possible to have program errors or memory errors which result in erroneous abnormal membership functions. Although these abnormal shapes do not correspond to any working systems, understanding how the CPU12 treats these cases can be helpful for debugging.

A close examination of the MEM instruction algorithm will show how such membership functions are evaluated. **Figure 9-5** is a complete flow diagram for the execution of a MEM instruction. Each rectangular box represents one CPU bus cycle. The number in the upper left corner corresponds to the cycle number and the letter corresponds to the cycle type (refer to **Section 6. Instruction Glossary** for details). The upper portion of the box includes information about bus activity during this cycle (if any). The lower portion of the box, which is separated by a dashed line, includes information about internal CPU processes. It is common for several internal functions to take place during a single CPU cycle (for example, in cycle 2, two 8-bit subtractions take place and a flag is set based on the results).

Consider 4a: If (((Slope_2 = 0) or (Grade_2 > $FF)) and (flag_d12n = 0)).

The flag_d12n is zero as long as the input value (in accumulator A) is within the trapezoid. Everywhere outside the trapezoid, one or the other delta term will be negative, and the flag will equal one. Slope_2 equals zero indicates the right side of the trapezoid has infinite slope, so the resulting grade should be $FF everywhere in the trapezoid, including at point_2, as far as this side is concerned. The term grade_2 greater than $FF means the value is far enough into the trapezoid that the right sloping side of the trapezoid has crossed above the $FF cutoff level and the resulting grade should be $FF as far as the right sloping side is concerned. 4a decides if the value is left of the right sloping side (Grade = $FF), or on the sloping portion of the right side of the trapezoid (Grade = Grade_2). 4b could still override this tentative value in grade.

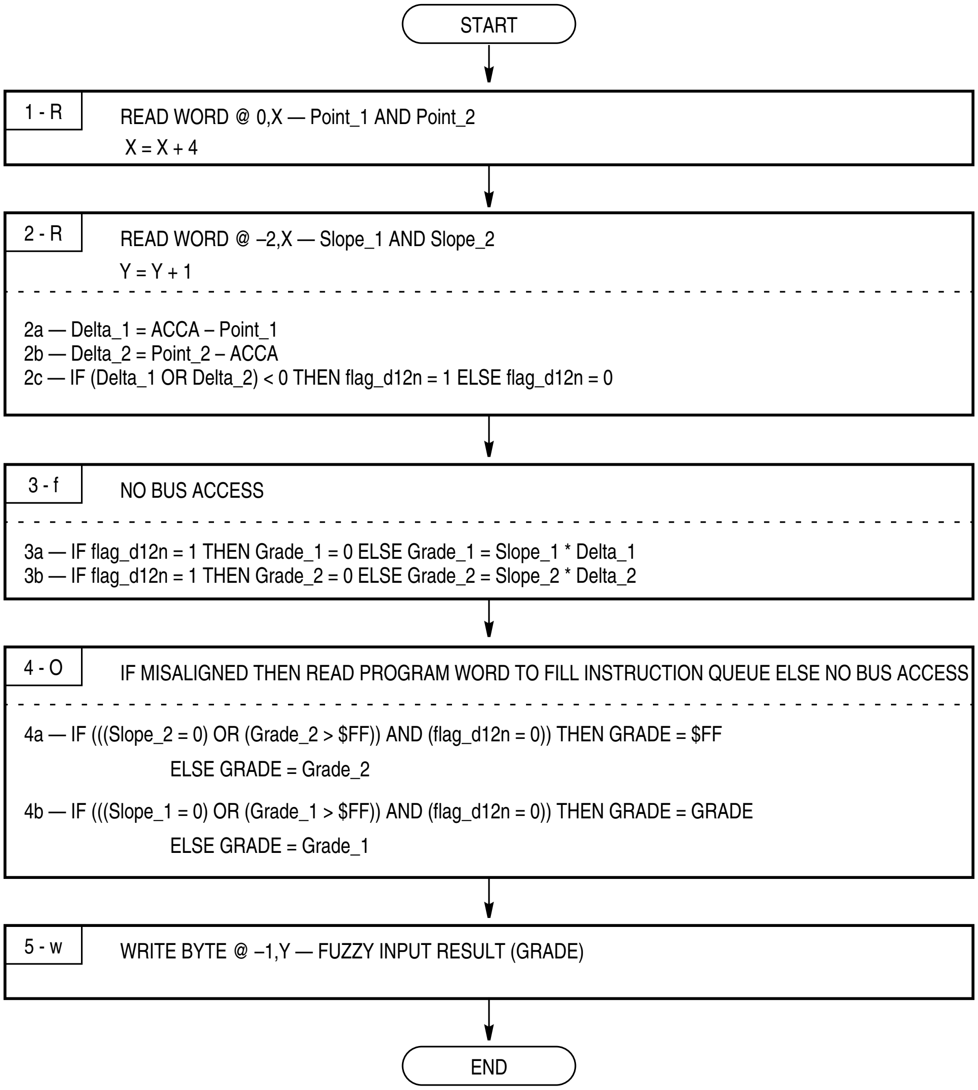

START
1 - R
READ WORD @ 0,X — Point_1 AND Point_2
X = X + 4
2 - R
READ WORD @ –2,X — Slope_1 AND Slope_2
Y = Y + 1
2a — Delta_1 = ACCA – Point_1
2b — Delta_2 = Point_2 – ACCA
2c — IF (Delta_1 OR Delta_2) < 0 THEN flag_d12n = 1 ELSE flag_d12n = 0
3 - f
NO BUS ACCESS
3a — IF flag_d12n = 1 THEN Grade_1 = 0 ELSE Grade_1 = Slope_1 * Delta_1
3b — IF flag_d12n = 1 THEN Grade_2 = 0 ELSE Grade_2 = Slope_2 * Delta_2
4 - O
IF MISALIGNED THEN READ PROGRAM WORD TO FILL INSTRUCTION QUEUE ELSE NO BUS ACCESS
4a — IF (((Slope_2 = 0) OR (Grade_2 > $FF)) AND (flag_d12n = 0)) THEN GRADE = $FF
ELSE GRADE = Grade_2
4b — IF (((Slope_1 = 0) OR (Grade_1 > $FF)) AND (flag_d12n = 0)) THEN GRADE = GRADE
ELSE GRADE = Grade_1
5 - w
WRITE BYTE @ –1,Y — FUZZY INPUT RESULT (GRADE)
END

Figure 9-5. MEM Instruction Flow Diagram

In 4b, slope_1 is zero if the left side of the trapezoid has infinite slope (vertical). If so, the result (grade) should be $FF at and to the right of point_1 everywhere within the trapezoid as far as the left side is concerned. The grade_1 greater than $FF term corresponds to the input being to the right of where the left sloping side passes the $FF cutoff level. If either of these conditions is true, the result (grade) is left at the value it got from 4a. The “else” condition in 4b corresponds to the input falling on the sloping portion of the left side of the trapezoid (or possibly outside the trapezoid), so the result is grade equal grade_1. If the input was outside the trapezoid, flag_d12n would be one and grade_1 and grade_2 would have been forced to $00 in cycle 3. The else condition of 4b would set the result to $00.

The special cases shown here represent abnormal membership function definitions. The explanations describe how the specific algorithm in the CPU12 resolves these unusual cases. The results are not all intuitively obvious, but rather fall out from the specific algorithm. Remember, these cases should not occur in a normal system.

9.4.2.1 Abnormal Membership Function Case 1

This membership function is abnormal because the sloping sides cross below the $FF cutoff level. The flag_d12n signal forces the membership function to evaluate to $00 everywhere except from point_1 to point_2. Within this interval, the tentative values for grade_1 and grade_2 calculated in cycle 3 fall on the crossed sloping sides. In step 4a, grade gets set to the grade_2 value, but in 4b this is overridden by the grade_1 value, which ends up as the result of the MEM instruction. One way to say this is that the result follows the left sloping side until the input passes point_2, where the result goes to $00.

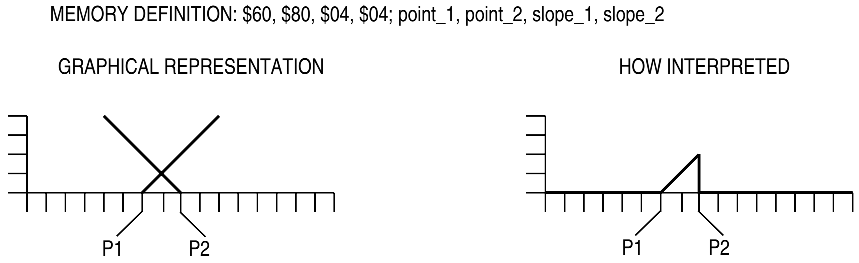

MEMORY DEFINITION: $60, $80, $04, $04; point_1, point_2, slope_1, slope_2
GRAPHICAL REPRESENTATION
HOW INTERPRETED
P1
P2
P1
P2

Figure 9-6. Abnormal Membership Function Case 1

If point_1 was to the right of point_2, flag_d12n would force the result to be $00 for all input values. In fact, flag_d12n always limits the region of interest to the space greater than or equal to point_1 and less than or equal to point_2.

9.4.2.2 Abnormal Membership Function Case 2

Like the previous example, the membership function in case 2 is abnormal because the sloping sides cross below the $FF cutoff level, but the left sloping side reaches the $FF cutoff level before the input gets to point_2. In this case, the result follows the left sloping side until it reaches the $FF cutoff level. At this point, the (grade_1 > $FF) term of 4b kicks in, making the expression true so grade equals grade (no overwrite). The result from here to point_2 becomes controlled by the “else” part of 4a (grade = grade_2), and the result follows the right sloping side.

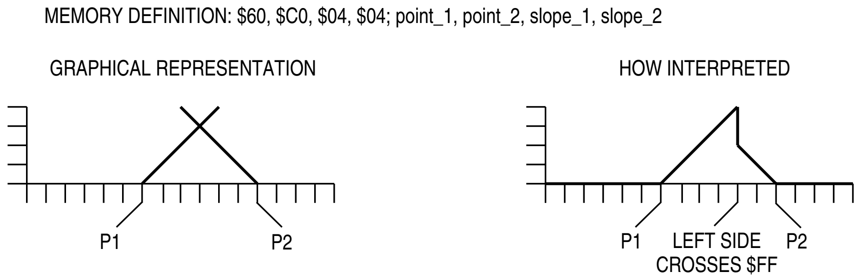

MEMORY DEFINITION: $60, $C0, $04, $04; point_1, point_2, slope_1, slope_2
GRAPHICAL REPRESENTATION
HOW INTERPRETED
P1
P2
P1
LEFT SIDE CROSSES $FF
P2

Figure 9-7. Abnormal Membership Function Case 2

9.4.2.3 Abnormal Membership Function Case 3

The membership function in case 3 is abnormal because the sloping sides cross below the $FF cutoff level, and the left sloping side has infinite slope. In this case, 4a is not true, so grade equals grade_2. 4b is true because slope_1 is zero, so 4b does not overwrite grade.

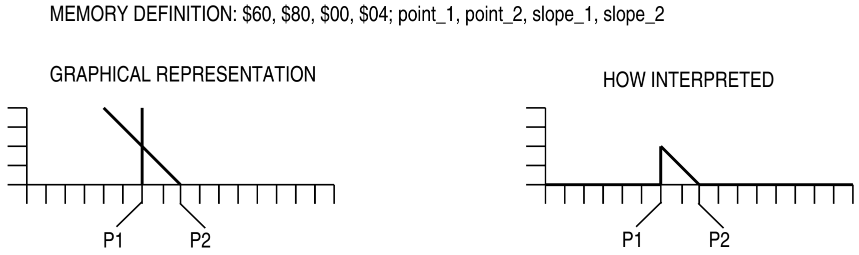

MEMORY DEFINITION: $60, $80, $00, $04; point_1, point_2, slope_1, slope_2
GRAPHICAL REPRESENTATION
HOW INTERPRETED
P1
P2
P1
P2

Figure 9-8. Abnormal Membership Function Case 3

9.5 REV and REVW Instruction Details

This section provides a more detailed explanation of the rule evaluation instructions (REV and REVW). The data structures used to specify rules are somewhat different for the weighted versus unweighted versions of the instruction. One uses 8-bit offsets in the encoded rules, while the other uses full 16-bit addresses. This affects the size of the rule data structure and execution time.

9.5.1 Unweighted Rule Evaluation (REV)

This instruction implements basic min-max rule evaluation. CPU registers are used for pointers and intermediate calculation results.

Since the REV instruction is essentially a list-processing instruction, execution time is dependent on the number of elements in the rule list. The REV instruction is interruptible (typically within three bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and REVW instructions.

9.5.1.1 Set Up Prior to Executing REV

Some CPU registers and memory locations need to be set up prior to executing the REV instruction. X and Y index registers are used as index pointers to the rule list and the fuzzy inputs and outputs. The A accumulator is used for intermediate calculation results and needs to be set to $FF initially. The V condition code bit is used as an instruction status indicator to show whether antecedents or consequents are being processed. Initially, the V bit is cleared to zero to indicate antecedents are being processed. The fuzzy outputs (working RAM locations) need to be cleared to $00. If these values are not initialized before executing the REV instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The REV instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the REV instruction finishes, X will point at the next address past the $FF separator character that marks the end of the rule list.

The Y index register is set to the base address for the fuzzy inputs and outputs (in working RAM). Each rule antecedent is an unsigned 8-bit offset from this base address to the referenced fuzzy input. Each rule consequent is an unsigned 8-bit offset from this base address to the referenced fuzzy output. The Y index register remains constant throughout execution of the REV instruction.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REV instruction. During antecedent processing, A starts out at $FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent (MIN). During consequent processing, A holds the truth value for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REV, A must be set to $FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to $FF when the instruction detects the $FE marker character between the last consequent of the previous rule and the first antecedent of a new rule.

The instruction LDAA #$FF clears the V bit at the same time it initializes A to $FF. This satisfies the REV setup requirement to clear the V bit as well as the requirement to initialize A to $FF. Once the REV instruction starts, the value in the V bit is automatically maintained as $FE separator characters are detected.

The final requirement to clear all fuzzy outputs to $00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value for the current rule. If the current truth value is larger, it is written over the previous value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REV finishes, A will hold the truth value for the last rule in the rule list. The V condition code bit should be one because the last element before the $FF end marker should have been a rule consequent. If V is zero after executing REV, it indicates the rule list was structured incorrectly.

9.5.1.2 Interrupt Details

The REV instruction includes a 3-cycle processing loop for each byte in the rule list (including antecedents, consequents, and special separator characters). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REV instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REV instruction, points to the REV instruction rather than the instruction that follows. This causes the CPU to try to execute a new REV instruction upon return from the interrupt. Since the CPU registers (including the V bit in the condition codes register) indicate the current status of the interrupted REV instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.5.1.3 Cycle-by-Cycle Details for REV

The central element of the REV instruction is a 3-cycle loop that is executed once for each byte in the rule list. There is a small amount of housekeeping activity to get this loop started as REV begins and a small sequence to end the instruction. If an interrupt comes, there is a special small sequence to save CPU status on the stack before honoring the requested interrupt.

Figure 9-9 is a REV instruction flow diagram. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to execution cycle codes (refer to **Section 6. Instruction Glossary** for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit or no data is transferred.

When a value is read from memory, it cannot be used by the CPU until the second cycle after the read takes place. This is due to access and propagation delays.

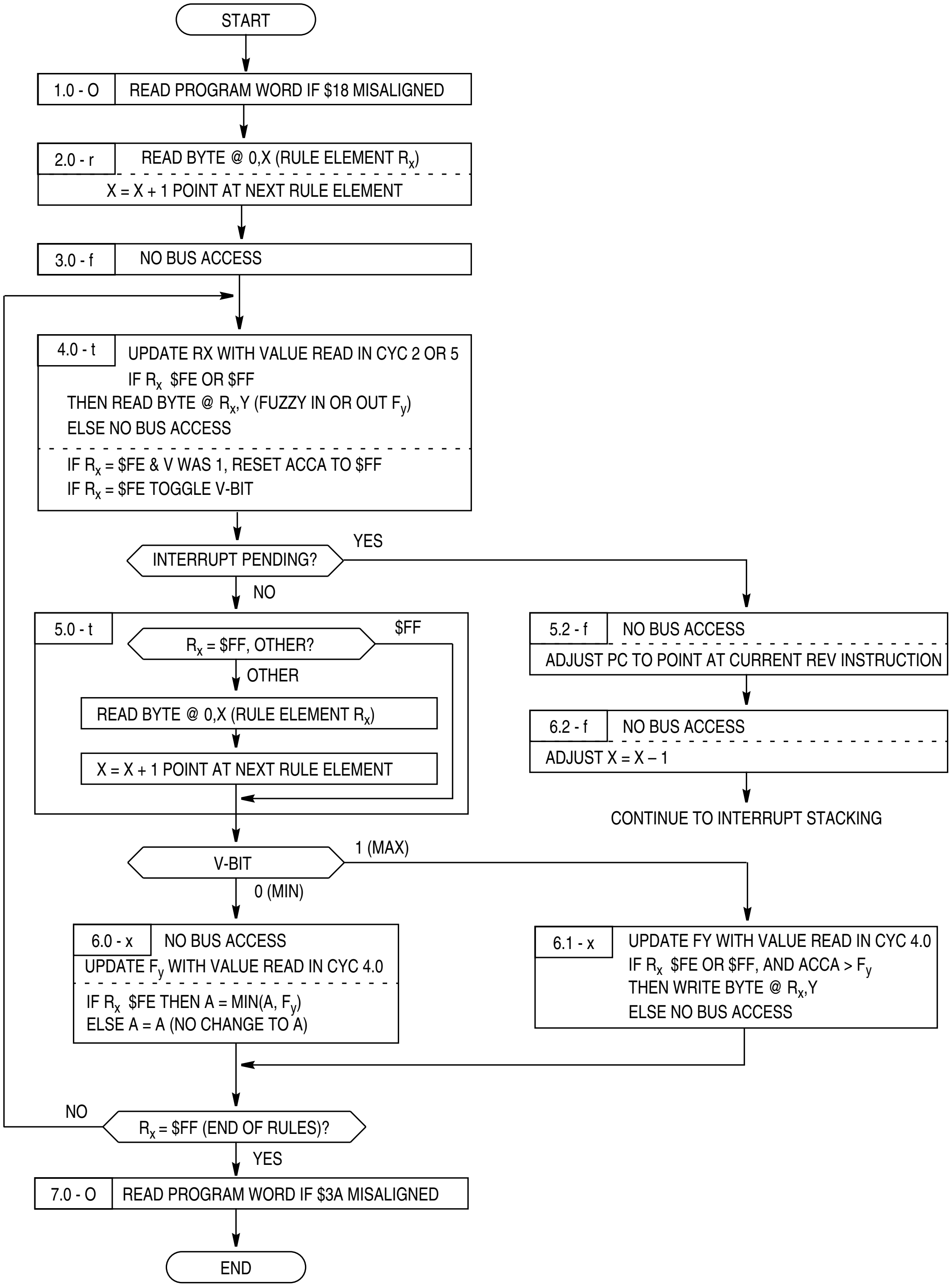

START
1.0 - O
READ PROGRAM WORD IF $18 MISALIGNED
2.0 - r
READ BYTE @ 0,X (RULE ELEMENT R_x)
X = X + 1 POINT AT NEXT RULE ELEMENT
3.0 - f
NO BUS ACCESS
4.0 - t
UPDATE RX WITH VALUE READ IN CYC 2 OR 5
IF R_x $FE OR $FF
THEN READ BYTE @ R_x,Y (FUZZY IN OR OUT F_y)
ELSE NO BUS ACCESS
IF R_x = $FE & V WAS 1, RESET ACCA TO $FF
IF R_x = $FE TOGGLE V-BIT
INTERRUPT PENDING?
YES
NO
5.0 - t
R_x = $FF, OTHER?
$FF
OTHER
READ BYTE @ 0,X (RULE ELEMENT R_x)
X = X + 1 POINT AT NEXT RULE ELEMENT
5.2 - f
NO BUS ACCESS
ADJUST PC TO POINT AT CURRENT REV INSTRUCTION
6.2 - f
NO BUS ACCESS
ADJUST X = X – 1
CONTINUE TO INTERRUPT STACKING
V-BIT
1 (MAX)
0 (MIN)
6.0 - x
NO BUS ACCESS
UPDATE F_y WITH VALUE READ IN CYC 4.0
IF R_x $FE THEN A = MIN(A, F_y)
ELSE A = A (NO CHANGE TO A)
6.1 - x
UPDATE FY WITH VALUE READ IN CYC 4.0
IF R_x $FE OR $FF, AND ACCA > F_y
THEN WRITE BYTE @ R_x,Y
ELSE NO BUS ACCESS
NO
R_x = $FF (END OF RULES)?
YES
7.0 - O
READ PROGRAM WORD IF $3A MISALIGNED
END

Figure 9-9. REV Instruction Flow Diagram

Since there is more than one flow path through the REV instruction, cycle numbers have a decimal place. This decimal place indicates which of several possible paths is being used. The CPU normally moves forward by one digit at a time within the same flow (flow number is indicated after the decimal point in the cycle number). There are two exceptions possible to this orderly sequence through an instruction. The first is a branch back to an earlier cycle number to form a loop as in 6.0 to 4.0. The second type of sequence change is from one flow to a parallel flow within the same instruction such as 4.0 to 5.2, which occurs if the REV instruction senses an interrupt. In this second type of sequence branch, the whole number advances by one and the flow number changes to a new value (the digit after the decimal point).

In cycle 1.0, the CPU12 does an optional program word access to replace the $18 prebyte of the REV instruction. Notice that cycle 7.0 is also an O type cycle. One or the other of these will be a program word fetch, while the other will be a free cycle where the CPU does not access the bus. Although the $18 page prebyte is a required part of the REV instruction, it is treated by the CPU12 as a somewhat separate single cycle instruction.

Rule evaluation begins at cycle 2.0 with a byte read of the first element in the rule list. Usually this would be the first antecedent of the first rule, but the REV instruction can be interrupted, so this could be a read of any byte in the rule list. The X index register is incremented so it points to the next element in the rule list. Cycle 3.0 is needed to satisfy the required delay between a read and when data is valid to the CPU. Some internal CPU housekeeping activity takes place during this cycle, but there is no bus activity. By cycle 4.0, the rule element that was read in cycle 2.0 is available to the CPU.

Cycle 4.0 is the first cycle of the main three cycle rule evaluation loop. Depending upon whether rule antecedents or consequents are being processed, the loop will consist of cycles 4.0, 5.0, 6.0, or the sequence 4.0, 5.0, 6.1. This loop is executed once for every byte in the rule list, including the $FE separators and the $FF end-of-rules marker.

At each cycle 4.0, a fuzzy input or fuzzy output is read, except during the loop passes associated with the $FE and $FF marker bytes, where no bus access takes place during cycle 4.0. The read access uses the Y index register as the base address and the previously read rule byte (R_X) as an unsigned offset from Y. The fuzzy input or output value read here will be used during the next cycle 6.0 or 6.1. Besides being used as the offset from Y for this read, the previously read R_X is checked to see if it is a separator character ($FE). If R_X was $FE and the V bit was one, this indicates a switch from processing consequents of one rule to starting to process antecedents of the next rule. At this transition, the A accumulator is initialized to $FF to prepare for the min operation to find the smallest fuzzy input. Also, if Rx is $FE, the V bit is toggled to indicate the change from antecedents to consequents, or consequents to antecedents.

During cycle 5.0, a new rule byte is read unless this is the last loop pass, and R_x is $FF (marking the end of the rule list). This new rule byte will not be used until cycle 4.0 of the next pass through the loop.

Between cycle 5.0 and 6.x, the V-bit is used to decide which of two paths to take. If V is zero, antecedents are being processed and the CPU progresses to cycle 6.0. If V is one, consequents are being processed and the CPU goes to cycle 6.1.

During cycle 6.0, the current value in the A accumulator is compared to the fuzzy input that was read in the previous cycle 4.0, and the lower value is placed in the A accumulator (min operation). If Rx is $FE, this is the transition between rule antecedents and rule consequents, and this min operation is skipped (although the cycle is still used). No bus access takes place during cycle 6.0 but cycle 6.x is considered an x type cycle because it could be a byte write (cycle 6.1) or a free cycle (cycle 6.0 or 6.1 with Rx = $FE or $FF).

If an interrupt arrives while the REV instruction is executing, REV can break between cycles 4.0 and 5.0 in an orderly fashion so that the rule evaluation operation can resume after the interrupt has been serviced. Cycles 5.2 and 6.2 are needed to adjust the PC and X index register so the REV operation can recover after the interrupt. PC is adjusted backward in cycle 5.2 so it points to the currently running REV instruction. After the interrupt, rule evaluation will resume, but the values that were stored on the stack for index registers, accumulator A, and CCR will cause the operation to pick up where it left off. In cycle 6.2, the X index register is adjusted backward by one because the last rule byte needs to be re-fetched when the REV instruction resumes.

After cycle 6.2, the REV instruction is finished, and execution would continue to the normal interrupt processing flow.

9.5.2 Weighted Rule Evaluation (REVW)

This instruction implements a weighted variation of min-max rule evaluation. The weighting factors are stored in a table with one 8-bit entry per rule. The weight is used to multiply the truth value of the rule (minimum of all antecedents) by a value from zero to one to get the weighted result. This weighted result is then applied to the consequents, just as it would be for unweighted rule evaluation.

Since the REVW instruction is essentially a list-processing instruction, execution time is dependent on the number of rules and the number of elements in the rule list. The REVW instruction is interruptible (typically within three to five bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and REVW instructions.

The rule structure is different for REVW than for REV. For REVW, the rule list is made up of 16-bit elements rather than 8-bit elements. Each antecedent is represented by the full 16-bit address of the corresponding fuzzy input. Each rule consequent is represented by the full address of the corresponding fuzzy output.

The markers separating antecedents from consequents are the reserved 16-bit value $FFFE, and the end of the last rule is marked by the reserved 16-bit value $FFFF. Since $FFFE and $FFFF correspond to the addresses of the reset vector, there would never be a fuzzy input or output at either of these locations.

9.5.2.1 Set Up Prior to Executing REVW

Some CPU registers and memory locations need to be set up prior to executing the REVW instruction. X and Y index registers are used as index pointers to the rule list and the list of rule weights. The A accumulator is used for intermediate calculation results and needs to be set to $FF initially. The V condition code bit is used as an instruction status indicator that shows whether antecedents or consequents are being processed. Initially the V bit is cleared to zero to indicate antecedents are being processed. The C condition code bit is used to indicate whether rule weights are to be used (1) or not (0). The fuzzy outputs (working RAM locations) need to be cleared to $00. If these values are not initialized before executing the REVW instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The REVW instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the REVW instruction finishes, X will point at the next address past the $FFFF separator word that marks the end of the rule list.

The Y index register is set to the starting address of the list of rule weights. Each rule weight is an 8-bit value. The weighted result is the truncated upper eight bits of the 16-bit result, which is derived by multiplying the minimum rule antecedent value ($00–$FF) by the weight plus one ($001–$100). This method of weighting rules allows an 8-bit weighting factor to represent a value between zero and one inclusive.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REVW instruction. During antecedent processing, A starts out at $FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent. If rule weights are enabled by the C condition code bit equal one, the rule truth value is multiplied by the rule weight just before consequent processing starts. During consequent processing, A holds the truth value (possibly weighted) for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REVW, A must be set to $FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to $FF when the instruction detects the $FFFE marker word between the last consequent of the previous rule, and the first antecedent of a new rule.

Both the C and V condition code bits must be set up prior to starting a REVW instruction. Once the REVW instruction starts, the C bit remains constant and the value in the V bit is automatically maintained as $FFFE separator words are detected.

The final requirement to clear all fuzzy outputs to $00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value (weighted) for the current rule. If the current truth value is larger, it is written over the previous value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REVW finishes, A will hold the truth value (weighted) for the last rule in the rule list. The V condition code bit should be one because the last element before the $FFFF end marker should have been a rule consequent. If V is zero after executing REVW, it indicates the rule list was structured incorrectly.

9.5.2.2 Interrupt Details

The REVW instruction includes a 3-cycle processing loop for each word in the rule list (this loop expands to five cycles between antecedents and consequents to allow time for the multiplication with the rule weight). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REVW instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REVW instruction, points to the REVW instruction rather than the instruction that follows. This causes the CPU to try to execute a new REVW instruction upon return from the interrupt. Since the CPU registers (including the C bit and V bit in the condition codes register) indicate the current status of the interrupted REVW instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.5.2.3 Cycle-by-Cycle Details for REVW

The central element of the REVW instruction is a 3-cycle loop that is executed once for each word in the rule list. For the special case pass (where the $FFFE separator word is read between the rule antecedents and the rule consequents, and weights are enabled by the C bit equal one), this loop takes five cycles. There is a small amount of housekeeping activity to get this loop started as REVW begins and a small sequence to end the instruction. If an interrupt comes, there is a special small sequence to save CPU status on the stack before the interrupt is serviced.

Figure 9-10 is a detailed flow diagram for the REVW instruction. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to the execution cycle codes (refer to **Section 6. Instruction Glossary** for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

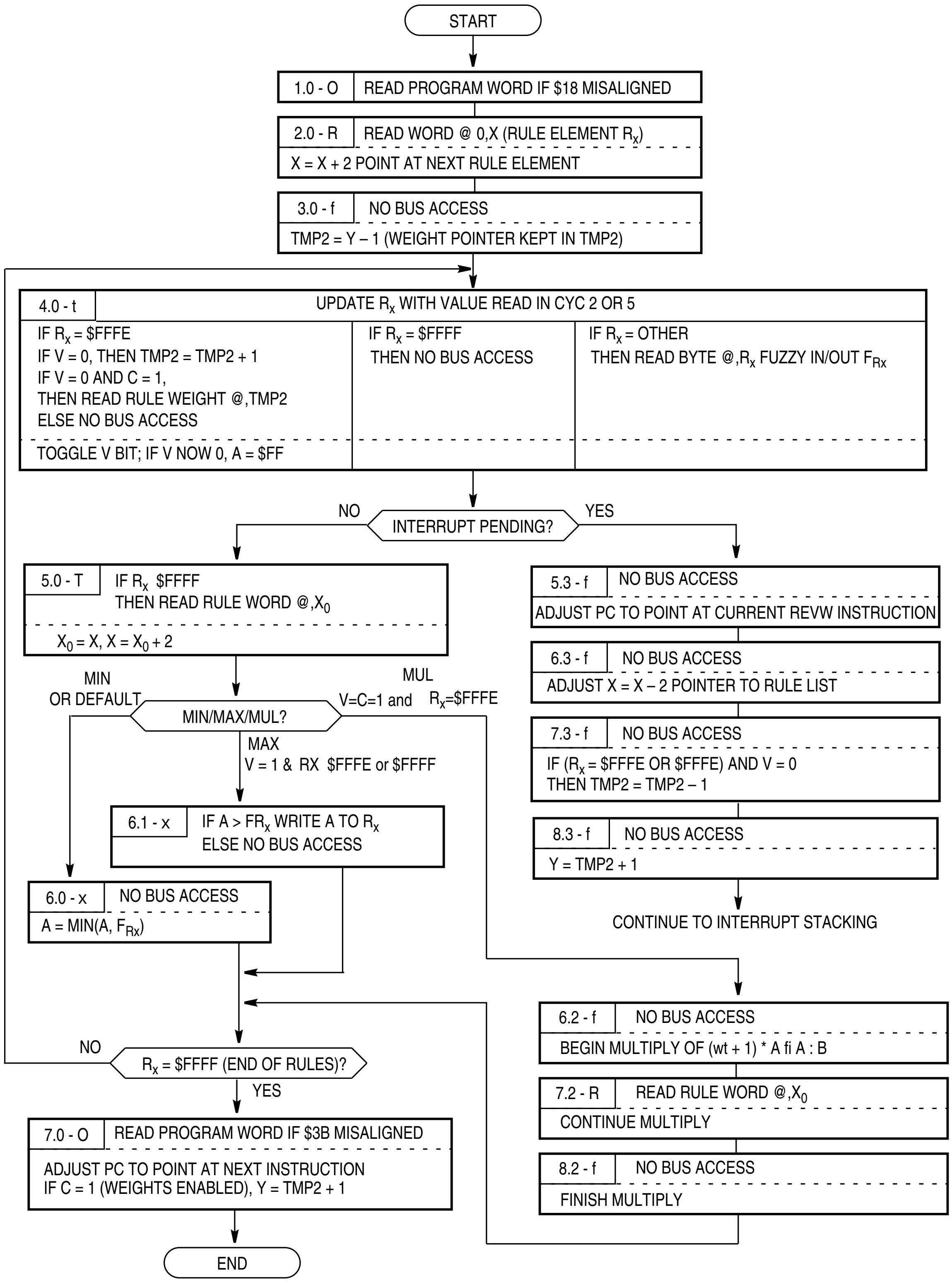

START
1.0 - O
READ PROGRAM WORD IF $18 MISALIGNED
2.0 - R
READ WORD @ 0,X (RULE ELEMENT R_x)
X = X + 2 POINT AT NEXT RULE ELEMENT
3.0 - f
NO BUS ACCESS
TMP2 = Y – 1 (WEIGHT POINTER KEPT IN TMP2)
4.0 - t
UPDATE R_x WITH VALUE READ IN CYC 2 OR 5
IF R_x = $FFFE
IF V = 0, THEN TMP2 = TMP2 + 1
IF V = 0 AND C = 1,
THEN READ RULE WEIGHT @,TMP2
ELSE NO BUS ACCESS
TOGGLE V BIT; IF V NOW 0, A = $FF
IF R_x = $FFFF
THEN NO BUS ACCESS
IF R_x = OTHER
THEN READ BYTE @,R_x FUZZY IN/OUT F_{Rx}
NO
INTERRUPT PENDING?
YES
5.0 - T
IF R_x $FFFF
THEN READ RULE WORD @,X_0
X_0 = X, X = X_0 + 2
MIN
OR DEFAULT
MIN/MAX/MUL?
MUL
V=C=1 and R_x=$FFFE
MAX
V = 1 & RX $FFFE or $FFFF
6.1 - x
IF A > FR_x WRITE A TO R_x
ELSE NO BUS ACCESS
6.0 - x
NO BUS ACCESS
A = MIN(A, F_{Rx})
NO
R_x = $FFFF (END OF RULES)?
YES
7.0 - O
READ PROGRAM WORD IF $3B MISALIGNED
ADJUST PC TO POINT AT NEXT INSTRUCTION
IF C = 1 (WEIGHTS ENABLED), Y = TMP2 + 1
END
5.3 - f
NO BUS ACCESS
ADJUST PC TO POINT AT CURRENT REVW INSTRUCTION
6.3 - f
NO BUS ACCESS
ADJUST X = X – 2 POINTER TO RULE LIST
7.3 - f
NO BUS ACCESS
IF (R_x = $FFFE OR $FFFE) AND V = 0
THEN TMP2 = TMP2 – 1
8.3 - f
NO BUS ACCESS
Y = TMP2 + 1
CONTINUE TO INTERRUPT STACKING
6.2 - f
NO BUS ACCESS
BEGIN MULTIPLY OF (wt + 1) * A fi A : B
7.2 - R
READ RULE WORD @,X_0
CONTINUE MULTIPLY
8.2 - f
NO BUS ACCESS
FINISH MULTIPLY

Figure 9-10. REVW Instruction Flow Diagram

In cycle 2.0, the first element of the rule list (a 16-bit address) is read from memory. Due to propagation delays, this value cannot be used for calculations until two cycles later (cycle 4.0). The X index register, which is used to access information from the rule list, is incremented by two to point at the next element of the rule list.

The operations performed in cycle 4.0 depend on the value of the word read from the rule list. $FFFE is a special token that indicates a transition from antecedents to consequents or from consequents to antecedents of a new rule. The V bit can be used to decide which transition is taking place, and V is toggled each time the $FFFE token is detected. If V was zero, a change from antecedents to consequents is taking place, and it is time to apply weighting (provided it is enabled by the C bit equal one). The address in TMP2 (derived from Y) is used to read the weight byte from memory. In this case, there is no bus access in cycle 5.0, but the index into the rule list is updated to point to the next rule element.

The old value of X (X_0) is temporarily held on internal nodes, so it can be used to access a rule word in cycle 7.2. The read of the rule word is timed to start two cycles before it will be used in cycle 4.0 of the next loop pass. The actual multiply takes place in cycles 6.2 through 8.2. The 8-bit weight from memory is incremented (possibly overflowing to $100) before the multiply, and the upper eight bits of the 16-bit internal result is used as the weighted result. By using weight+1, the result can range from 0.0 times A to 1.0 times A. After 8.2, flow continues to the next loop pass at cycle 4.0.

At cycle 4.0, if R_x is $FFFE and V was one, a change from consequents to antecedents of a new rule is taking place, so accumulator A must be reinitialized to $FF. During processing of rule antecedents, A is updated with the smaller of A, or the current fuzzy input (cycle 6.0). Cycle 5.0 is usually used to read the next rule word and update the pointer in X. This read is skipped if the current R_x is $FFFF (end of rules mark). If this is a weight multiply pass, the read is delayed until cycle 7.2. During processing of consequents, cycle 6.1 is used to optionally update a fuzzy output if the value in accumulator A is larger.

After all rules have been processed, cycle 7.0 is used to update the PC to point at the next instruction. If weights were enabled, Y is updated to point at the location that immediately follows the last rule weight.

9.6 WAV Instruction Details

The WAV instruction performs weighted average calculations used in defuzzification. The pseudo-instruction wavr is used to resume an interrupted weighted average operation. WAV calculates the numerator and denominator sums using:

$$\text{System Output} = \frac{\sum_{i=1}^{n} S_i F_i}{\sum_{i=1}^{n} F_i}$$

Where n is the number of labels of a system output, S_i are the singleton positions from the knowledge base, and F_i are fuzzy outputs from RAM. S_i and F_i are 8-bit values. The 8-bit B accumulator holds the iteration count n. Internal temporary registers hold intermediate sums, 24 bits for the numerator and 16 bits for the denominator. This makes this instruction suitable for n values up to 255 although eight is a more typical value. The final long division is performed with a separate EDIV instruction immediately after the WAV instruction. The WAV instruction returns the numerator and denominator sums in the correct registers for the EDIV. (EDIV performs the unsigned division Y = Y : D / X; remainder in D.)

Execution time for this instruction depends on the number of iterations (labels for the system output). WAV is interruptible so that worst case interrupt latency is not affected by the execution time for the complete weighted average operation. WAV includes initialization for the 24-bit and 16-bit partial sums so the first entry into WAV looks different than a resume from interrupt operation. The CPU12 handles this difficulty with a pseudo-instruction (wavr), which is specifically intended to resume an interrupted weighted average calculation. Refer to **9.6.3 Cycle-by-Cycle Details for WAV and wavr** for more detail.

9.6.1 Set Up Prior to Executing WAV

Before executing the WAV instruction, index registers X and Y and accumulator B must be set up. Index register X is a pointer to the S_i singleton list. X must have the address of the first singleton value in the knowledge base. Index register Y is a pointer to the fuzzy outputs F_i. Y must have the address of the first fuzzy output for this system output. B is the iteration count n. The B accumulator must be set to the number of labels for this system output.

9.6.2 WAV Interrupt Details

The WAV instruction includes a 7-cycle processing loop for each label of the system output (8 cycles in M68HC12). Within this loop, the CPU checks whether a qualified interrupt request is pending. If an interrupt is detected, the current values of the internal temporary registers for the 24-bit and 16-bit sums are stacked, the CPU registers are stacked, and the interrupt is serviced.

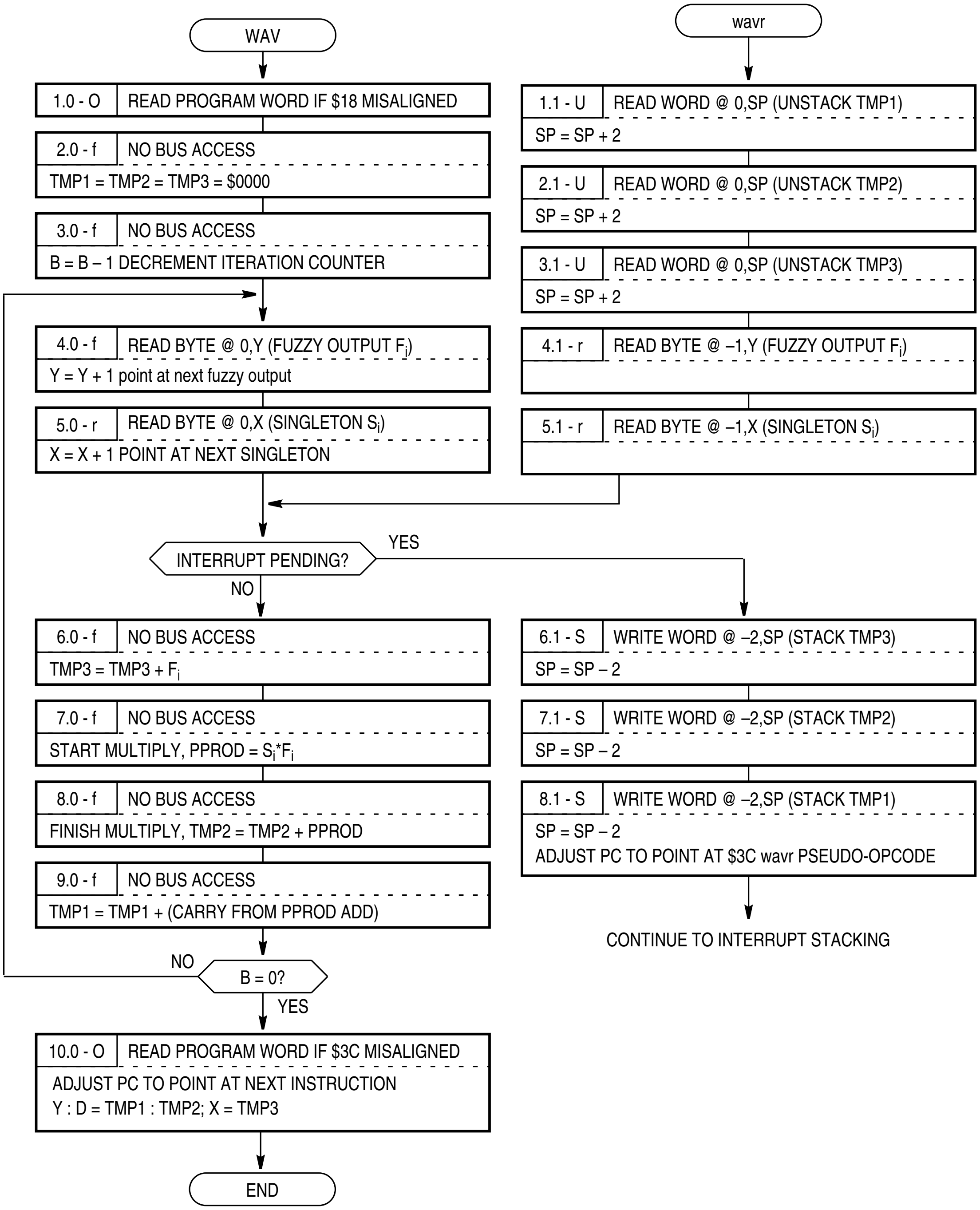
A special processing sequence is executed when an interrupt is detected during a weighted average calculation. This exit sequence adjusts the PC so that it points to the second byte of the WAV object code ($3C), before the PC is stacked. Upon return from the interrupt, the $3C value is interpreted as a wavr pseudo-instruction. The wavr pseudo-instruction causes the CPU to execute a special WAV resumption sequence. The wavr recovery sequence adjusts the PC so that it looks like it did during execution of the original WAV instruction, then jumps back into the WAV processing loop. If another interrupt occurs before the weighted average calculation finishes, the PC is adjusted again as it was for the first interrupt. WAV can be interrupted any number of times, and additional WAV instructions can be executed while a WAV instruction is interrupted.

9.6.3 Cycle-by-Cycle Details for WAV and wavr

The WAV instruction is unusual in that the logic flow has two separate entry points. The first entry point is the normal start of a WAV instruction. The second entry point is used to resume the weighted average operation after a WAV instruction has been interrupted. This recovery operation is called the wavr pseudo-instruction.

Figure 9-12 is a flow diagram of the WAV instruction in the HCS12, including the wavr pseudo-instruction. **Figure 9-12** is a flow diagram of the WAV instruction in the M68HC12, including the wavr pseudo-instruction. Each rectangular box in these figures represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of the boxes correspond to execution cycle codes (refer to **Section 6. Instruction Glossary** for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

The cycle-by-cycle description provided here refers to the HCS12 flow in **Figure 9-11**. In terms of cycle-by-cycle bus activity, the $18 page select prebyte is treated as a special 1-byte instruction. In cycle 1.0 of the WAV instruction, one word of program information will be fetched into the instruction queue if the $18 is located at an odd address. If the $18 is at an even address, the instruction queue cannot advance so there is no bus access in this cycle.



WAV
1.0 - O
READ PROGRAM WORD IF $18 MISALIGNED
2.0 - f
NO BUS ACCESS
TMP1 = TMP2 = TMP3 = $0000
3.0 - f
NO BUS ACCESS
B = B – 1 DECREMENT ITERATION COUNTER
4.0 - f
READ BYTE @ 0,Y (FUZZY OUTPUT F_i)
Y = Y + 1 point at next fuzzy output
5.0 - r
READ BYTE @ 0,X (SINGLETON S_i)
X = X + 1 POINT AT NEXT SINGLETON
wavr
1.1 - U
READ WORD @ 0,SP (UNSTACK TMP1)
SP = SP + 2
2.1 - U
READ WORD @ 0,SP (UNSTACK TMP2)
SP = SP + 2
3.1 - U
READ WORD @ 0,SP (UNSTACK TMP3)
SP = SP + 2
4.1 - r
READ BYTE @ –1,Y (FUZZY OUTPUT F_i)
5.1 - r
READ BYTE @ –1,X (SINGLETON S_i)
INTERRUPT PENDING?
YES
NO
6.0 - f
NO BUS ACCESS
TMP3 = TMP3 + F_i
7.0 - f
NO BUS ACCESS
START MULTIPLY, PPROD = S_i*F_i
8.0 - f
NO BUS ACCESS
FINISH MULTIPLY, TMP2 = TMP2 + PPROD
9.0 - f
NO BUS ACCESS
TMP1 = TMP1 + (CARRY FROM PPROD ADD)
6.1 - S
WRITE WORD @ –2,SP (STACK TMP3)
SP = SP – 2
7.1 - S
WRITE WORD @ –2,SP (STACK TMP2)
SP = SP – 2
8.1 - S
WRITE WORD @ –2,SP (STACK TMP1)
SP = SP – 2
ADJUST PC TO POINT AT $3C wavr PSEUDO-OPCODE
CONTINUE TO INTERRUPT STACKING
NO
B = 0?
YES
10.0 - O
READ PROGRAM WORD IF $3C MISALIGNED
ADJUST PC TO POINT AT NEXT INSTRUCTION
Y : D = TMP1 : TMP2; X = TMP3
END

Figure 9-11. WAV and wavr Instruction Flow Diagram (for HCS12)

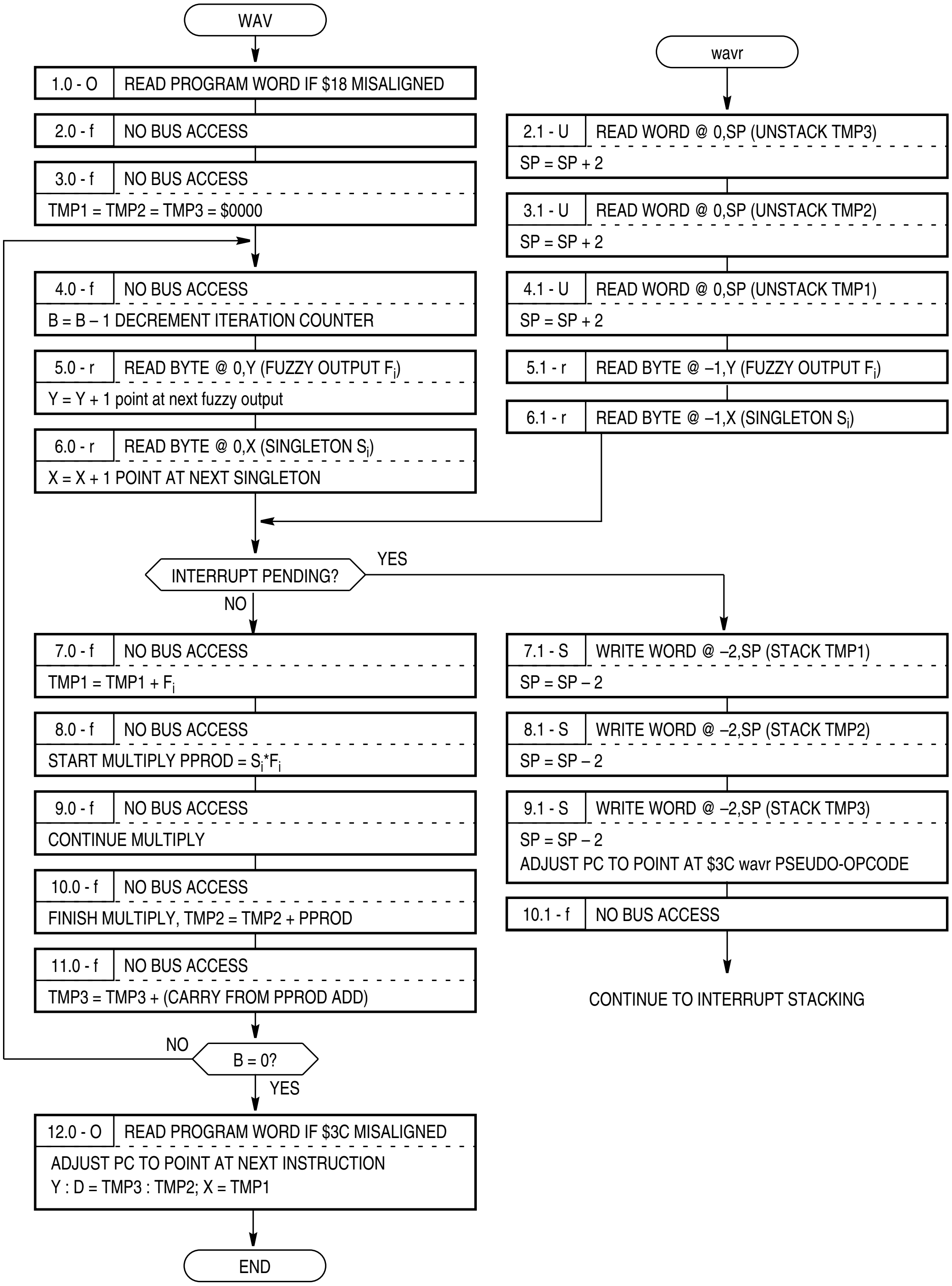

WAV
1.0 - O
READ PROGRAM WORD IF $18 MISALIGNED
2.0 - f
NO BUS ACCESS
3.0 - f
NO BUS ACCESS
TMP1 = TMP2 = TMP3 = $0000
4.0 - f
NO BUS ACCESS
B = B – 1 DECREMENT ITERATION COUNTER
5.0 - r
READ BYTE @ 0,Y (FUZZY OUTPUT F_i)
Y = Y + 1 point at next fuzzy output
6.0 - r
READ BYTE @ 0,X (SINGLETON S_i)
X = X + 1 POINT AT NEXT SINGLETON
wavr
2.1 - U
READ WORD @ 0,SP (UNSTACK TMP3)
SP = SP + 2
3.1 - U
READ WORD @ 0,SP (UNSTACK TMP2)
SP = SP + 2
4.1 - U
READ WORD @ 0,SP (UNSTACK TMP1)
SP = SP + 2
5.1 - r
READ BYTE @ –1,Y (FUZZY OUTPUT F_i)
6.1 - r
READ BYTE @ –1,X (SINGLETON S_i)
INTERRUPT PENDING?
YES
NO
7.0 - f
NO BUS ACCESS
TMP1 = TMP1 + F_i
8.0 - f
NO BUS ACCESS
START MULTIPLY PPROD = S_i*F_i
9.0 - f
NO BUS ACCESS
CONTINUE MULTIPLY
10.0 - f
NO BUS ACCESS
FINISH MULTIPLY, TMP2 = TMP2 + PPROD
11.0 - f
NO BUS ACCESS
TMP3 = TMP3 + (CARRY FROM PPROD ADD)
7.1 - S
WRITE WORD @ –2,SP (STACK TMP1)
SP = SP – 2
8.1 - S
WRITE WORD @ –2,SP (STACK TMP2)
SP = SP – 2
9.1 - S
WRITE WORD @ –2,SP (STACK TMP3)
SP = SP – 2
ADJUST PC TO POINT AT $3C wavr PSEUDO-OPCODE
10.1 - f
NO BUS ACCESS
CONTINUE TO INTERRUPT STACKING
NO
B = 0?
YES
12.0 - O
READ PROGRAM WORD IF $3C MISALIGNED
ADJUST PC TO POINT AT NEXT INSTRUCTION
Y : D = TMP3 : TMP2; X = TMP1
END

Figure 9-12. WAV and wavr Instruction Flow Diagram (for M68HC12)

In cycle 2.0, three internal 16-bit temporary registers are cleared in preparation for summation operations, but there is no bus access. The WAV instruction maintains a 32-bit sum-of-products in TMP1 : TMP2 and a 16-bit sum-of-weights in TMP3. By keeping these sums inside the CPU, bus accesses are reduced and the WAV operation is optimized for high speed.

Cycles 3.0 through 9.0 form the 7-cycle main loop for WAV. The value in the 8-bit B accumulator is used to count the number of loop iterations. B is decremented at the top of the loop in cycle 3.0, and the test for zero is located at the bottom of the loop after cycle 9.0. Cycle 4.0 and 5.0 are used to fetch the 8-bit operands for one iteration of the loop. X and Y index registers are used to access these operands. The index registers are incremented as the operands are fetched. Cycle 6.0 is used to accumulate the current fuzzy output into TMP3. Cycles 7.0 through 9.0 are used to perform the eight by eight multiply of F_i times S_i, and accumulate this result into TMP1 : TMP2. Even though the sum-of-products will not exceed 24 bits, the sum is maintained in the 32-bit combined TMP1 : TMP2 register because it is easier to use existing 16-bit operations than it would be to create a new smaller operation to handle the high order bits of this sum.

Since the weighted average operation could be quite long, it is made to be interruptible. The usual longest latency path is from very early in cycle 6.0, through cycle 9.0, to the top of the loop to cycle 3.0, through cycle 5.0 to the interrupt check.

If the WAV instruction is interrupted, the internal temporary registers TMP3, TMP2, and TMP1 need to be stored on the stack so the operation can be resumed. Since the WAV instruction included initialization in cycle 2.0, the recovery path after an interrupt needs to be different. The wavr pseudo-instruction has the same opcode as WAV, but it is on the first page of the opcode map so there is no page prebyte ($18) like there is for WAV. When WAV is interrupted, the PC is adjusted to point at the second byte of the WAV object code, so that it will be interpreted as the wavr pseudo-instruction on return from the interrupt, rather than the WAV instruction. During the recovery sequence, the PC is readjusted in case another interrupt comes before the weighted average operation finishes.

The resume sequence includes recovery of the temporary registers from the stack (1.1 through 3.1), and reads to get the operands for the current iteration. The normal WAV flow is then rejoined at cycle 6.0.

Upon normal completion of the instruction (cycle 10.0), the PC is adjusted so it points to the next instruction. The results are transferred from the TMP registers into CPU registers in such a way that the EDIV instruction can be used to divide the sum-of-products by the sum-of-weights. TMP1 : TMP2 is transferred into Y : D and TMP3 is transferred into X.

9.7 Custom Fuzzy Logic Programming

The basic fuzzy logic inference techniques described earlier are suitable for a broad range of applications, but some systems may require customization. The built-in fuzzy instructions use 8-bit resolution and some systems may require finer resolution. The rule evaluation instructions only support variations of MIN-MAX rule evaluation and other methods have been discussed in fuzzy logic literature. The weighted average of singletons is not the only defuzzification technique. The CPU12 has several instructions and addressing modes that can be helpful when developing custom fuzzy logic systems.

9.7.1 Fuzzification Variations

The MEM instruction supports trapezoidal membership functions and several other varieties, including membership functions with vertical sides (infinite slope sides). Triangular membership functions are a subset of trapezoidal functions. Some practitioners refer to s-, z-, and π-shaped membership functions. These refer to a trapezoid butted against the right end of the x-axis, a trapezoid butted against the left end of the x-axis, and a trapezoidal membership function that isn’t butted against either end of the x-axis, respectively. Many other membership function shapes are possible, if memory space and processing bandwidth are sufficient.

Tabular membership functions offer complete flexibility in shape and very fast evaluation time. However, tables take a very large amount of memory space (as many as 256 bytes per label of one system input). The excessive size to specify tabular membership functions makes them impractical for most microcontroller-based fuzzy systems. The CPU12 instruction set includes two instructions (TBL and ETBL) for lookup and interpolation of compressed tables.

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. A flexible 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 can effectively be 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-TBL or word-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.

Because indexed addressing mode is used to identify the starting point of the line segment of interest, there is a great deal of flexibility in constructing tables. A common method is to break the x-axis range into 256 equal width segments and store the y value for each of the resulting 257 endpoints. The 16-bit D accumulator is then used as the x input to the table. The upper eight bits (A) is used as a coarse lookup to find the line segment of interest, and the lower eight bits (B) is used to interpolate within this line segment.

In the program sequence

```
    LDX        #TBL_START
    LDD        DATA_IN
    TBL        A,X
```

The notation A,X causes the TBL instruction to use the Ath line segment in the table. The low-order half of D (B) is used by TBL to calculate the exact data value from this line segment. This type of table uses only 257 entries to approximate a table with 16 bits of resolution. This type of table has the disadvantage of equal width line segments, which means just as many points are needed to describe a flat portion of the desired function as are needed for the most active portions.

Another type of table stores x:y coordinate pairs for the endpoints of each linear segment. This type of table may reduce the table storage space compared to the previous fixed-width segments because flat areas of the functions can be specified with a single pair of endpoints. This type of table is a little harder to use with the CPU12 TBL and ETBL instructions because the table instructions expect y-values for segment endpoints to be in consecutive memory locations.

Consider a table made up of an arbitrary number of x:y coordinate pairs, where all values are eight bits. The table is entered with the x-coordinate of the desired point to lookup in the A accumulator. When the table is exited, the corresponding y-value is in the A accumulator. **Figure 9-13** shows one way to work with this type of table.

```
BEGIN       LDY      #TABLE_START-2    ;setup initial table pointer
FIND_LOOP   CMPA     2,+Y              ;find first Xn > XL
                                       ;(auto pre-inc Y by 2)
            BLS      FIND_LOOP         ;loop if XL .le. Xn
* on fall thru, XB@-2,Y YB@-1,Y XE@0,Y and YE@1,Y
            TFR      D,X               ;save XL in high half of X
            CLRA                       ;zero upper half of D
            LDAB     0,Y               ;D = 0:XE
            SUBB     -2,Y              ;D = 0:(XE-XB)
            EXG      D,X               ;X = (XE-XB).. D = XL:junk
            SUBA     -2,Y              ;A = (XL-XB)
            EXG      A,D               ;D = 0:(XL-XB), uses trick of EXG
            FDIV                       ;X reg = (XL-XB)/(XE-XB)
            EXG      D,X               ;move fractional result to A:B
            EXG      A,B               ;byte swap - need result in B
            TSTA                       ;check for rounding
            BPL      NO_ROUND
            INCB                       ;round B up by 1
NO_ROUND    LDAA     1,Y               ;YE
            PSHA                       ;put on stack for TBL later
            LDAA     -1,Y              ;YB
            PSHA                       ;now YB@0,SP and YE@1,SP
            TBL      2,SP+             ;interpolate and deallocate
                                       ;stack temps
```

Figure 9-13. Endpoint Table Handling

The basic idea is to find the segment of interest, temporarily build a 1-segment table of the correct format on the stack, then use TBL with stack relative indexed addressing to interpolate. The most difficult part of the routine is calculating the proportional distance from the beginning of the segment to the lookup point versus the width of the segment ((XL–XB)/(XE–XB)). With this type of table, this calculation must be done at run time. In the previous type of table, this proportional term is an inherent part (the lowest order bits) of the data input to the table.

Some fuzzy theorists have suggested membership functions should be shaped like normal distribution curves or other mathematical functions. This may be correct, but the processing requirements to solve for an intercept on such a function would be unacceptable for most microcontroller-based fuzzy systems. Such a function could be encoded into a table of one of the previously described types.

For many common systems, the thing that is most important about membership function shape is that there is a gradual transition from non-membership to membership as the system input value approaches the central range of the membership function.

Examine the human problem of stopping a car at an intersection. Rules such as “If intersection is close and speed is fast, apply brakes” might be used.

The meaning (reflected in membership function shape and position) of the labels “close” and “fast” will be different for a teenager than they are for a grandmother, but both can accomplish the goal of stopping. It makes intuitive sense that the exact shape of a membership function is much less important than the fact that it has gradual boundaries.

9.7.2 Rule Evaluation Variations

The REV and REVW instructions expect fuzzy input and fuzzy output values to be 8-bit values. In a custom fuzzy inference program, higher resolution may be desirable (although this is not a common requirement). The CPU12 includes variations of minimum and maximum operations that work with the fuzzy MIN-MAX inference algorithm. The problem with the fuzzy inference algorithm is that the min and max operations need to store their results differently, so the min and max instructions must work differently or more than one variation of these instructions is needed.

The CPU12 has MIN and MAX instructions for 8- or 16-bit operands, where one operand is in an accumulator and the other is a referenced memory location. There are separate variations that replace the accumulator or the memory location with the result. While processing rule antecedents in a fuzzy inference program, a reference value must be compared to each of the referenced fuzzy inputs, and the smallest input must end up in an accumulator. The instruction

```
     EMIND     2,X+    ;process one rule antecedent
```

automates the central operations needed to process rule antecedents. The E stands for extended, so this instruction compares 16-bit operands. The D at the end of the mnemonic stands for the D accumulator, which is both the first operand for the comparison and the destination of the result. The 2,X+ is an indexed addressing specification that says X points to the second operand for the comparison and it will be post-incremented by 2 to point at the next rule antecedent.

When processing rule consequents, the operand in the accumulator must remain constant (in case there is more than one consequent in the rule), and the result of the comparison must replace the referenced fuzzy output in RAM. To do this, use the instruction

```
     EMAXM     2,X+    ;process one rule consequent
```

The M at the end of the mnemonic indicates that the result will replace the referenced memory operand. Again, indexed addressing is used. These two instructions would form the working part of a 16-bit resolution fuzzy inference routine.

There are many other methods of performing inference, but none of these are as widely used as the min-max method. Since the CPU12 is a general-purpose microcontroller, the programmer has complete freedom to program any algorithm desired. A custom programmed algorithm would typically take more code space and execution time than a routine that used the built-in REV or REVW instructions.

9.7.3 Defuzzification Variations

Other CPU12 instructions can help with custom defuzzification routines in two main areas:

- The first case is working with operands that are more than eight bits.
- The second case involves using an entirely different approach than weighted average of singletons.

The primary part of the WAV instruction is a multiply and accumulate operation to get the numerator for the weighted average calculation. When working with operands as large as 16 bits, the EMACS instruction could at least be used to automate the multiply and accumulate function. The CPU12 has extended math capabilities, including the EMACS instruction which uses 16-bit input operands and accumulates the sum to a 32-bit memory location and 32-bit by 16-bit divide instructions.

One benefit of the WAV instruction is that both a sum of products and a sum of weights are maintained, while the fuzzy output operand is only accessed from memory once. Since memory access time is such a significant part of execution time, this provides a speed advantage compared to conventional instructions.

The weighted average of singletons is the most commonly used technique in microcontrollers because it is computationally less difficult than most other methods. The simplest method is called max defuzzification, which simply uses the largest fuzzy output as the system result. However, this approach does not take into account any other fuzzy outputs, even when they are almost as true as the chosen max output. Max defuzzification is not a good general choice because it only works for a subset of fuzzy logic applications.

The CPU12 is well suited for more computationally challenging algorithms than weighted average. A 32-bit by 16-bit divide instruction takes 11 or 12 25-MHz cycles for unsigned or signed variations. A 16-bit by 16-bit multiply with a 32-bit result takes only three 25-MHz cycles. The EMACS instruction uses 16-bit operands and accumulates the result in a 32-bit memory location, taking only 12 25-MHz cycles per iteration, including accessing all operands from memory and storing the result to memory.

Appendix A. Instruction Reference

A.1 Introduction

This appendix provides quick references for the instruction set, opcode map, and encoding.

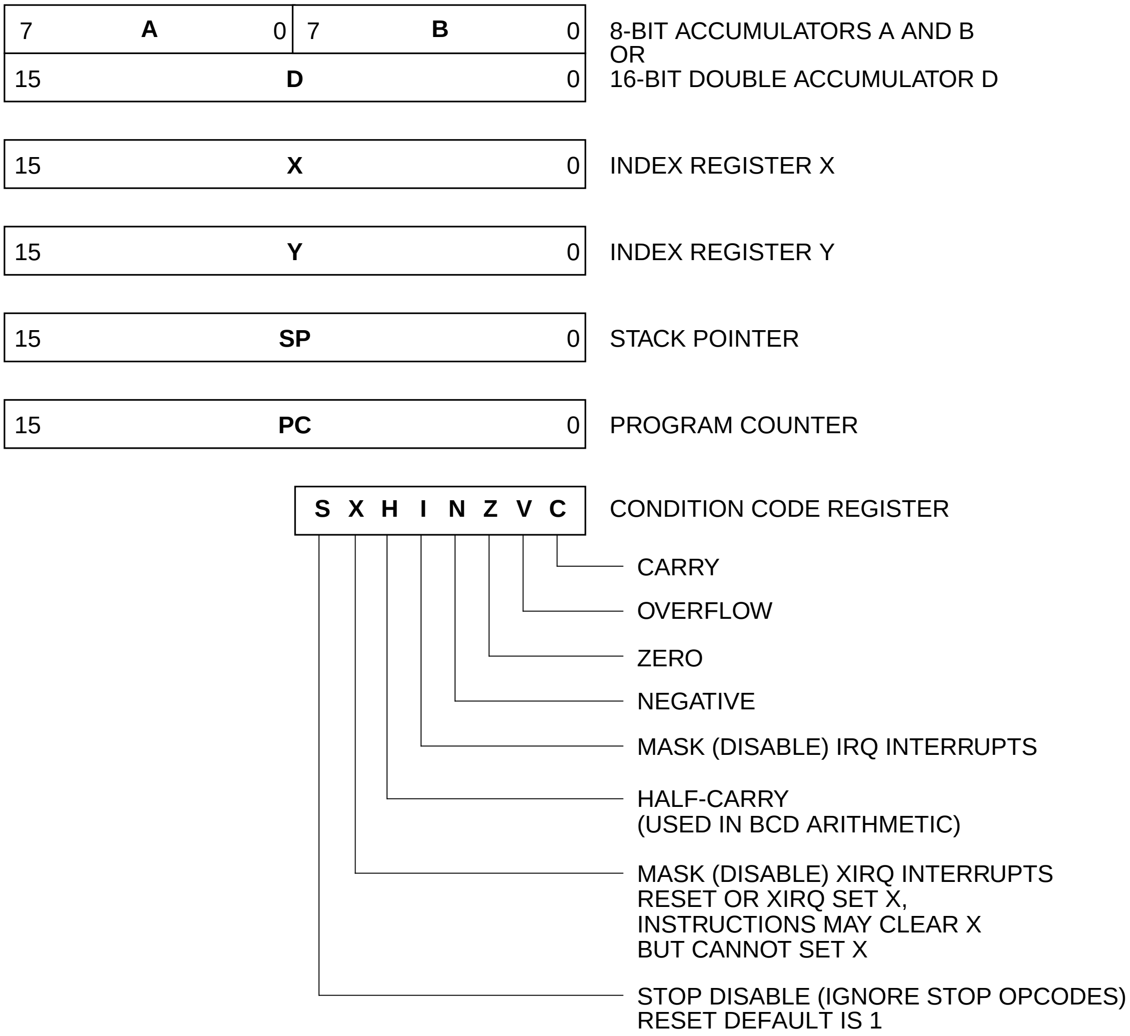
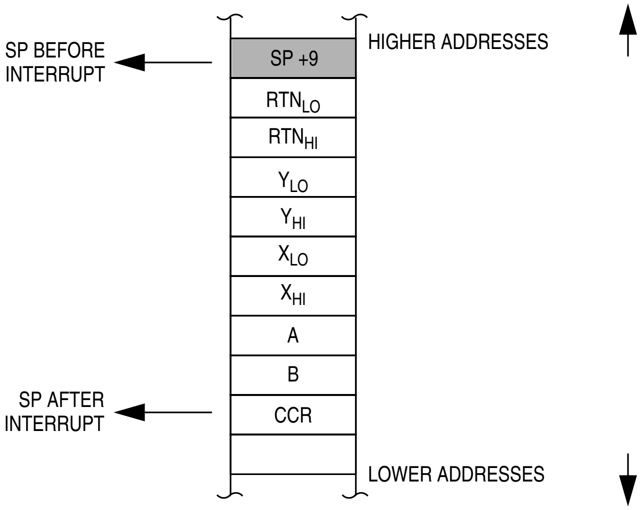


Figure A-1. Programming Model

A.2 Stack and Memory Layout



STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS ODD BEFORE INTERRUPT

SP +8	RTN_{LO}	▨	SP +9
SP +6	Y_{LO}	RTN_{HI}	SP +7
SP +4	X_{LO}	Y_{HI}	SP +5
SP +2	A	X_{HI}	SP +3
SP	CCR	B	SP +1
SP –2			SP –1

STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS EVEN BEFORE INTERRUPT

SP +9	▨		SP +10
SP +7	RTN_{HI}	RTN_{LO}	SP +8
SP +5	Y_{HI}	Y_{LO}	SP +6
SP +4	X_{HI}	X_{LO}	SP +4
SP +1	B	A	SP +2
SP –1		CCR	SP

A.3 Interrupt Vector Locations

$FFFE, $FFFF	Power-On (POR) or External Reset
$FFFC, $FFFD	Clock Monitor Reset
$FFFA, $FFFB	Computer Operating Properly (COP Watchdog Reset
$FFF8, $FFF9	Unimplemented Opcode Trap
$FFF6, $FFF7	Software Interrupt Instruction (SWI)
$FFF4, $FFF5	XIRQ
$FFF2, $FFF3	IRQ
$FFC0–$FFF1 (M68HC12)	Device-Specific Interrupt Sources
$FF00–$FFF1 (HCS12)	Device-Specific Interrupt Sources

A.4 Notation Used in Instruction Set Summary

CPU Register Notation

Accumulator A — A or a
Accumulator B — B or b
Accumulator D — D or d
Index Register X — X or x
Index Register Y — Y or y
Stack Pointer — SP, sp, or s
Program Counter — PC, pc, or p
Condition Code Register — CCR or c

Explanation of Italic Expressions in Source Form Column

abc — A or B or CCR
abcdxys — A or B or CCR or D or X or Y or SP. Some assemblers also allow T2 or T3.
abd — A or B or D
abdxys — A or B or D or X or Y or SP
dxys — D or X or Y or SP
msk8 — 8-bit mask, some assemblers require # symbol before value
opr8i — 8-bit immediate value
opr16i — 16-bit immediate value
opr8a — 8-bit address used with direct address mode
opr16a — 16-bit address value
oprx0_xysp — Indexed addressing postbyte code:
oprx3,–*xys* Predecrement X or Y or SP by 1 . . . 8
oprx3,+*xys* Preincrement X or Y or SP by 1 . . . 8
oprx3,*xys*– Postdecrement X or Y or SP by 1 . . . 8
oprx3,*xys*+ Postincrement X or Y or SP by 1 . . . 8
oprx5,*xysp* 5-bit constant offset from X or Y or SP or PC
abd,*xysp* Accumulator A or B or D offset from X or Y or SP or PC
oprx3 — Any positive integer 1 . . . 8 for pre/post increment/decrement
oprx5 — Any integer in the range –16 . . . +15
oprx9 — Any integer in the range –256 . . . +255
oprx16 — Any integer in the range –32,768 . . . 65,535
page — 8-bit value for PPAGE, some assemblers require # symbol before this value
rel8 — Label of branch destination within –128 to +127 locations
rel9 — Label of branch destination within –256 to +255 locations
rel16 — Any label within 64K memory space
trapnum — Any 8-bit integer in the range $30-$39 or $40-$FF
xys — X or Y or SP
xysp — X or Y or SP or PC

Operators

+ — Addition
– — Subtraction
• — Logical AND
+ — Logical OR (inclusive)

Continued on next page

Operators (continued)

⊕ — Logical exclusive OR

× — Multiplication

÷ — Division

$\overline{M}$ — Negation. One’s complement (invert each bit of M)

: — Concatenate
Example: A : B means the 16-bit value formed by concatenating 8-bit accumulator A with 8-bit accumulator B.
A is in the high-order position.

⇒ — Transfer
Example: (A) ⇒ M means the content of accumulator A is transferred to memory location M.

⇔ — Exchange
Example: D ⇔ X means exchange the contents of D with those of X.

Address Mode Notation

INH — Inherent; no operands in object code

IMM — Immediate; operand in object code

DIR — Direct; operand is the lower byte of an address from $0000 to $00FF

EXT — Operand is a 16-bit address

REL — Two’s complement relative offset; for branch instructions

IDX — Indexed (no extension bytes); includes:
5-bit constant offset from X, Y, SP, or PC
Pre/post increment/decrement by 1 . . . 8
Accumulator A, B, or D offset

IDX1 — 9-bit signed offset from X, Y, SP, or PC; 1 extension byte

IDX2 — 16-bit signed offset from X, Y, SP, or PC; 2 extension bytes

[IDX2] — Indexed-indirect; 16-bit offset from X, Y, SP, or PC

[D, IDX] — Indexed-indirect; accumulator D offset from X, Y, SP, or PC

Machine Coding

`dd` — 8-bit direct address $0000 to $00FF. (High byte assumed to be $00).

`ee` — High-order byte of a 16-bit constant offset for indexed addressing.

`eb` — Exchange/Transfer post-byte. See **Table A-5** on page 399.

`ff` — Low-order eight bits of a 9-bit signed constant offset for indexed addressing, or low-order byte of a 16-bit constant offset for indexed addressing.

`hh` — High-order byte of a 16-bit extended address.

`ii` — 8-bit immediate data value.

`jj` — High-order byte of a 16-bit immediate data value.

`kk` — Low-order byte of a 16-bit immediate data value.

`lb` — Loop primitive (DBNE) post-byte. See **Table A-6** on page 400.

`ll` — Low-order byte of a 16-bit extended address.

`mm` — 8-bit immediate mask value for bit manipulation instructions.
Set bits indicate bits to be affected.

`pg` — Program page (bank) number used in CALL instruction.

`qq` — High-order byte of a 16-bit relative offset for long branches.

`tn` — Trap number $30–$39 or $40–$FF.

`rr` — Signed relative offset $80 (–128) to $7F (+127).
Offset relative to the byte following the relative offset byte, or
low-order byte of a 16-bit relative offset for long branches.

`xb` — Indexed addressing post-byte. See **Table A-3** on page 397
and **Table A-4** on page 398.

Access Detail

Each code letter except (,), and comma equals one CPU cycle. Uppercase = 16-bit operation and lowercase = 8-bit operation. For complex sequences see the *CPU12 Reference Manual* (CPU12RM/AD) for more detailed information.

`f` — Free cycle, CPU doesn’t use bus
`g` — Read PPAGE internally
`I` — Read indirect pointer (indexed indirect)
`i` — Read indirect PPAGE value (CALL indirect only)
`n` — Write PPAGE internally
`O` — Optional program word fetch (P) if instruction is misaligned and has an odd number of bytes of object code — otherwise, appears as a free cycle (f); Page 2 prebyte treated as a separate 1-byte instruction
`P` — Program word fetch (always an aligned-word read)
`r` — 8-bit data read
`R` — 16-bit data read
`s` — 8-bit stack write
`S` — 16-bit stack write
`w` — 8-bit data write
`W` — 16-bit data write
`u` — 8-bit stack read
`U` — 16-bit stack read
`V` — 16-bit vector fetch (always an aligned-word read)
`t` — 8-bit conditional read (or free cycle)
`T` — 16-bit conditional read (or free cycle)
`x` — 8-bit conditional write (or free cycle)
`()` — Indicate a microcode loop
`,` — Indicates where an interrupt could be honored

Special Cases

`PPP/P` — Short branch, PPP if branch taken, P if not
`OPPP/OPO` — Long branch, OPPP if branch taken, OPO if not

Condition Codes Columns

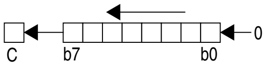
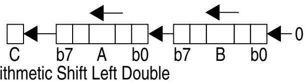
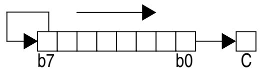
– — Status bit not affected by operation.
0 — Status bit cleared by operation.
1 — Status bit set by operation.
Δ — Status bit affected by operation.
⇓ — Status bit may be cleared or remain set, but is not set by operation.
⇑ — Status bit may be set or remain cleared, but is not cleared by operation.
? — Status bit may be changed by operation but the final state is not defined.
! — Status bit used for a special purpose.

Table A-1. Instruction Set Summary (Sheet 1 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12	Access Detail M68HC12	S X H I	N Z V C
ABA	(A) + (B) ⇒ A Add Accumulators A and B	INH	`18 06`	`OO`	`OO`	– – Δ –	Δ Δ Δ Δ
ABX	(B) + (X) ⇒ X *Translates to* LEAX B,X	IDX	`1A E5`	`Pf`	`PP`[1]	– – – –	– – – –
ABY	(B) + (Y) ⇒ Y *Translates to* LEAY B,Y	IDX	`19 ED`	`Pf`	`PP`[1]	– – – –	– – – –
ADCA *#opr8i*	(A) + (M) + C ⇒ A	IMM	`89 ii`	`P`	`P`	– – Δ –	Δ Δ Δ Δ
ADCA *opr8a*	Add with Carry to A	DIR	`99 dd`	`rPf`	`rfP`		
ADCA *opr16a*		EXT	`B9 hh ll`	`rPO`	`rOP`		
ADCA *oprx0_xysp*		IDX	`A9 xb`	`rPf`	`rfP`		
ADCA *oprx9,xysp*		IDX1	`A9 xb ff`	`rPO`	`rPO`		
ADCA *oprx16,xysp*		IDX2	`A9 xb ee ff`	`frPP`	`frPP`		
ADCA [D,*xysp*]		[D,IDX]	`A9 xb`	`fIfrPf`	`fIfrfP`		
ADCA [*oprx16,xysp*]		[IDX2]	`A9 xb ee ff`	`fIPrPf`	`fIPrfP`		
ADCB *#opr8i*	(B) + (M) + C ⇒ B	IMM	`C9 ii`	`P`	`P`	– – Δ –	Δ Δ Δ Δ
ADCB *opr8a*	Add with Carry to B	DIR	`D9 dd`	`rPf`	`rfP`		
ADCB *opr16a*		EXT	`F9 hh ll`	`rPO`	`rOP`		
ADCB *oprx0_xysp*		IDX	`E9 xb`	`rPf`	`rfP`		
ADCB *oprx9,xysp*		IDX1	`E9 xb ff`	`rPO`	`rPO`		
ADCB *oprx16,xysp*		IDX2	`E9 xb ee ff`	`frPP`	`frPP`		
ADCB [D,*xysp*]		[D,IDX]	`E9 xb`	`fIfrPf`	`fIfrfP`		
ADCB [*oprx16,xysp*]		[IDX2]	`E9 xb ee ff`	`fIPrPf`	`fIPrfP`		
ADDA *#opr8i*	(A) + (M) ⇒ A	IMM	`8B ii`	`P`	`P`	– – Δ –	Δ Δ Δ Δ
ADDA *opr8a*	Add without Carry to A	DIR	`9B dd`	`rPf`	`rfP`		
ADDA *opr16a*		EXT	`BB hh ll`	`rPO`	`rOP`		
ADDA *oprx0_xysp*		IDX	`AB xb`	`rPf`	`rfP`		
ADDA *oprx9,xysp*		IDX1	`AB xb ff`	`rPO`	`rPO`		
ADDA *oprx16,xysp*		IDX2	`AB xb ee ff`	`frPP`	`frPP`		
ADDA [D,*xysp*]		[D,IDX]	`AB xb`	`fIfrPf`	`fIfrfP`		
ADDA [*oprx16,xysp*]		[IDX2]	`AB xb ee ff`	`fIPrPf`	`fIPrfP`		
ADDB *#opr8i*	(B) + (M) ⇒ B	IMM	`CB ii`	`P`	`P`	– – Δ –	Δ Δ Δ Δ
ADDB *opr8a*	Add without Carry to B	DIR	`DB dd`	`rPf`	`rfP`		
ADDB *opr16a*		EXT	`FB hh ll`	`rPO`	`rOP`		
ADDB *oprx0_xysp*		IDX	`EB xb`	`rPf`	`rfP`		
ADDB *oprx9,xysp*		IDX1	`EB xb ff`	`rPO`	`rPO`		
ADDB *oprx16,xysp*		IDX2	`EB xb ee ff`	`frPP`	`frPP`		
ADDB [D,*xysp*]		[D,IDX]	`EB xb`	`fIfrPf`	`fIfrfP`		
ADDB [*oprx16,xysp*]		[IDX2]	`EB xb ee ff`	`fIPrPf`	`fIPrfP`		
ADDD *#opr16i*	(A:B) + (M:M+1) ⇒ A:B	IMM	`C3 jj kk`	`PO`	`OP`	– – – –	Δ Δ Δ Δ
ADDD *opr8a*	Add 16-Bit to D (A:B)	DIR	`D3 dd`	`RPf`	`RfP`		
ADDD *opr16a*		EXT	`F3 hh ll`	`RPO`	`ROP`		
ADDD *oprx0_xysp*		IDX	`E3 xb`	`RPf`	`RfP`		
ADDD *oprx9,xysp*		IDX1	`E3 xb ff`	`RPO`	`RPO`		
ADDD *oprx16,xysp*		IDX2	`E3 xb ee ff`	`fRPP`	`fRPP`		
ADDD [D,*xysp*]		[D,IDX]	`E3 xb`	`fIfRPf`	`fIfRfP`		
ADDD [*oprx16,xysp*]		[IDX2]	`E3 xb ee ff`	`fIPRPf`	`fIPRfP`		
ANDA *#opr8i*	(A) • (M) ⇒ A	IMM	`84 ii`	`P`	`P`	– – – –	Δ Δ 0 –
ANDA *opr8a*	Logical AND A with Memory	DIR	`94 dd`	`rPf`	`rfP`		
ANDA *opr16a*		EXT	`B4 hh ll`	`rPO`	`rOP`		
ANDA *oprx0_xysp*		IDX	`A4 xb`	`rPf`	`rfP`		
ANDA *oprx9,xysp*		IDX1	`A4 xb ff`	`rPO`	`rPO`		
ANDA *oprx16,xysp*		IDX2	`A4 xb ee ff`	`frPP`	`frPP`		
ANDA [D,*xysp*]		[D,IDX]	`A4 xb`	`fIfrPf`	`fIfrfP`		
ANDA [*oprx16,xysp*]		[IDX2]	`A4 xb ee ff`	`fIPrPf`	`fIPrfP`		
ANDB *#opr8i*	(B) • (M) ⇒ B	IMM	`C4 ii`	`P`	`P`	– – – –	Δ Δ 0 –
ANDB *opr8a*	Logical AND B with Memory	DIR	`D4 dd`	`rPf`	`rfP`		
ANDB *opr16a*		EXT	`F4 hh ll`	`rPO`	`rOP`		
ANDB *oprx0_xysp*		IDX	`E4 xb`	`rPf`	`rfP`		
ANDB *oprx9,xysp*		IDX1	`E4 xb ff`	`rPO`	`rPO`		
ANDB *oprx16,xysp*		IDX2	`E4 xb ee ff`	`frPP`	`frPP`		
ANDB [D,*xysp*]		[D,IDX]	`E4 xb`	`fIfrPf`	`fIfrfP`		
ANDB [*oprx16,xysp*]		[IDX2]	`E4 xb ee ff`	`fIPrPf`	`fIPrfP`		
ANDCC *#opr8i*	(CCR) • (M) ⇒ CCR Logical AND CCR with Memory	IMM	`10 ii`	`P`	`P`	⇓ ⇓ ⇓ ⇓	⇓ ⇓ ⇓ ⇓

Note 1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.

Table A-1. Instruction Set Summary (Sheet 2 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
ASL *opr16a* ASL *oprx0_xysp* ASL *oprx9,xysp* ASL *oprx16,xysp* ASL [D,*xysp*] ASL [*oprx16,xysp*] ASLA ASLB	 Arithmetic Shift Left Arithmetic Shift Left Accumulator A Arithmetic Shift Left Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	----	Δ Δ Δ Δ
ASLD	 Arithmetic Shift Left Double	INH	59	O	O	----	Δ Δ Δ Δ
ASR *opr16a* ASR *oprx0_xysp* ASR *oprx9,xysp* ASR *oprx16,xysp* ASR [D,*xysp*] ASR [*oprx16,xysp*] 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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	----	Δ Δ Δ Δ
BCC *rel8*	Branch if Carry Clear (if C = 0)	REL	24 rr	PPP/P[1]	PPP/P[1]	----	----
BCLR *opr*8a, *msk8* BCLR *opr16a*, *msk8* BCLR *oprx0_xysp*, *msk8* BCLR *oprx9,xysp*, *msk8* BCLR *oprx16,xysp*, *msk8*	$(M) \bullet (\overline{mm}) \Rightarrow M$ 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	rPwO rPwP rPwO rPwP frPwPO	rPOw rPPw rPOw rPwP frPwOP	----	Δ Δ 0 –
BCS *rel8*	Branch if Carry Set (if C = 1)	REL	25 rr	PPP/P[1]	PPP/P[1]	----	----
BEQ *rel8*	Branch if Equal (if Z = 1)	REL	27 rr	PPP/P[1]	PPP/P[1]	----	----
BGE *rel8*	Branch if Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	2C rr	PPP/P[1]	PPP/P[1]	----	----
BGND	Place CPU in Background Mode see *CPU12 Reference Manual*	INH	00	VfPPP	VfPPP	----	----
BGT *rel8*	Branch if Greater Than (if Z + (N ⊕ V) = 0) (signed)	REL	2E rr	PPP/P[1]	PPP/P[1]	----	----
BHI *rel8*	Branch if Higher (if C + Z = 0) (unsigned)	REL	22 rr	PPP/P[1]	PPP/P[1]	----	----
BHS *rel8*	Branch if Higher or Same (if C = 0) (unsigned) same function as BCC	REL	24 rr	PPP/P[1]	PPP/P[1]	----	----
BITA #*opr8i* BITA *opr8a* BITA *opr16a* BITA *oprx0_xysp* BITA *oprx9,xysp* BITA *oprx16,xysp* BITA [D,*xysp*] BITA [*oprx16,xysp*]	(A) • (M) Logical AND A with Memory Does not change Accumulator or 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	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	P rfP rOP rfP rPO frPP fIfrfP fIPrfP	----	Δ Δ 0 –
BITB #*opr8i* BITB *opr8a* BITB *opr16a* BITB *oprx0_xysp* BITB *oprx9,xysp* BITB *oprx16,xysp* BITB [D,*xysp*] BITB [*oprx16,xysp*]	(B) • (M) Logical AND B with Memory Does not change Accumulator or 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	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	P rfP rOP rfP rPO frPP fIfrfP fIPrfP	----	Δ Δ 0 –
BLE *rel8*	Branch if Less Than or Equal (if Z + (N ⊕ V) = 1) (signed)	REL	2F rr	PPP/P[1]	PPP/P[1]	----	----
BLO *rel8*	Branch if Lower (if C = 1) (unsigned) same function as BCS	REL	25 rr	PPP/P[1]	PPP/P[1]	----	----

Note 1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Table A-1. Instruction Set Summary (Sheet 3 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
BLS *rel8*	Branch if Lower or Same (if C + Z = 1) (unsigned)	REL	`23 rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BLT *rel8*	Branch if Less Than (if N ⊕ V = 1) (signed)	REL	`2D rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BMI *rel8*	Branch if Minus (if N = 1)	REL	`2B rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BNE *rel8*	Branch if Not Equal (if Z = 0)	REL	`26 rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BPL *rel8*	Branch if Plus (if N = 0)	REL	`2A rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BRA *rel8*	Branch Always (if 1 = 1)	REL	`20 rr`	PPP	PPP	– – – –	– – – –

BRCLR *opr8a, msk8, rel8*
BRCLR *opr16a, msk8, rel8*
BRCLR *oprx0_xysp, msk8, rel8*
BRCLR *oprx9,xysp, msk8, rel8*
BRCLR *oprx16,xysp, msk8, rel8*
	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`
				rPPP
				rfPPP
				rPPP
				rfPPP
				PrfPPP
					rPPP
					rfPPP
					rPPP
					rffPPP
					frPffPPP
						– – – –	– – – –

BRN *rel8*	Branch Never (if 1 = 0)	REL	`21 rr`	P	P	– – – –	– – – –

BRSET *opr8, msk8, rel8*
BRSET *opr16a, msk8, rel8*
BRSET *oprx0_xysp, msk8, rel8*
BRSET *oprx9,xysp, msk8, rel8*
BRSET *oprx16,xysp, msk8, rel8*
	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`
				rPPP
				rfPPP
				rPPP
				rfPPP
				PrfPPP
					rPPP
					rfPPP
					rPPP
					rffPPP
					frPffPPP
						– – – –	– – – –

BSET *opr8, msk8*
BSET *opr16a, msk8*
BSET *oprx0_xysp, msk8*
BSET *oprx9,xysp, msk8*
BSET *oprx16,xysp, msk8*
	(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`
				rPwO
				rPwP
				rPwO
				rPwP
				frPwPO
					rPOw
					rPPw
					rPOw
					rPwP
					frPwOP
						– – – –	Δ Δ 0 –

BSR *rel8*	(SP) – 2 ⇒ SP; RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$ Subroutine address ⇒ PC Branch to Subroutine	REL	`07 rr`	SPPP	PPPS	– – – –	– – – –
BVC *rel8*	Branch if Overflow Bit Clear (if V = 0)	REL	`28 rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BVS *rel8*	Branch if Overflow Bit Set (if V = 1)	REL	`29 rr`	PPP/P[1]	PPP/P[1]	– – – –	– – – –

CALL *opr16a*, *page*
CALL *oprx0_xysp*, *page*
CALL *oprx9,xysp*, *page*
CALL *oprx16,xysp*, *page*
CALL [D,*xysp*]
CALL [*oprx16, xysp*]
	(SP) – 2 ⇒ SP; RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$
	(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.)
	
	Indirect modes get program address
	and new pg value based on pointer.
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`4A hh ll pg`
			`4B xb pg`
			`4B xb ff pg`
			`4B xb ee ff pg`
			`4B xb`
			`4B xb ee ff`
				gnSsPPP
				gnSsPPP
				gnSsPPP
				fgnSsPPP
				fIignSsPPP
				fIignSsPPP
					gnfSsPPP
					gnfSsPPP
					gnfSsPPP
					fgnfSsPPP
					fIignSsPPP
					fIignSsPPP
						– – – –	– – – –

CBA	(A) – (B) Compare 8-Bit Accumulators	INH	`18 17`	OO	OO	– – – –	Δ Δ Δ Δ
CLC	0 ⇒ C *Translates to* ANDCC #$FE	IMM	`10 FE`	P	P	– – – –	– – – 0
CLI	0 ⇒ I *Translates to* ANDCC #$EF (enables I-bit interrupts)	IMM	`10 EF`	P	P	– – – 0	– – – –

CLR *opr16a*
CLR *oprx0_xysp*
CLR *oprx9,xysp*
CLR *oprx16,xysp*
CLR [D,*xysp*]
CLR [*oprx16,xysp*]
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`
				PwO
				Pw
				PwO
				PwP
				PIfw
				PIPw
				O
				O
					wOP
					Pw
					PwO
					PwP
					PIfPw
					PIPPw
					O
					O
						– – – –	0 1 0 0

CLV	0 ⇒ V *Translates to* ANDCC #$FD	IMM	`10 FD`	P	P	– – – –	– – 0 –

Note 1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.

Source Form	Operation	Addr. Mode	Machine Coding (hex)	HCS12	M68HC12	S X H I	N Z V C

CMPA #*opr8i*
CMPA *opr8a*
CMPA *opr16a*
CMPA *oprx0_xysp*
CMPA *oprx9,xysp*
CMPA *oprx16,xysp*
CMPA [D,*xysp*]
CMPA [*oprx16,xysp*]
	(A) – (M)
	Compare Accumulator A with Memory
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`81 ii`
			`91 dd`
			`B1 hh ll`
			`A1 xb`
			`A1 xb ff`
			`A1 xb ee ff`
			`A1 xb`
			`A1 xb ee ff`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ Δ Δ

Table A-1. Instruction Set Summary (Sheet 4 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
CMPB #*opr8i*	(B) – (M)	IMM	C1 ii	P	P	– – – –	Δ Δ Δ Δ
CMPB *opr8a*	Compare Accumulator B with Memory	DIR	D1 dd	rPf	rfP		
CMPB *opr16a*		EXT	F1 hh ll	rPO	rOP		
CMPB *oprx0_xysp*		IDX	E1 xb	rPf	rfP		
CMPB *oprx9,xysp*		IDX1	E1 xb ff	rPO	rPO		
CMPB *oprx16,xysp*		IDX2	E1 xb ee ff	frPP	frPP		
CMPB [D,*xysp*]		[D,IDX]	E1 xb	fIfrPf	fIfrfP		
CMPB [*oprx16,xysp*]		[IDX2]	E1 xb ee ff	fIPrPf	fIPrfP		
COM *opr16a*	$(\overline{M}) \Rightarrow M$ *equivalent to* $FF – (M) ⇒ M	EXT	71 hh ll	rPwO	rOPw	– – – –	Δ Δ 0 1
COM *oprx0_xysp*	1's Complement Memory Location	IDX	61 xb	rPw	rPw		
COM *oprx9,xysp*		IDX1	61 xb ff	rPwO	rPOw		
COM *oprx16,xysp*		IDX2	61 xb ee ff	frPwP	frPPw		
COM [D,*xysp*]		[D,IDX]	61 xb	fIfrPw	fIfrPw		
COM [*oprx16,xysp*]		[IDX2]	61 xb ee ff	fIPrPw	fIPrPw		
COMA	$(\overline{A}) \Rightarrow A$ Complement Accumulator A	INH	41	O	O		
COMB	$(\overline{B}) \Rightarrow B$ Complement Accumulator B	INH	51	O	O		
CPD #*opr16i*	(A:B) – (M:M+1)	IMM	8C jj kk	PO	OP	– – – –	Δ Δ Δ Δ
CPD *opr8a*	Compare D to Memory (16-Bit)	DIR	9C dd	RPf	RfP		
CPD *opr16a*		EXT	BC hh ll	RPO	ROP		
CPD *oprx0_xysp*		IDX	AC xb	RPf	RfP		
CPD *oprx9,xysp*		IDX1	AC xb ff	RPO	RPO		
CPD *oprx16,xysp*		IDX2	AC xb ee ff	fRPP	fRPP		
CPD [D,*xysp*]		[D,IDX]	AC xb	fIfRPf	fIfRfP		
CPD [*oprx16,xysp*]		[IDX2]	AC xb ee ff	fIPRPf	fIPRfP		
CPS #*opr16i*	(SP) – (M:M+1)	IMM	8F jj kk	PO	OP	– – – –	Δ Δ Δ Δ
CPS *opr8a*	Compare SP to Memory (16-Bit)	DIR	9F dd	RPf	RfP		
CPS *opr16a*		EXT	BF hh ll	RPO	ROP		
CPS *oprx0_xysp*		IDX	AF xb	RPf	RfP		
CPS *oprx9,xysp*		IDX1	AF xb ff	RPO	RPO		
CPS *oprx16,xysp*		IDX2	AF xb ee ff	fRPP	fRPP		
CPS [D,*xysp*]		[D,IDX]	AF xb	fIfRPf	fIfRfP		
CPS [*oprx16,xysp*]		[IDX2]	AF xb ee ff	fIPRPf	fIPRfP		
CPX #*opr16i*	(X) – (M:M+1)	IMM	8E jj kk	PO	OP	– – – –	Δ Δ Δ Δ
CPX *opr8a*	Compare X to Memory (16-Bit)	DIR	9E dd	RPf	RfP		
CPX *opr16a*		EXT	BE hh ll	RPO	ROP		
CPX *oprx0_xysp*		IDX	AE xb	RPf	RfP		
CPX *oprx9,xysp*		IDX1	AE xb ff	RPO	RPO		
CPX *oprx16,xysp*		IDX2	AE xb ee ff	fRPP	fRPP		
CPX [D,*xysp*]		[D,IDX]	AE xb	fIfRPf	fIfRfP		
CPX [*oprx16,xysp*]		[IDX2]	AE xb ee ff	fIPRPf	fIPRfP		
CPY #*opr16i*	(Y) – (M:M+1)	IMM	8D jj kk	PO	OP	– – – –	Δ Δ Δ Δ
CPY *opr8a*	Compare Y to Memory (16-Bit)	DIR	9D dd	RPf	RfP		
CPY *opr16a*		EXT	BD hh ll	RPO	ROP		
CPY *oprx0_xysp*		IDX	AD xb	RPf	RfP		
CPY *oprx9,xysp*		IDX1	AD xb ff	RPO	RPO		
CPY *oprx16,xysp*		IDX2	AD xb ee ff	fRPP	fRPP		
CPY [D,*xysp*]		[D,IDX]	AD xb	fIfRPf	fIfRfP		
CPY [*oprx16,xysp*]		[IDX2]	AD xb ee ff	fIPRPf	fIPRfP		
DAA	Adjust Sum to BCD Decimal Adjust Accumulator A	INH	18 07	OfO	OfO	– – – –	Δ Δ ? Δ
DBEQ *abdxys, rel9*	(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	PPP (branch) PPO (no branch)	PPP	– – – –	– – – –
DBNE *abdxys, rel9*	(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	PPP (branch) PPO (no branch)	PPP	– – – –	– – – –

Table A-1. Instruction Set Summary (Sheet 5 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		

DEC *opr16a*
DEC *oprx0_xysp*
DEC *oprx9,xysp*
DEC *oprx16,xysp*
DEC [D,*xysp*]
DEC [*oprx16,xysp*]
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`
				`rPwO`
				`rPw`
				`rPwO`
				`frPwP`
				`fIfrPw`
				`fIPrPw`
				`O`
				`O`
					`rOPw`
					`rPw`
					`rPOw`
					`frPPw`
					`fIfrPw`
					`fIPrPw`
					`O`
					`O`
						– – – –	Δ Δ Δ –

DES	(SP) – $0001 ⇒ SP *Translates to* LEAS –1,SP	IDX	`1B 9F`	`Pf`	`PP`[1]	– – – –	– – – –
DEX	(X) – $0001 ⇒ X Decrement Index Register X	INH	`09`	`O`	`O`	– – – –	– Δ – –
DEY	(Y) – $0001 ⇒ Y Decrement Index Register Y	INH	`03`	`O`	`O`	– – – –	– Δ – –
EDIV	(Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 by 16 Bit ⇒ 16 Bit Divide (unsigned)	INH	`11`	`ffffffffffO`	`ffffffffffO`	– – – –	Δ Δ Δ Δ
EDIVS	(Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 by 16 Bit ⇒ 16 Bit Divide (signed)	INH	`18 14`	`OffffffffffO`	`OffffffffffO`	– – – –	Δ Δ Δ Δ
EMACS *opr16a* [2]	$(M_{(X)}:M_{(X+1)}) \times (M_{(Y)}:M_{(Y+1)}) + (M\sim M+3) \Rightarrow M\sim M+3$ 16 by 16 Bit ⇒ 32 Bit Multiply and Accumulate (signed)	Special	`18 12 hh ll`	`ORROfffRRfWWP`	`ORROfffRRfWWP`	– – – –	Δ Δ Δ Δ

EMAXD *oprx0_xysp*
EMAXD *oprx9,xysp*
EMAXD *oprx16,xysp*
EMAXD [D,*xysp*]
EMAXD [*oprx16,xysp*]
	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`
				`ORPf`
				`ORPO`
				`OfRPP`
				`OfIfRPf`
				`OfIPRPf`
					`ORfP`
					`ORPO`
					`OfRPP`
					`OfIfRfP`
					`OfIPRfP`
						– – – –	Δ Δ Δ Δ

EMAXM *oprx0_xysp*
EMAXM *oprx9,xysp*
EMAXM *oprx16,xysp*
EMAXM [D,*xysp*]
EMAXM [*oprx16,xysp*]
	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`
				`ORPW`
				`ORPWO`
				`OfRPWP`
				`OfIfRPW`
				`OfIPRPW`
					`ORPW`
					`ORPWO`
					`OfRPWP`
					`OfIfRPW`
					`OfIPRPW`
						– – – –	Δ Δ Δ Δ

EMIND *oprx0_xysp*
EMIND *oprx9,xysp*
EMIND *oprx16,xysp*
EMIND [D,*xysp*]
EMIND [*oprx16,xysp*]
	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`
				`ORPf`
				`ORPO`
				`OfRPP`
				`OfIfRPf`
				`OfIPRPf`
					`ORfP`
					`ORPO`
					`OfRPP`
					`OfIfRfP`
					`OfIPRfP`
						– – – –	Δ Δ Δ Δ

EMINM *oprx0_xysp*
EMINM *oprx9,xysp*
EMINM *oprx16,xysp*
EMINM [D,*xysp*]
EMINM [*oprx16,xysp*]
	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`
				`ORPW`
				`ORPWO`
				`OfRPWP`
				`OfIfRPW`
				`OfIPRPW`
					`ORPW`
					`ORPWO`
					`OfRPWP`
					`OfIfRPW`
					`OfIPRPW`
						– – – –	Δ Δ Δ Δ

EMUL	(D) × (Y) ⇒ Y:D 16 by 16 Bit Multiply (unsigned)	INH	`13`	`ffO`	`ffO`	– – – –	Δ Δ – Δ

EMULS	(D) × (Y) ⇒ Y:D
	16 by 16 Bit Multiply (signed)
		INH	`18 13`	`OfO`
				(if followed by page 2 instruction)
				`OffO`
					`OfO`
					
					`OfO`
						– – – –	Δ Δ – Δ

EORA #*opr8i*
EORA *opr8a*
EORA *opr16a*
EORA *oprx0_xysp*
EORA *oprx9,xysp*
EORA *oprx16,xysp*
EORA [D,*xysp*]
EORA [*oprx16,xysp*]
	(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`
				`P`
				`rPf`
				`rPO`
				`rPf`
				`rPO`
				`frPP`
				`fIfrPf`
				`fIPrPf`
					`P`
					`rfP`
					`rOP`
					`rfP`
					`rPO`
					`frPP`
					`fIfrfP`
					`fIPrfP`
						– – – –	Δ Δ 0 –

Notes:

1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.
2. *opr16a* is an extended address specification. Both X and Y point to source operands.

EORB #*opr8i*
EORB *opr8a*
EORB *opr16a*
EORB *oprx0_xysp*
EORB *oprx9,xysp*
EORB *oprx16,xysp*
EORB [D,*xysp*]
EORB [*oprx16,xysp*]
	(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`
				`P`
				`rPf`
				`rPO`
				`rPf`
				`rPO`
				`frPP`
				`fIfrPf`
				`fIPrPf`
					`P`
					`rfP`
					`rOP`
					`rfP`
					`rPO`
					`frPP`
					`fIfrfP`
					`fIPrfP`
						– – – –	Δ Δ 0 –

Table A-1. Instruction Set Summary (Sheet 6 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
ETBL *oprx0_xysp*	(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 or extensions allowed)	IDX	`18 3F xb`	`ORRffffffP`	`ORRffffffP`	– – – – C Bit is undefined in HC12	Δ Δ – Δ ?
EXG *abcdxys,abcdxys*	(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`	`P`	`P`	– – – –	– – – –
FDIV	(D) ÷ (X) ⇒ X; Remainder ⇒ D 16 by 16 Bit Fractional Divide	INH	`18 11`	`OffffffffffO`	`OffffffffffO`	– – – –	– Δ Δ Δ

IBEQ *abdxys, rel9*	(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`	`PPP (branch)`
				`PPO (no branch)`
					`PPP`	– – – –	– – – –

IBNE *abdxys, rel9*	(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`	`PPP (branch)`
				`PPO (no branch)`
					`PPP`	– – – –	– – – –

IDIV	(D) ÷ (X) ⇒ X; Remainder ⇒ D 16 by 16 Bit Integer Divide (unsigned)	INH	`18 10`	`OffffffffffO`	`OffffffffffO`	– – – –	– Δ 0 Δ
IDIVS	(D) ÷ (X) ⇒ X; Remainder ⇒ D 16 by 16 Bit Integer Divide (signed)	INH	`18 15`	`OffffffffffO`	`OffffffffffO`	– – – –	Δ Δ Δ Δ

INC *opr16a*
INC *oprx0_xysp*
INC *oprx9,xysp*
INC *oprx16,xysp*
INC [D,*xysp*]
INC [*oprx16,xysp*]
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`
				`rPwO`
				`rPw`
				`rPwO`
				`frPwP`
				`fIfrPw`
				`fIPrPw`
				`O`
				`O`
					`rOPw`
					`rPw`
					`rPOw`
					`frPPw`
					`fIfrPw`
					`fIPrPw`
					`O`
					`O`
						– – – –	Δ Δ Δ –

INS	(SP) + $0001 ⇒ SP *Translates to* LEAS 1,SP	IDX	`1B 81`	`Pf`	`PP`[1]	– – – –	– – – –
INX	(X) + $0001 ⇒ X Increment Index Register X	INH	`08`	`O`	`O`	– – – –	– Δ – –
INY	(Y) + $0001 ⇒ Y Increment Index Register Y	INH	`02`	`O`	`O`	– – – –	– Δ – –

JMP *opr16a*
JMP *oprx0_xysp*
JMP *oprx9,xysp*
JMP *oprx16,xysp*
JMP [D,*xysp*]
JMP [*oprx16,xysp*]
	Routine 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`
				`PPP`
				`PPP`
				`PPP`
				`fPPP`
				`fIfPPP`
				`fIfPPP`
					`PPP`
					`PPP`
					`PPP`
					`fPPP`
					`fIfPPP`
					`fIfPPP`
						– – – –	– – – –

JSR *opr8a*
JSR *opr16a*
JSR *oprx0_xysp*
JSR *oprx9,xysp*
JSR *oprx16,xysp*
JSR [D,*xysp*]
JSR [*oprx16,xysp*]
	(SP) – 2 ⇒ SP;
	RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$;
	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`
				`SPPP`
				`SPPP`
				`PPPS`
				`PPPS`
				`fPPPS`
				`fIfPPPS`
				`fIfPPPS`
					`PPPS`
					`PPPS`
					`PPPS`
					`PPPS`
					`fPPPS`
					`fIfPPPS`
					`fIfPPPS`
						– – – –	– – – –

LBCC *rel16*	Long Branch if Carry Clear (if C = 0)	REL	`18 24 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]	– – – –	– – – –
LBCS *rel16*	Long Branch if Carry Set (if C = 1)	REL	`18 25 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]	– – – –	– – – –
LBEQ *rel16*	Long Branch if Equal (if Z = 1)	REL	`18 27 qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]	– – – –	– – – –
LBGE *rel16*	Long Branch Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	`18 2C qq rr`	`OPPP/OPO`[1]	`OPPP/OPO`[1]	– – – –	– – – –

Note 1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.

Table A-1. Instruction Set Summary (Sheet 7 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12	 M68HC12	S X H I	N Z V C
LBGT *rel16*	Long Branch if Greater Than (if Z + (N ⊕ V) = 0) (signed)	REL	`18 2E qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBHI *rel16*	Long Branch if Higher (if C + Z = 0) (unsigned)	REL	`18 22 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBHS *rel16*	Long Branch if Higher or Same (if C = 0) (unsigned) same function as LBCC	REL	`18 24 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBLE *rel16*	Long Branch if Less Than or Equal (if Z + (N ⊕ V) = 1) (signed)	REL	`18 2F qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBLO *rel16*	Long Branch if Lower (if C = 1) (unsigned) *same function as LBC*S	REL	`18 25 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBLS *rel16*	Long Branch if Lower or Same (if C + Z = 1) (unsigned)	REL	`18 23 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBLT *rel16*	Long Branch if Less Than (if N ⊕ V = 1) (signed)	REL	`18 2D qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBMI *rel16*	Long Branch if Minus (if N = 1)	REL	`18 2B qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBNE *rel16*	Long Branch if Not Equal (if Z = 0)	REL	`18 26 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBPL *rel16*	Long Branch if Plus (if N = 0)	REL	`18 2A qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBRA *rel16*	Long Branch Always (if 1=1)	REL	`18 20 qq rr`	OPPP	OPPP	– – – –	– – – –
LBRN *rel16*	Long Branch Never (if 1 = 0)	REL	`18 21 qq rr`	OPO	OPO	– – – –	– – – –
LBVC *rel16*	Long Branch if Overflow Bit Clear (if V=0)	REL	`18 28 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –
LBVS *rel16*	Long Branch if Overflow Bit Set (if V = 1)	REL	`18 29 qq rr`	OPPP/OPO[1]	OPPP/OPO[1]	– – – –	– – – –

LDAA #*opr8i*
LDAA *opr8a*
LDAA *opr16a*
LDAA *oprx0_xysp*
LDAA *oprx9,xysp*
LDAA *oprx16,xysp*
LDAA [D,*xysp*]
LDAA [*oprx16,xysp*]
	(M) ⇒ A
	Load Accumulator A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`86 ii`
			`96 dd`
			`B6 hh ll`
			`A6 xb`
			`A6 xb ff`
			`A6 xb ee ff`
			`A6 xb`
			`A6 xb ee ff`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ 0 –

LDAB #*opr8i*
LDAB *opr8a*
LDAB *opr16a*
LDAB *oprx0_xysp*
LDAB *oprx9,xysp*
LDAB *oprx16,xysp*
LDAB [D,*xysp*]
LDAB [*oprx16,xysp*]
	(M) ⇒ B
	Load Accumulator B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`C6 ii`
			`D6 dd`
			`F6 hh ll`
			`E6 xb`
			`E6 xb ff`
			`E6 xb ee ff`
			`E6 xb`
			`E6 xb ee ff`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ 0 –

LDD #*opr16i*
LDD *opr8a*
LDD *opr16a*
LDD *oprx0_xysp*
LDD *oprx9,xysp*
LDD *oprx16,xysp*
LDD [D,*xysp*]
LDD [*oprx16,xysp*]
	(M:M+1) ⇒ A:B
	Load Double Accumulator D (A:B)
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`CC jj kk`
			`DC dd`
			`FC hh ll`
			`EC xb`
			`EC xb ff`
			`EC xb ee ff`
			`EC xb`
			`EC xb ee ff`
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					OP
					RfP
					ROP
					RfP
					RPO
					fRPP
					fIfRfP
					fIPRfP
						– – – –	Δ Δ 0 –

Note 1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.							

LDS #*opr16i*
LDS *opr8a*
LDS *opr16a*
LDS *oprx0_xysp*
LDS *oprx9,xysp*
LDS *oprx16,xysp*
LDS [D,*xysp*]
LDS [*oprx16,xysp*]
	(M:M+1) ⇒ SP
	Load Stack Pointer
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`CF jj kk`
			`DF dd`
			`FF hh ll`
			`EF xb`
			`EF xb ff`
			`EF xb ee ff`
			`EF xb`
			`EF xb ee ff`
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					OP
					RfP
					ROP
					RfP
					RPO
					fRPP
					fIfRfP
					fIPRfP
						– – – –	Δ Δ 0 –

LDX #*opr16i*
LDX *opr8a*
LDX *opr16a*
LDX *oprx0_xysp*
LDX *oprx9,xysp*
LDX *oprx16,xysp*
LDX [D,*xysp*]
LDX [*oprx16,xysp*]
	(M:M+1) ⇒ X
	Load Index Register X
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`CE jj kk`
			`DE dd`
			`FE hh ll`
			`EE xb`
			`EE xb ff`
			`EE xb ee ff`
			`EE xb`
			`EE xb ee ff`
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					OP
					RfP
					ROP
					RfP
					RPO
					fRPP
					fIfRfP
					fIPRfP
						– – – –	Δ Δ 0 –

Table A-1. Instruction Set Summary (Sheet 8 of 14)

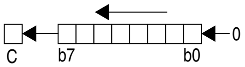
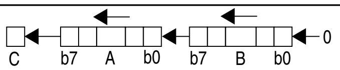
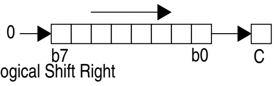
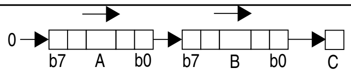
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
LDY #*opr16i*	(M:M+1) ⇒ Y	IMM	CD jj kk	PO	OP	––––	ΔΔ0–
LDY *opr8a*	Load Index Register Y	DIR	DD dd	RPf	RfP		
LDY *opr16a*		EXT	FD hh ll	RPO	ROP		
LDY *oprx0_xysp*		IDX	ED xb	RPf	RfP		
LDY *oprx9,xysp*		IDX1	ED xb ff	RPO	RPO		
LDY *oprx16,xysp*		IDX2	ED xb ee ff	fRPP	fRPP		
LDY [D,*xysp*]		[D,IDX]	ED xb	fIfRPf	fIfRfP		
LDY [*oprx16,xysp*]		[IDX2]	ED xb ee ff	fIPRPf	fIPRfP		
LEAS *oprx0_xysp*	Effective Address ⇒ SP	IDX	1B xb	Pf	PP[1]	––––	––––
LEAS *oprx9,xysp*	Load Effective Address into SP	IDX1	1B xb ff	PO	PO		
LEAS *oprx16,xysp*		IDX2	1B xb ee ff	PP	PP		
LEAX *oprx0_xysp*	Effective Address ⇒ X	IDX	1A xb	Pf	PP[1]	––––	––––
LEAX *oprx9,xysp*	Load Effective Address into X	IDX1	1A xb ff	PO	PO		
LEAX *oprx16,xysp*		IDX2	1A xb ee ff	PP	PP		
LEAY *oprx0_xysp*	Effective Address ⇒ Y	IDX	19 xb	Pf	PP[1]	––––	––––
LEAY *oprx9,xysp*	Load Effective Address into Y	IDX1	19 xb ff	PO	PO		
LEAY *oprx16,xysp*		IDX2	19 xb ee ff	PP	PP		
LSL *opr16a*		EXT	78 hh ll	rPwO	rOPw	––––	ΔΔΔΔ
LSL *oprx0_xysp*		IDX	68 xb	rPw	rPw		
LSL *oprx9,xysp*		IDX1	68 xb ff	rPwO	rPOw		
LSL *oprx16,xysp*	Logical Shift Left	IDX2	68 xb ee ff	frPPw	frPPw		
LSL [D,*xysp*]	same function as ASL	[D,IDX]	68 xb	fIfrPw	fIfrPw		
LSL [*oprx16,xysp*]		[IDX2]	68 xb ee ff	fIPrPw	fIPrPw		
LSLA	Logical Shift Accumulator A to Left	INH	48	O	O		
LSLB	Logical Shift Accumulator B to Left	INH	58	O	O		
LSLD		INH	59	O	O	––––	ΔΔΔΔ
	Logical Shift Left D Accumulator						
	same function as ASLD						
LSR *opr16a*		EXT	74 hh ll	rPwO	rOPw	––––	0ΔΔΔ
LSR *oprx0_xysp*		IDX	64 xb	rPw	rPw		
LSR *oprx9,xysp*		IDX1	64 xb ff	rPwO	rPOw		
LSR *oprx16,xysp*	Logical Shift Right	IDX2	64 xb ee ff	frPwP	frPPw		
LSR [D,*xysp*]		[D,IDX]	64 xb	fIfrPw	fIfrPw		
LSR [*oprx16,xysp*]		[IDX2]	64 xb ee ff	fIPrPw	fIPrPw		
LSRA	Logical Shift Accumulator A to Right	INH	44	O	O		
LSRB	Logical Shift Accumulator B to Right	INH	54	O	O		
LSRD		INH	49	O	O	––––	0ΔΔΔ
	Logical Shift Right D Accumulator						
MAXA *oprx0_xysp*	MAX((A), (M)) ⇒ A	IDX	18 18 xb	OrPf	OrfP	––––	ΔΔΔΔ
MAXA *oprx9,xysp*	MAX of 2 Unsigned 8-Bit Values	IDX1	18 18 xb ff	OrPO	OrPO		
MAXA *oprx16,xysp*		IDX2	18 18 xb ee ff	OfrPP	OfrPP		
MAXA [D,*xysp*]	N, Z, V and C status bits reflect result of	[D,IDX]	18 18 xb	OfIfrPf	OfIfrfP		
MAXA [*oprx16,xysp*]	internal compare ((A) – (M)).	[IDX2]	18 18 xb ee ff	OfIPrPf	OfIPrfP		
Note 1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.							
MAXM *oprx0_xysp*	MAX((A), (M)) ⇒ M	IDX	18 1C xb	OrPw	OrPw	––––	ΔΔΔΔ
MAXM *oprx9,xysp*	MAX of 2 Unsigned 8-Bit Values	IDX1	18 1C xb ff	OrPwO	OrPwO		
MAXM *oprx16,xysp*		IDX2	18 1C xb ee ff	OfrPwP	OfrPwP		
MAXM [D,*xysp*]	N, Z, V and C status bits reflect result of	[D,IDX]	18 1C xb	OfIfrPw	OfIfrPw		
MAXM [*oprx16,xysp*]	internal compare ((A) – (M)).	[IDX2]	18 1C xb ee ff	OfIPrPw	OfIPrPw		
MEM	μ (grade) ⇒ $M_{(Y)}$; (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 *CPU12 Reference Manual* for special cases.	Special	01	RRfOw	RRfOw	––?–	????

Table A-1. Instruction Set Summary (Sheet 9 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		

MINA *oprx0_xysp*
MINA *oprx9,xysp*
MINA *oprx16,xysp*
MINA [D,*xysp*]
MINA [*oprx16,xysp*]
	MIN((A), (M)) ⇒ A
	MIN of 2 Unsigned 8-Bit Values
	
	N, Z, V and C status bits reflect result of internal compare ((A) – (M)).
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`18 19 xb`
			`18 19 xb ff`
			`18 19 xb ee ff`
			`18 19 xb`
			`18 19 xb ee ff`
				OrPf
				OrPO
				OfrPP
				OfIfrPf
				OfIPrPf
					OrfP
					OrPO
					OfrPP
					OfIfrfP
					OfIPrfP
						– – – –	Δ Δ Δ Δ

MINM *oprx0_xysp*
MINM *oprx9,xysp*
MINM *oprx16,xysp*
MINM [D,*xysp*]
MINM [*oprx16,xysp*]
	MIN((A), (M)) ⇒ M
	MIN of 2 Unsigned 8-Bit Values
	
	N, Z, V and C status bits reflect result of internal compare ((A) – (M)).
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`18 1D xb`
			`18 1D xb ff`
			`18 1D xb ee ff`
			`18 1D xb`
			`18 1D xb ee ff`
				OrPw
				OrPwO
				OfrPwP
				OfIfrPw
				OfIPrPw
					OrPw
					OrPwO
					OfrPwP
					OfIfrPw
					OfIPrPw
						– – – –	Δ Δ Δ Δ

MOVB #*opr8, opr16a*[1]
MOVB #*opr8i, oprx0_xysp*[1]
MOVB *opr16a, opr16a*[1]
MOVB *opr16a, oprx0_xysp*[1]
MOVB *oprx0_xysp, opr16a*[1]
MOVB *oprx0_xysp, oprx0_xysp*[1]
	$(M_1) \Rightarrow M_2$
	Memory to Memory Byte-Move (8-Bit)
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			`18 0B ii hh ll`
			`18 08 xb ii`
			`18 0C hh ll hh ll`
			`18 09 xb hh ll`
			`18 0D xb hh ll`
			`18 0A xb xb`
				OPwP
				OPwO
				OrPwPO
				OPrPw
				OrPwP
				OrPwO
					OPwP
					OPwO
					OrPwPO
					OPrPw
					OrPwP
					OrPwO
						– – – –	– – – –

MOVW #*oprx16, opr16a*[1]
MOVW #*opr16i, oprx0_xysp*[1]
MOVW *opr16a, opr16a*[1]
MOVW *opr16a, oprx0_xysp*[1]
MOVW *oprx0_xysp, opr16a*[1]
MOVW *oprx0_xysp, oprx0_xysp*[1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit)
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			`18 03 jj kk hh ll`
			`18 00 xb jj kk`
			`18 04 hh ll hh ll`
			`18 01 xb hh ll`
			`18 05 xb hh ll`
			`18 02 xb xb`
				OPWPO
				OPPW
				ORPWPO
				OPRPW
				ORPWP
				ORPWO
					OPWPO
					OPPW
					ORPWPO
					OPRPW
					ORPWP
					ORPWO
						– – – –	– – – –

MUL	(A) × (B) ⇒ A:B 8 by 8 Unsigned Multiply	INH	`12`	O	ffO	– – – –	– – – Δ

NEG *opr16a*
NEG *oprx0_xysp*
NEG *oprx9,xysp*
NEG *oprx16,xysp*
NEG [D,*xysp*]
NEG [*oprx16,xysp*]
NEGA
 	
NEGB
	0 – (M) ⇒ M *equivalent to* ($\overline{M}$) + 1 ⇒ M
	Two's Complement Negate
	
	0 – (A) ⇒ A *equivalent to* ($\overline{A}$) + 1 ⇒ A
	Negate Accumulator A
	0 – (B) ⇒ B *equivalent to* ($\overline{B}$) + 1 ⇒ B
	Negate Accumulator B
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
		INH
		
		INH
			`70 hh ll`
			`60 xb`
			`60 xb ff`
			`60 xb ee ff`
			`60 xb`
			`60 xb ee ff`
			`40`
			
			`50`
				rPwO
				rPw
				rPwO
				frPwP
				fIfrPw
				fIPrPw
				O
				
				O
					rOPw
					rPw
					rPOw
					frPPw
					fIfrPw
					fIPrPw
					O
					
					O
						– – – –	Δ Δ Δ Δ

NOP	No Operation	INH	`A7`	O	O	– – – –	– – – –

ORAA #*opr8i*
ORAA *opr8a*
ORAA *opr16a*
ORAA *oprx0_xysp*
ORAA *oprx9,xysp*
ORAA *oprx16,xysp*
ORAA [D,*xysp*]
ORAA [*oprx16,xysp*]
	(A) + (M) ⇒ A
	Logical OR A with Memory
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`8A ii`
			`9A dd`
			`BA hh ll`
			`AA xb`
			`AA xb ff`
			`AA xb ee ff`
			`AA xb`
			`AA xb ee ff`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ 0 –

ORAB #*opr8i*
ORAB *opr8a*
ORAB *opr16a*
ORAB *oprx0_xysp*
ORAB *oprx9,xysp*
ORAB *oprx16,xysp*
ORAB [D,*xysp*]
ORAB [*oprx16,xysp*]
	(B) + (M) ⇒ B
	Logical OR B with Memory
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			`CA ii`
			`DA dd`
			`FA hh ll`
			`EA xb`
			`EA xb ff`
			`EA xb ee ff`
			`EA xb`
			`EA xb ee ff`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ 0 –

ORCC #*opr8i*	(CCR) + M ⇒ CCR Logical OR CCR with Memory	IMM	`14 ii`	P	P	⇑ – ⇑ ⇑	⇑ ⇑ ⇑ ⇑
PSHA	(SP) – 1 ⇒ SP; (A) ⇒ $M_{(SP)}$ Push Accumulator A onto Stack	INH	`36`	Os	Os	– – – –	– – – –
PSHB	(SP) – 1 ⇒ SP; (B) ⇒ $M_{(SP)}$ Push Accumulator B onto Stack	INH	`37`	Os	Os	– – – –	– – – –
PSHC	(SP) – 1 ⇒ SP; (CCR) ⇒ $M_{(SP)}$ Push CCR onto Stack	INH	`39`	Os	Os	– – – –	– – – –
PSHD	(SP) – 2 ⇒ SP; (A:B) ⇒ $M_{(SP)}{:}M_{(SP+1)}$ Push D Accumulator onto Stack	INH	`3B`	OS	OS	– – – –	– – – –
PSHX	(SP) – 2 ⇒ SP; ($X_H{:}X_L$) ⇒ $M_{(SP)}{:}M_{(SP+1)}$ Push Index Register X onto Stack	INH	`34`	OS	OS	– – – –	– – – –
PSHY	(SP) – 2 ⇒ SP; ($Y_H{:}Y_L$) ⇒ $M_{(SP)}{:}M_{(SP+1)}$ Push Index Register Y onto Stack	INH	`35`	OS	OS	– – – –	– – – –

Note 1. The first operand in the source code statement specifies the source for the move.

Table A-1. Instruction Set Summary (Sheet 10 of 14)

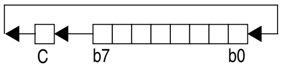
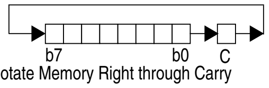
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
PULA	$(M_{(SP)}) \Rightarrow A$; $(SP) + 1 \Rightarrow SP$ Pull Accumulator A from Stack	INH	32	ufO	ufO	– – – –	– – – –
PULB	$(M_{(SP)}) \Rightarrow B$; $(SP) + 1 \Rightarrow SP$ Pull Accumulator B from Stack	INH	33	ufO	ufO	– – – –	– – – –
PULC	$(M_{(SP)}) \Rightarrow CCR$; $(SP) + 1 \Rightarrow SP$ Pull CCR from Stack	INH	38	ufO	ufO	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	UfO	UfO	– – – –	– – – –
PULX	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow X_H:X_L$; $(SP) + 2 \Rightarrow SP$ Pull Index Register X from Stack	INH	30	UfO	UfO	– – – –	– – – –
PULY	$(M_{(SP)}:M_{(SP+1)}) \Rightarrow Y_H:Y_L$; $(SP) + 2 \Rightarrow SP$ Pull Index Register Y from Stack	INH	31	UfO	UfO	– – – –	– – – –
REV	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	Orf(t,tx)O (exit + re-entry replaces comma above if interrupted) ff + Orf(t,	Orf(t,tx)O ff + Orf(t,	– – ? –	? ? Δ ?
REVW	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	ORf(t,Tx)O (loop to read weight if enabled) (r,RfRf) (exit + re-entry replaces comma above if interrupted) ffff + ORf(t,	ORf(t,Tx)O (r,RfRf) fff + ORf(t,	– – ? –	? ? Δ !
ROL *opr16a* ROL *oprx0_xysp* ROL *oprx9,xysp* ROL *oprx16,xysp* ROL [D,*xysp*] ROL [*oprx16,xysp*] 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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	– – – –	Δ Δ Δ Δ
ROR *opr16a* ROR *oprx0_xysp* ROR *oprx9,xysp* ROR *oprx16,xysp* ROR [D,*xysp*] ROR [*oprx16,xysp*] 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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	– – – –	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	uUnfPPP	uUnPPP	– – – –	– – – –
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	uUUUUPPP (with interrupt pending) uUUUUVfPPP	uUUUUPPP uUUUUfVfPPP	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	UfPPP	UfPPP	– – – –	– – – –

Table A-1. Instruction Set Summary (Sheet 11 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
SBA	(A) – (B) ⇒ A Subtract B from A	INH	18 16	OO	OO	– – – –	Δ Δ Δ Δ
SBCA #*opr8i* SBCA *opr8a* SBCA *opr16a* SBCA *oprx0_xysp* SBCA *oprx9,xysp* SBCA *oprx16,xysp* SBCA [D,*xysp*] SBCA [*oprx16,xysp*]	(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	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	P rfP rOP rfP rPO frPP fIfrfP fIPrfP	– – – –	Δ Δ Δ Δ
SBCB #*opr8i* SBCB *opr8a* SBCB *opr16a* SBCB *oprx0_xysp* SBCB *oprx9,xysp* SBCB *oprx16,xysp* SBCB [D,*xysp*] SBCB [*oprx16,xysp*]	(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	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	P rfP rOP rfP rPO frPP fIfrfP fIPrfP	– – – –	Δ Δ Δ Δ
SEC	1 ⇒ C *Translates to* ORCC #$01	IMM	14 01	P	P	– – – –	– – – 1
SEI	1 ⇒ I; (inhibit I interrupts) *Translates to* ORCC #$10	IMM	14 10	P	P	– – – 1	– – – –
SEV	1 ⇒ V *Translates to* ORCC #$02	IMM	14 02	P	P	– – – –	– – 1 –
SEX *abc,dxys*	$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	P	P	– – – –	– – – –
STAA *opr8a* STAA *opr16a* STAA *oprx0_xysp* STAA *oprx9,xysp* STAA *oprx16,xysp* STAA [D,*xysp*] STAA [*oprx16,xysp*]	(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	Pw PwO Pw PwO PwP PIfw PIPw	Pw wOP Pw PwO PwP PIfPw PIPPw	– – – –	Δ Δ 0 –
STAB *opr8a* STAB *opr16a* STAB *oprx0_xysp* STAB *oprx9,xysp* STAB *oprx16,xysp* STAB [D,*xysp*] STAB [*oprx16,xysp*]	(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	Pw PwO Pw PwO PwP PIfw PIPw	Pw wOP Pw PwO PwP PIfPw PIPPw	– – – –	Δ Δ 0 –
STD *opr8a* STD *opr16a* STD *oprx0_xysp* STD *oprx9,xysp* STD *oprx16,xysp* STD [D,*xysp*] STD [*oprx16,xysp*]	(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	PW PWO PW PWO PWP PIfW PIPW	PW WOP PW PWO PWP PIfPW PIPPW	– – – –	Δ Δ 0 –
STOP	(SP) – 2 ⇒ SP; $\mathrm{RTN_H{:}RTN_L} \Rightarrow \mathrm{M_{(SP)}{:}M_{(SP+1)}}$; (SP) – 2 ⇒ SP; $\mathrm{(Y_H{:}Y_L)} \Rightarrow \mathrm{M_{(SP)}{:}M_{(SP+1)}}$; (SP) – 2 ⇒ SP; $\mathrm{(X_H{:}X_L)} \Rightarrow \mathrm{M_{(SP)}{:}M_{(SP+1)}}$; (SP) – 2 ⇒ SP; (B:A) ⇒ $\mathrm{M_{(SP)}{:}M_{(SP+1)}}$; (SP) – 1 ⇒ SP; (CCR) ⇒ $\mathrm{M_{(SP)}}$; STOP All Clocks Registers stacked to allow quicker recovery by interrupt. If S control bit = 1, the STOP instruction is disabled and acts like a two-cycle NOP.	INH	18 3E	(entering STOP) OOSSSSsf (exiting STOP) fVfPPP (continue) ff (if STOP disabled) OO	 OOSSSfSs fVfPPP fO OO	– – – –	– – – –

Table A-1. Instruction Set Summary (Sheet 12 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		

STS *opr8a*
STS *opr16a*
STS *oprx0_xysp*
STS *oprx9,xysp*
STS *oprx16,xysp*
STS [D,*xysp*]
STS [*oprx16,xysp*]
	(SP$_H$:SP$_L$) ⇒ 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`
				PW
				PWO
				PW
				PWO
				PWP
				PIfW
				PIPW
					PW
					WOP
					PW
					PWO
					PWP
					PIfPW
					PIPPW
						– – – –	Δ Δ 0 –

STX *opr8a*
STX *opr16a*
STX *oprx0_xysp*
STX *oprx9,xysp*
STX *oprx16,xysp*
STX [D,*xysp*]
STX [*oprx16,xysp*]
	(X$_H$:X$_L$) ⇒ 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`
				PW
				PWO
				PW
				PWO
				PWP
				PIfW
				PIPW
					PW
					WOP
					PW
					PWO
					PWP
					PIfPW
					PIPPW
						– – – –	Δ Δ 0 –

STY *opr8a*
STY *opr16a*
STY *oprx0_xysp*
STY *oprx9,xysp*
STY *oprx16,xysp*
STY [D,*xysp*]
STY [*oprx16,xysp*]
	(Y$_H$:Y$_L$) ⇒ 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`
				PW
				PWO
				PW
				PWO
				PWP
				PIfW
				PIPW
					PW
					WOP
					PW
					PWO
					PWP
					PIfPW
					PIPPW
						– – – –	Δ Δ 0 –

SUBA #*opr8i*
SUBA *opr8a*
SUBA *opr16a*
SUBA *oprx0_xysp*
SUBA *oprx9,xysp*
SUBA *oprx16,xysp*
SUBA [D,*xysp*]
SUBA [*oprx16,xysp*]
	(A) – (M) ⇒ 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`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ Δ Δ

SUBB #*opr8i*
SUBB *opr8a*
SUBB *opr16a*
SUBB *oprx0_xysp*
SUBB *oprx9,xysp*
SUBB *oprx16,xysp*
SUBB [D,*xysp*]
SUBB [*oprx16,xysp*]
	(B) – (M) ⇒ 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`
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rfP
					rOP
					rfP
					rPO
					frPP
					fIfrfP
					fIPrfP
						– – – –	Δ Δ Δ Δ

SUBD #*opr16i*
SUBD *opr8a*
SUBD *opr16a*
SUBD *oprx0_xysp*
SUBD *oprx9,xysp*
SUBD *oprx16,xysp*
SUBD [D,*xysp*]
SUBD [*oprx16,xysp*]
	(D) – (M:M+1) ⇒ 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`
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					OP
					RfP
					ROP
					RfP
					RPO
					fRPP
					fIfRfP
					fIPRfP
						– – – –	Δ Δ Δ Δ

SWI	(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$_{(SP)}$ 1 ⇒ I; (SWI Vector) ⇒ PC Software Interrupt	INH	`3F`	VSPSSPSsP* (for Reset) VfPPP	VSPSSPSsP* VfPPP	– – – 1 1 1 – 1	– – – – – – – –
*The CPU also uses the SWI microcode sequence for hardware interrupts and unimplemented opcode traps. Reset uses the `VfPPP` variation of this sequence.							
TAB	(A) ⇒ B Transfer A to B	INH	`18 0E`	OO	OO	– – – –	Δ Δ 0 –
TAP	(A) ⇒ CCR *Translates to* TFR A , CCR	INH	`B7 02`	P	P	Δ ⇓ Δ Δ	Δ Δ Δ Δ
TBA	(B) ⇒ A Transfer B to A	INH	`18 0F`	OO	OO	– – – –	Δ Δ 0 –
TBEQ *abdxys,rel9*	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`	PPP (branch) PPO (no branch)	PPP	– – – –	– – – –

Table A-1. Instruction Set Summary (Sheet 13 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
				HCS12	M68HC12		
TBL *oprx0_xysp*	(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 or extensions allowed)	IDX	`18 3D xb`	`ORfffP`	`OrrffffP`	– – – –	Δ Δ – Δ ? C Bit is undefined in HC12

TBNE *abdxys,rel9*	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`	`PPP (branch)`
				`PPO (no branch)`
					`PPP`	– – – –	– – – –

TFR *abcdxys,abcdxys*	(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`	`P`	`P`	– – – – or Δ ⇓ Δ Δ	– – – – Δ Δ Δ Δ
TPA	(CCR) ⇒ A *Translates to* TFR CCR ,A	INH	`B7 20`	`P`	`P`	– – – –	– – – –

TRAP *trapnum*	(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$_{(SP)}$
	1 ⇒ I; (TRAP Vector) ⇒ PC
	
	Unimplemented opcode trap
		INH	`18 tn`
			`tn = $30-$39`
			`or`
			`$40-$FF`
				`OVSPSSPSsP`	`OfVSPSSPSsP`	– – – 1	– – – –

TST *opr16a*	(M) – 0	EXT	`F7 hh ll`	`rPO`	`rOP`	– – – –	Δ Δ 0 0
TST *oprx0_xysp*	Test Memory for Zero or Minus	IDX	`E7 xb`	`rPf`	`rfP`		
TST *oprx9,xysp*		IDX1	`E7 xb ff`	`rPO`	`rPO`		
TST *oprx16,xysp*		IDX2	`E7 xb ee ff`	`frPP`	`frPP`		
TST [D,*xysp*]		[D,IDX]	`E7 xb`	`fIfrPf`	`fIfrfP`		
TST [*oprx16,xysp*]		[IDX2]	`E7 xb ee ff`	`fIPrPf`	`fIPrfP`		
TSTA	(A) – 0 Test A for Zero or Minus	INH	`97`	`O`	`O`		
TSTB	(B) – 0 Test B for Zero or Minus	INH	`D7`	`O`	`O`		
TSX	(SP) ⇒ X *Translates to* TFR SP,X	INH	`B7 75`	`P`	`P`	– – – –	– – – –
TSY	(SP) ⇒ Y *Translates to* TFR SP,Y	INH	`B7 76`	`P`	`P`	– – – –	– – – –
TXS	(X) ⇒ SP *Translates to* TFR X,SP	INH	`B7 57`	`P`	`P`	– – – –	– – – –
TYS	(Y) ⇒ SP *Translates to* TFR Y,SP	INH	`B7 67`	`P`	`P`	– – – –	– – – –

WAI	(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`	`OSSSSsf`
				(after interrupt)
				`fVfPPP`
					`OSSSfSsf`
					
					`VfPPP`
						– – – –
						or
						– – – 1
						or
						– 1 – 1
							– – – –
							
							– – – –
							
							– – – –

Table A-1. Instruction Set Summary (Sheet 14 of 14)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12	 M68HC12	S X H I	N Z V C

WAV	$\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ and $\sum_{i=1}^{B} F_i \Rightarrow X$
	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, six extra bytes of stack used for intermediate values
		Special	`18 3C`	`Of(frr,ffff)O`
				(add if interrupt)
				`SSS + UUUrr,`
					`Off(frr,fffff)O`
					
					`SSSf + UUUrr`
						– – ? –	? Δ ? ?

wavr
pseudo-instruction
	see WAV
	Resume executing an interrupted WAV instruction (recover intermediate results from stack rather than initializing them to zero)
		Special	`3C`	`UUUrr,ffff`
				`(frr,ffff)O`
				(exit + re-entry replaces comma above if interrupted)
				`SSS + UUUrr,`
					`UUUrrfffff`
					`(frr,fffff)O`
					
					`SSSf + UUUrr`
						– – ? –	? Δ ? ?

XGDX	(D) ⇔ (X) *Translates to* EXG D, X	INH	`B7 C5`	`P`	`P`	– – – –	– – – –
XGDY	(D) ⇔ (Y) *Translates to* EXG D, Y	INH	`B7 C6`	`P`	`P`	– – – –	– – – –

Table A-2. CPU12 Opcode Map (Sheet 1 of 2)

00 †5 BGND IH 1	10 1 ANDCC IM 2	20 3 BRA RL 2	30 3 PULX IH 1	40 1 NEGA IH 1	50 1 NEGB IH 1	60 3-6 NEG ID 2-4	70 4 NEG EX 3	80 1 SUBA IM 2	90 3 SUBA DI 2	A0 3-6 SUBA ID 2-4	B0 3 SUBA EX 3	C0 1 SUBB IM 2	D0 3 SUBB DI 2	E0 3-6 SUBB ID 2-4	F0 3 SUBB EX 3
01 5 MEM IH 1	11 11 EDIV IH 1	21 1 BRN RL 2	31 3 PULY IH 1	41 1 COMA IH 1	51 1 COMB IH 1	61 3-6 COM ID 2-4	71 4 COM EX 3	81 1 CMPA IM 2	91 3 CMPA DI 2	A1 3-6 CMPA ID 2-4	B1 3 CMPA EX 3	C1 1 CMPB IM 2	D1 3 CMPB DI 2	E1 3-6 CMPB ID 2-4	F1 3 CMPB EX 3
02 1 INY IH 1	12 ‡1 MUL IH 1	22 3/1 BHI RL 2	32 3 PULA IH 1	42 1 INCA IH 1	52 1 INCB IH 1	62 3-6 INC ID 2-4	72 4 INC EX 3	82 1 SBCA IM 2	92 3 SBCA DI 2	A2 3-6 SBCA ID 2-4	B2 3 SBCA EX 3	C2 1 SBCB IM 2	D2 3 SBCB DI 2	E2 3-6 SBCB ID 2-4	F2 3 SBCB EX 3
03 1 DEY IH 1	13 3 EMUL IH 1	23 3/1 BLS RL 2	33 3 PULB IH 1	43 1 DECA IH 1	53 1 DECB IH 1	63 3-6 DEC ID 2-4	73 4 DEC EX 3	83 2 SUBD IM 3	93 3 SUBD DI 2	A3 3-6 SUBD ID 2-4	B3 3 SUBD EX 3	C3 2 ADDD IM 3	D3 3 ADDD DI 2	E3 3-6 ADDD ID 2-4	F3 3 ADDD EX 3
04 3 loop* RL 3	14 1 ORCC IM 2	24 3/1 BCC RL 2	34 2 PSHX IH 1	44 1 LSRA IH 1	54 1 LSRB IH 1	64 3-6 LSR ID 2-4	74 4 LSR EX 3	84 1 ANDA IM 2	94 3 ANDA DI 2	A4 3-6 ANDA ID 2-4	B4 3 ANDA EX 3	C4 1 ANDB IM 2	D4 3 ANDB DI 2	E4 3-6 ANDB ID 2-4	F4 3 ANDB EX 3
05 3-6 JMP ID 2-4	15 4-7 JSR ID 2-4	25 3/1 BCS RL 2	35 2 PSHY IH 1	45 1 ROLA IH 1	55 1 ROLB IH 1	65 3-6 ROL ID 2-4	75 4 ROL EX 3	85 1 BITA IM 2	95 3 BITA DI 2	A5 3-6 BITA ID 2-4	B5 3 BITA EX 3	C5 1 BITB IM 2	D5 3 BITB DI 2	E5 3-6 BITB ID 2-4	F5 3 BITB EX 3
06 3 JMP EX 3	16 4 JSR EX 3	26 3/1 BNE RL 2	36 2 PSHA IH 1	46 1 RORA IH 1	56 1 RORB IH 1	66 3-6 ROR ID 2-4	76 4 ROR EX 3	86 1 LDAA IM 2	96 3 LDAA DI 2	A6 3-6 LDAA ID 2-4	B6 3 LDAA EX 3	C6 1 LDAB IM 2	D6 3 LDAB DI 2	E6 3-6 LDAB ID 2-4	F6 3 LDAB EX 3
07 4 BSR RL 2	17 4 JSR DI 2	27 3/1 BEQ RL 2	37 2 PSHB IH 1	47 1 ASRA IH 1	57 1 ASRB IH 1	67 3-6 ASR ID 2-4	77 4 ASR EX 3	87 1 CLRA IH 1	97 1 TSTA IH 1	A7 1 NOP IH 1	B7 1 TFR/EXG IH 2	C7 1 CLRB IH 1	D7 1 TSTB IH 1	E7 3-6 TST ID 2-4	F7 3 TST EX 3
08 1 INX IH 1	18 - Page 2 - -	28 3/1 BVC RL 2	38 3 PULC IH 1	48 1 ASLA IH 1	58 1 ASLB IH 1	68 3-6 ASL ID 2-4	78 4 ASL EX 3	88 1 EORA IM 2	98 3 EORA DI 2	A8 3-6 EORA ID 2-4	B8 3 EORA EX 3	C8 1 EORB IM 2	D8 3 EORB DI 2	E8 3-6 EORB ID 2-4	F8 3 EORB EX 3
09 1 DEX IH 1	19 2 LEAY ID 2-4	29 3/1 BVS RL 2	39 2 PSHC IH 1	49 1 LSRD IH 1	59 1 ASLD IH 1	69 ‡2-4 CLR ID 2-4	79 3 CLR EX 3	89 1 ADCA IM 2	99 3 ADCA DI 2	A9 3-6 ADCA ID 2-4	B9 3 ADCA EX 3	C9 1 ADCB IM 2	D9 3 ADCB DI 2	E9 3-6 ADCB ID 2-4	F9 3 ADCB EX 3
0A ‡7 RTC IH 1	1A 2 LEAX ID 2-4	2A 3/1 BPL RL 2	3A 3 PULD IH 1	4A ‡7 CALL EX 4	5A 2 STAA DI 2	6A ‡2-4 STAA ID 2-4	7A 3 STAA EX 3	8A 1 ORAA IM 2	9A 3 ORAA DI 2	AA 3-6 ORAA ID 2-4	BA 3 ORAA EX 3	CA 1 ORAB IM 2	DA 3 ORAB DI 2	EA 3-6 ORAB ID 2-4	FA 3 ORAB EX 3
0B †8 RTI IH 1	1B 2 LEAS ID 2-4	2B 3/1 BMI RL 2	3B 2 PSHD IH 1	4B ‡7-10 CALL ID 2-5	5B 2 STAB DI 2	6B ‡2-4 STAB ID 2-4	7B 3 STAB EX 3	8B 1 ADDA IM 2	9B 3 ADDA DI 2	AB 3-6 ADDA ID 2-4	BB 3 ADDA EX 3	CB 1 ADDB IM 2	DB 3 ADDB DI 2	EB 3-6 ADDB ID 2-4	FB 3 ADDB EX 3
0C 4-6 BSET ID 3-5	1C 4 BSET EX 4	2C 3/1 BGE RL 2	3C ‡+5 wavr SP 1	4C 4 BSET DI 3	5C 2 STD DI 2	6C ‡2-4 STD ID 2-4	7C 3 STD EX 3	8C 2 CPD IM 3	9C 3 CPD DI 2	AC 3-6 CPD ID 2-4	BC 3 CPD EX 3	CC 2 LDD IM 3	DC 3 LDD DI 2	EC 3-6 LDD ID 2-4	FC 3 LDD EX 3
0D 4-6 BCLR ID 3-5	1D 4 BCLR EX 4	2D 3/1 BLT RL 2	3D 5 RTS IH 1	4D 4 BCLR DI 3	5D 2 STY DI 2	6D ‡2-4 STY ID 2-4	7D 3 STY EX 3	8D 2 CPY IM 3	9D 3 CPY DI 2	AD 3-6 CPY ID 2-4	BD 3 CPY EX 3	CD 2 LDY IM 3	DD 3 LDY DI 2	ED 3-6 LDY ID 2-4	FD 3 LDY EX 3
0E ‡4-6 BRSET ID 4-6	1E 5 BRSET EX 5	2E 3/1 BGT RL 2	3E ‡†7 WAI IH 1	4E 4 BRSET DI 4	5E 2 STX DI 2	6E ‡2-4 STX ID 2-4	7E 3 STX EX 3	8E 2 CPX IM 3	9E 3 CPX DI 2	AE 3-6 CPX ID 2-4	BE 3 CPX EX 3	CE 2 LDX IM 3	DE 3 LDX DI 2	EE 3-6 LDX ID 2-4	FE 3 LDX EX 3
0F ‡4-6 BRCLR ID 4-6	1F 5 BRCLR EX 5	2F 3/1 BLE RL 2	3F 9 SWI IH 1	4F 4 BRCLR DI 4	5F 2 STS DI 2	6F ‡2-4 STS ID 2-4	7F 3 STS EX 3	8F 2 CPS IM 3	9F 3 CPS DI 2	AF 3-6 CPS ID 2-4	BF 3 CPS EX 3	CF 2 LDS IM 3	DF 3 LDS DI 2	EF 3-6 LDS ID 2-4	FF 3 LDS EX 3

Key to Table A-2

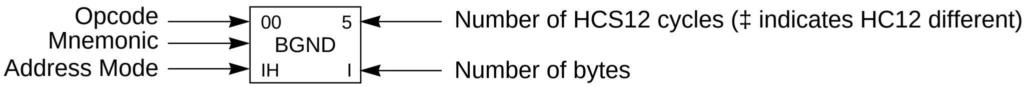


Table A-2. CPU12 Opcode Map (Sheet 2 of 2)

00 4 MOVW IM-ID 5	10 12 IDIV IH 2	20 4 LBRA RL 4	30 10 TRAP IH 2	40 10 TRAP IH 2	50 10 TRAP IH 2	60 10 TRAP IH 2	70 10 TRAP IH 2	80 10 TRAP IH 2	90 10 TRAP IH 2	A0 10 TRAP IH 2	B0 10 TRAP IH 2	C0 10 TRAP IH 2	D0 10 TRAP IH 2	E0 10 TRAP IH 2	F0 10 TRAP IH 2
01 5 MOVW EX-ID 5	11 12 FDIV IH 2	21 3 LBRN RL 4	31 10 TRAP IH 2	41 10 TRAP IH 2	51 10 TRAP IH 2	61 10 TRAP IH 2	71 10 TRAP IH 2	81 10 TRAP IH 2	91 10 TRAP IH 2	A1 10 TRAP IH 2	B1 10 TRAP IH 2	C1 10 TRAP IH 2	D1 10 TRAP IH 2	E1 10 TRAP IH 2	F1 10 TRAP IH 2
02 5 MOVW ID-ID 4	12 13 EMACS SP 4	22 4/3 LBHI RL 4	32 10 TRAP IH 2	42 10 TRAP IH 2	52 10 TRAP IH 2	62 10 TRAP IH 2	72 10 TRAP IH 2	82 10 TRAP IH 2	92 10 TRAP IH 2	A2 10 TRAP IH 2	B2 10 TRAP IH 2	C2 10 TRAP IH 2	D2 10 TRAP IH 2	E2 10 TRAP IH 2	F2 10 TRAP IH 2
03 5 MOVW IM-EX 6	13 3 EMULS IH 2	23 4/3 LBLS RL 4	33 10 TRAP IH 2	43 10 TRAP IH 2	53 10 TRAP IH 2	63 10 TRAP IH 2	73 10 TRAP IH 2	83 10 TRAP IH 2	93 10 TRAP IH 2	A3 10 TRAP IH 2	B3 10 TRAP IH 2	C3 10 TRAP IH 2	D3 10 TRAP IH 2	E3 10 TRAP IH 2	F3 10 TRAP IH 2
04 6 MOVW EX-EX 6	14 12 EDIVS IH 2	24 4/3 LBCC RL 4	34 10 TRAP IH 2	44 10 TRAP IH 2	54 10 TRAP IH 2	64 10 TRAP IH 2	74 10 TRAP IH 2	84 10 TRAP IH 2	94 10 TRAP IH 2	A4 10 TRAP IH 2	B4 10 TRAP IH 2	C4 10 TRAP IH 2	D4 10 TRAP IH 2	E4 10 TRAP IH 2	F4 10 TRAP IH 2
05 5 MOVW ID-EX 5	15 12 IDIVS IH 2	25 4/3 LBCS RL 4	35 10 TRAP IH 2	45 10 TRAP IH 2	55 10 TRAP IH 2	65 10 TRAP IH 2	75 10 TRAP IH 2	85 10 TRAP IH 2	95 10 TRAP IH 2	A5 10 TRAP IH 2	B5 10 TRAP IH 2	C5 10 TRAP IH 2	D5 10 TRAP IH 2	E5 10 TRAP IH 2	F5 10 TRAP IH 2
06 2 ABA IH 2	16 2 SBA IH 2	26 4/3 LBNE RL 4	36 10 TRAP IH 2	46 10 TRAP IH 2	56 10 TRAP IH 2	66 10 TRAP IH 2	76 10 TRAP IH 2	86 10 TRAP IH 2	96 10 TRAP IH 2	A6 10 TRAP IH 2	B6 10 TRAP IH 2	C6 10 TRAP IH 2	D6 10 TRAP IH 2	E6 10 TRAP IH 2	F6 10 TRAP IH 2
07 3 DAA IH 2	17 2 CBA IH 2	27 4/3 LBEQ RL 4	37 10 TRAP IH 2	47 10 TRAP IH 2	57 10 TRAP IH 2	67 10 TRAP IH 2	77 10 TRAP IH 2	87 10 TRAP IH 2	97 10 TRAP IH 2	A7 10 TRAP IH 2	B7 10 TRAP IH 2	C7 10 TRAP IH 2	D7 10 TRAP IH 2	E7 10 TRAP IH 2	F7 10 TRAP IH 2
08 4 MOVB IM-ID 4	18 4-7 MAXA ID 3-5	28 4/3 LBVC RL 4	38 10 TRAP IH 2	48 10 TRAP IH 2	58 10 TRAP IH 2	68 10 TRAP IH 2	78 10 TRAP IH 2	88 10 TRAP IH 2	98 10 TRAP IH 2	A8 10 TRAP IH 2	B8 10 TRAP IH 2	C8 10 TRAP IH 2	D8 10 TRAP IH 2	E8 10 TRAP IH 2	F8 10 TRAP IH 2
09 5 MOVB EX-ID 5	19 4-7 MINA ID 3-5	29 4/3 LBVS RL 4	39 10 TRAP IH 2	49 10 TRAP IH 2	59 10 TRAP IH 2	69 10 TRAP IH 2	79 10 TRAP IH 2	89 10 TRAP IH 2	99 10 TRAP IH 2	A9 10 TRAP IH 2	B9 10 TRAP IH 2	C9 10 TRAP IH 2	D9 10 TRAP IH 2	E9 10 TRAP IH 2	F9 10 TRAP IH 2
0A 5 MOVB ID-ID 4	1A 4-7 EMAXD ID 3-5	2A 4/3 LBPL RL 4	3A †3n REV SP 2	4A 10 TRAP IH 2	5A 10 TRAP IH 2	6A 10 TRAP IH 2	7A 10 TRAP IH 2	8A 10 TRAP IH 2	9A 10 TRAP IH 2	AA 10 TRAP IH 2	BA 10 TRAP IH 2	CA 10 TRAP IH 2	DA 10 TRAP IH 2	EA 10 TRAP IH 2	FA 10 TRAP IH 2
0B 4 MOVB IM-EX 5	1B 4-7 EMIND ID 3-5	2B 4/3 LBMI RL 4	3B †5n/3n REVW SP 2	4B 10 TRAP IH 2	5B 10 TRAP IH 2	6B 10 TRAP IH 2	7B 10 TRAP IH 2	8B 10 TRAP IH 2	9B 10 TRAP IH 2	AB 10 TRAP IH 2	BB 10 TRAP IH 2	CB 10 TRAP IH 2	DB 10 TRAP IH 2	EB 10 TRAP IH 2	FB 10 TRAP IH 2
0C 6 MOVB EX-EX 6	1C 4-7 MAXM ID 3-5	2C 4/3 LBGE RL 4	3C ‡†7B WAV SP 2	4C 10 TRAP IH 2	5C 10 TRAP IH 2	6C 10 TRAP IH 2	7C 10 TRAP IH 2	8C 10 TRAP IH 2	9C 10 TRAP IH 2	AC 10 TRAP IH 2	BC 10 TRAP IH 2	CC 10 TRAP IH 2	DC 10 TRAP IH 2	EC 10 TRAP IH 2	FC 10 TRAP IH 2
0D 5 MOVB ID-EX 5	1D D4-7 MINM ID 3-5	2D 4/3 LBLT RL 4	3D ‡6 TBL ID 3	4D 10 TRAP IH 2	5D 10 TRAP IH 2	6D 10 TRAP IH 2	7D 10 TRAP IH 2	8D 10 TRAP IH 2	9D 10 TRAP IH 2	AD 10 TRAP IH 2	BD 10 TRAP IH 2	CD 10 TRAP IH 2	DD 10 TRAP IH 2	ED 10 TRAP IH 2	FD 10 TRAP IH 2
0E 2 TAB IH 2	1E 4-7 EMAXM ID 3-5	2E 4/3 LBGT RL 4	3E ‡8 STOP IH 2	4E 10 TRAP IH 2	5E 10 TRAP IH 2	6E 10 TRAP IH 2	7E 10 TRAP IH 2	8E 10 TRAP IH 2	9E 10 TRAP IH 2	AE 10 TRAP IH 2	BE 10 TRAP IH 2	CE 10 TRAP IH 2	DE 10 TRAP IH 2	EE 10 TRAP IH 2	FE 10 TRAP IH 2
0F 2 TBA IH 2	1F 4-7 EMINM ID 3-5	2F 4/3 LBLE RL 4	3F 10 ETBL ID 3	4F 10 TRAP IH 2	5F 10 TRAP IH 2	6F 10 TRAP IH 2	7F 10 TRAP IH 2	8F 10 TRAP IH 2	9F 10 TRAP IH 2	AF 10 TRAP IH 2	BF 10 TRAP IH 2	CF 10 TRAP IH 2	DF 10 TRAP IH 2	EF 10 TRAP IH 2	FF 10 TRAP IH 2

* The opcode $04 (on sheet 1 of 2) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

† Refer to instruction summary for more information.

‡ Refer to instruction summary for different HC12 cycle count.

Page 2: When the CPU encounters a page 2 opcode ($18 on page 1 of the opcode map), it treats the next byte of object code as a page 2 instruction opcode.

Table A-3. Indexed Addressing Mode Postbyte Encoding (xb)

00 0,X 5b const	10 –16,X 5b const	20 1,+X pre-inc	30 1,X+ post-inc	40 0,Y 5b const	50 –16,Y 5b const	60 1,+Y pre-inc	70 1,Y+ post-inc	80 0,SP 5b const	90 –16,SP 5b const	A0 1,+SP pre-inc	B0 1,SP+ post-inc	C0 0,PC 5b const	D0 –16,PC 5b const	E0 n,X 9b const	F0 n,SP 9b const
01 1,X 5b const	11 –15,X 5b const	21 2,+X pre-inc	31 2,X+ post-inc	41 1,Y 5b const	51 –15,Y 5b const	61 2,+Y pre-inc	71 2,Y+ post-inc	81 1,SP 5b const	91 –15,SP 5b const	A1 2,+SP pre-inc	B1 2,SP+ post-inc	C1 1,PC 5b const	D1 –15,PC 5b const	E1 –n,X 9b const	F1 –n,SP 9b const
02 2,X 5b const	12 –14,X 5b const	22 3,+X pre-inc	32 3,X+ post-inc	42 2,Y 5b const	52 –14,Y 5b const	62 3,+Y pre-inc	72 3,Y+ post-inc	82 2,SP 5b const	92 –14,SP 5b const	A2 3,+SP pre-inc	B2 3,SP+ post-inc	C2 2,PC 5b const	D2 –14,PC 5b const	E2 n,X 16b const	F2 n,SP 16b const
03 3,X 5b const	13 –13,X 5b const	23 4,+X pre-inc	33 4,X+ post-inc	43 3,Y 5b const	53 –13,Y 5b const	63 4,+Y pre-inc	73 4,Y+ post-inc	83 3,SP 5b const	93 –13,SP 5b const	A3 4,+SP pre-inc	B3 4,SP+ post-inc	C3 3,PC 5b const	D3 –13,PC 5b const	E3 [n,X] 16b indr	F3 [n,SP] 16b indr
04 4,X 5b const	14 –12,X 5b const	24 5,+X pre-inc	34 5,X+ post-inc	44 4,Y 5b const	54 –12,Y 5b const	64 5,+Y pre-inc	74 5,Y+ post-inc	84 4,SP 5b const	94 –12,SP 5b const	A4 5,+SP pre-inc	B4 5,SP+ post-inc	C4 4,PC 5b const	D4 –12,PC 5b const	E4 A,X A offset	F4 A,SP A offset
05 5,X 5b const	15 –11,X 5b const	25 6,+X pre-inc	35 6,X+ post-inc	45 5,Y 5b const	55 –11,Y 5b const	65 6,+Y pre-inc	75 6,Y+ post-inc	85 5,SP 5b const	95 –11,SP 5b const	A5 6,+SP pre-inc	B5 6,SP+ post-inc	C5 5,PC 5b const	D5 –11,PC 5b const	E5 B,X B offset	F5 B,SP B offset
06 6,X 5b const	16 –10,X 5b const	26 7,+X pre-inc	36 7,X+ post-inc	46 6,Y 5b const	56 –10,Y 5b const	66 7,+Y pre-inc	76 7,Y+ post-inc	86 6,SP 5b const	96 –10,SP 5b const	A6 7,+SP pre-inc	B6 7,SP+ post-inc	C6 6,PC 5b const	D6 –10,PC 5b const	E6 D,X D offset	F6 D,SP D offset
07 7,X 5b const	17 –9,X 5b const	27 8,+X pre-inc	37 8,X+ post-inc	47 7,Y 5b const	57 –9,Y 5b const	67 8,+Y pre-inc	77 8,Y+ post-inc	87 7,SP 5b const	97 –9,SP 5b const	A7 8,+SP pre-inc	B7 8,SP+ post-inc	C7 7,PC 5b const	D7 –9,PC 5b const	E7 [D,X] D indirect	F7 [D,SP] D indirect
08 8,X 5b const	18 –8,X 5b const	28 8,–X pre-dec	38 8,X– post-dec	48 8,Y 5b const	58 –8,Y 5b const	68 8,–Y pre-dec	78 8,Y– post-dec	88 8,SP 5b const	98 –8,SP 5b const	A8 8,–SP pre-dec	B8 8,SP– post-dec	C8 8,PC 5b const	D8 –8,PC 5b const	E8 n,Y 9b const	F8 n,PC 9b const
09 9,X 5b const	19 –7,X 5b const	29 7,–X pre-dec	39 7,X– post-dec	49 9,Y 5b const	59 –7,Y 5b const	69 7,–Y pre-dec	79 7,Y– post-dec	89 9,SP 5b const	99 –7,SP 5b const	A9 7,–SP pre-dec	B9 7,SP– post-dec	C9 9,PC 5b const	D9 –7,PC 5b const	E9 –n,Y 9b const	F9 –n,PC 9b const
0A 10,X 5b const	1A –6,X 5b const	2A 6,–X pre-dec	3A 6,X– post-dec	4A 10,Y 5b const	5A –6,Y 5b const	6A 6,–Y pre-dec	7A 6,Y– post-dec	8A 10,SP 5b const	9A –6,SP 5b const	AA 6,–SP pre-dec	BA 6,SP– post-dec	CA 10,PC 5b const	DA –6,PC 5b const	EA n,Y 16b const	FA n,PC 16b const
0B 11,X 5b const	1B –5,X 5b const	2B 5,–X pre-dec	3B 5,X– post-dec	4B 11,Y 5b const	5B –5,Y 5b const	6B 5,–Y pre-dec	7B 5,Y– post-dec	8B 11,SP 5b const	9B –5,SP 5b const	AB 5,–SP pre-dec	BB 5,SP– post-dec	CB 11,PC 5b const	DB –5,PC 5b const	EB [n,Y] 16b indr	FB [n,PC] 16b indr
0C 12,X 5b const	1C –4,X 5b const	2C 4,–X pre-dec	3C 4,X– post-dec	4C 12,Y 5b const	5C –4,Y 5b const	6C 4,–Y pre-dec	7C 4,Y– post-dec	8C 12,SP 5b const	9C –4,SP 5b const	AC 4,–SP pre-dec	BC 4,SP– post-dec	CC 12,PC 5b const	DC –4,PC 5b const	EC A,Y A offset	FC A,PC A offset
0D 13,X 5b const	1D –3,X 5b const	2D 3,–X pre-dec	3D 3,X– post-dec	4D 13,Y 5b const	5D –3,Y 5b const	6D 3,–Y pre-dec	7D 3,Y– post-dec	8D 13,SP 5b const	9D –3,SP 5b const	AD 3,–SP pre-dec	BD 3,SP– post-dec	CD 13,PC 5b const	DD –3,PC 5b const	ED B,Y B offset	FD B,PC B offset
0E 14,X 5b const	1E –2,X 5b const	2E 2,–X pre-dec	3E 2,X– post-dec	4E 14,Y 5b const	5E –2,Y 5b const	6E 2,–Y pre-dec	7E 2,Y– post-dec	8E 14,SP 5b const	9E –2,SP 5b const	AE 2,–SP pre-dec	BE 2,SP– post-dec	CE 14,PC 5b const	DE –2,PC 5b const	EE D,Y D offset	FE D,PC D offset
0F 15,X 5b const	1F –1,X 5b const	2F 1,–X pre-dec	3F 1,X– post-dec	4F 15,Y 5b const	5F –1,Y 5b const	6F 1,–Y pre-dec	7F 1,Y– post-dec	8F 15,SP 5b const	9F –1,SP 5b const	AF 1,–SP pre-dec	BF 1,SP– post-dec	CF 15,PC 5b const	DF –1,PC 5b const	EF [D,Y] D indirect	FF [D,PC] D indirect

Key to Table A-3

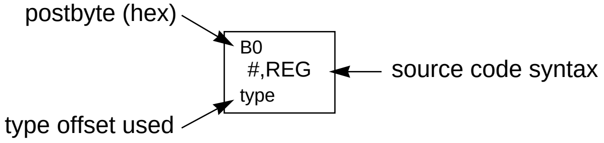


Table A-4. Indexed Addressing Mode Summary

Postbyte Code (xb)	Operand Syntax	Comments
rr0nnnnn	,r n,r –n,r	**5-bit constant offset** n = –16 to +15 rr can specify X, Y, SP, or PC
111rr0zs	n,r –n,r	**Constant offset** (9- or 16-bit signed) z- 0 = 9-bit with sign in LSB of postbyte (s) 1 = 16-bit if z = s = 1, 16-bit offset indexed-indirect (see below) rr can specify X, Y, SP, or PC
rr1pnnnn	n,–r n,+r n,r– n,r+	**Auto predecrement, preincrement, postdecrement, or postincrement**; p = pre-(0) or post-(1), n = –8 to –1, +1 to +8 rr can specify X, Y, or SP (PC not a valid choice)
111rr1aa	A,r B,r D,r	**Accumulator offset** (unsigned 8-bit or 16-bit) aa - 00 = A 01 = B 10 = D (16-bit) 11 = see accumulator D offset indexed-indirect rr can specify X, Y, SP, or PC
111rr011	[n,r]	**16-bit offset indexed-indirect** rr can specify X, Y, SP, or PC
111rr111	[D,r]	**Accumulator D offset indexed-indirect** rr can specify X, Y, SP, or PC

Table A-5. Transfer and Exchange Postbyte Encoding

TRANSFERS

⇓LS	MS⇒	0	1	2	3	4	5	6	7
0		A ⇒ A	B ⇒ A	CCR ⇒ A	$TMP3_L$ ⇒ A	B ⇒ A	X_L ⇒ A	Y_L ⇒ A	SP_L ⇒ A
1		A ⇒ B	B ⇒ B	CCR ⇒ B	$TMP3_L$ ⇒ B	B ⇒ B	X_L ⇒ B	Y_L ⇒ B	SP_L ⇒ B
2		A ⇒ CCR	B ⇒ CCR	CCR ⇒ CCR	$TMP3_L$ ⇒ CCR	B ⇒ CCR	X_L ⇒ CCR	Y_L ⇒ CCR	SP_L ⇒ CCR
3		sex:A ⇒ TMP2	sex:B ⇒ TMP2	sex:CCR ⇒ TMP2	TMP3 ⇒ TMP2	D ⇒ TMP2	X ⇒ TMP2	Y ⇒ TMP2	SP ⇒ TMP2
4		sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR ⇒ D SEX CCR,D	TMP3 ⇒ D	D ⇒ D	X ⇒ D	Y ⇒ D	SP ⇒ D
5		sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR ⇒ X SEX CCR,X	TMP3 ⇒ X	D ⇒ X	X ⇒ X	Y ⇒ X	SP ⇒ X
6		sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR ⇒ Y SEX CCR,Y	TMP3 ⇒ Y	D ⇒ Y	X ⇒ Y	Y ⇒ Y	SP ⇒ Y
7		sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR ⇒ SP SEX CCR,SP	TMP3 ⇒ SP	D ⇒ SP	X ⇒ SP	Y ⇒ SP	SP ⇒ SP

EXCHANGES

⇓LS	MS⇒	8	9	A	B	C	D	E	F
0		A ⇔ A	B ⇔ A	CCR ⇔ A	$TMP3_L$ ⇒ A $00:A ⇒ TMP3	B ⇒ A A ⇒ B	X_L ⇒ A $00:A ⇒ X	Y_L ⇒ A $00:A ⇒ Y	SP_L ⇒ A $00:A ⇒ SP
1		A ⇔ B	B ⇔ B	CCR ⇔ B	$TMP3_L$ ⇒ B $FF:B ⇒ TMP3	B ⇒ B $FF ⇒ A	X_L ⇒ B $FF:B ⇒ X	Y_L ⇒ B $FF:B ⇒ Y	SP_L ⇒ B $FF:B ⇒ SP
2		A ⇔ CCR	B ⇔ CCR	CCR ⇔ CCR	$TMP3_L$ ⇒ CCR $FF:CCR ⇒ TMP3	B ⇒ CCR $FF:CCR ⇒ D	X_L ⇒ CCR $FF:CCR ⇒ X	Y_L ⇒ CCR $FF:CCR ⇒ Y	SP_L ⇒ CCR $FF:CCR ⇒ SP
3		$00:A ⇒ TMP2 $TMP2_L$ ⇒ A	$00:B ⇒ TMP2 $TMP2_L$ ⇒ B	$00:CCR ⇒ TMP2 $TMP2_L$ ⇒ CCR	TMP3 ⇔ TMP2	D ⇔ TMP2	X ⇔ TMP2	Y ⇔ TMP2	SP ⇔ TMP2
4		$00:A ⇒ D	$00:B ⇒ D	$00:CCR ⇒ D B ⇒ CCR	TMP3 ⇔ D	D ⇔ D	X ⇔ D	Y ⇔ D	SP ⇔ D
5		$00:A ⇒ X X_L ⇒ A	$00:B ⇒ X X_L ⇒ B	$00:CCR ⇒ X X_L ⇒ CCR	TMP3 ⇔ X	D ⇔ X	X ⇔ X	Y ⇔ X	SP ⇔ X
6		$00:A ⇒ Y Y_L ⇒ A	$00:B ⇒ Y Y_L ⇒ B	$00:CCR ⇒ Y Y_L ⇒ CCR	TMP3 ⇔ Y	D ⇔ Y	X ⇔ Y	Y ⇔ Y	SP ⇔ Y
7		$00:A ⇒ SP SP_L ⇒ A	$00:B ⇒ SP SP_L ⇒ B	$00:CCR ⇒ SP SP_L ⇒ CCR	TMP3 ⇔ SP	D ⇔ SP	X ⇔ SP	Y ⇔ SP	SP ⇔ SP

TMP2 and TMP3 registers are for factory use only.

Table A-6. Loop Primitive Postbyte Encoding (lb)

00 A DBEQ (+)	10 A DBEQ (–)	20 A DBNE (+)	30 A DBNE (–)	40 A TBEQ (+)	50 A TBEQ (–)	60 A TBNE (+)	70 A TBNE (–)	80 A IBEQ (+)	90 A IBEQ (–)	A0 A IBNE (+)	B0 A IBNE (–)
01 B DBEQ (+)	11 B DBEQ (–)	21 B DBNE (+)	31 B DBNE (–)	41 B TBEQ (+)	51 B TBEQ (–)	61 B TBNE (+)	71 B TBNE (–)	81 B IBEQ (+)	91 B IBEQ (–)	A1 B IBNE (+)	B1 B IBNE (–)
02 —	12 —	22 —	32 —	42 —	52 —	62 —	72 —	82 —	92 —	A2 —	B2 —
03 —	13 —	23 —	33 —	43 —	53 —	63 —	73 —	83 —	93 —	A3 —	B3 —
04 D DBEQ (+)	14 D DBEQ (–)	24 D DBNE (+)	34 D DBNE (–)	44 D TBEQ (+)	54 D TBEQ (–)	64 D TBNE (+)	74 D TBNE (–)	84 D IBEQ (+)	94 D IBEQ (–)	A4 D IBNE (+)	B4 D IBNE (–)
05 X DBEQ (+)	15 X DBEQ (–)	25 X DBNE (+)	35 X DBNE (–)	45 X TBEQ (+)	55 X TBEQ (–)	65 X TBNE (+)	75 X TBNE (–)	85 X IBEQ (+)	95 X IBEQ (–)	A5 X IBNE (+)	B5 X IBNE (–)
06 Y DBEQ (+)	16 Y DBEQ (–)	26 Y DBNE (+)	36 Y DBNE (–)	46 Y TBEQ (+)	56 Y TBEQ (–)	66 Y TBNE (+)	76 Y TBNE (–)	86 Y IBEQ (+)	96 Y IBEQ (–)	A6 Y IBNE (+)	B6 Y IBNE (–)
07 SP DBEQ (+)	17 SP DBEQ (–)	27 SP DBNE (+)	37 SP DBNE (–)	47 SP TBEQ (+)	57 SP TBEQ (–)	67 SP TBNE (+)	77 SP TBNE (–)	87 SP IBEQ (+)	97 SP IBEQ (–)	A7 SP IBNE (+)	B7 SP IBNE (–)

Key to Table A-6

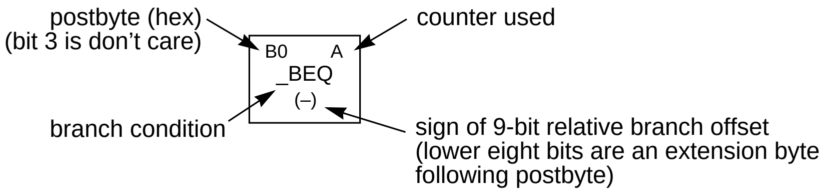


Table A-7. Branch/Complementary Branch

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	Z + (N ⊕ V) = 0	r≤m	BLE	2F	Signed
r≥m	BGE	2C	N ⊕ V = 0	r<m	BLT	2D	Signed
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Signed
r≤m	BLE	2F	Z + (N ⊕ V) = 1	r>m	BGT	2E	Signed
r<m	BLT	2D	N ⊕ V = 1	r≥m	BGE	2C	Signed
r>m	BHI	22	C + Z = 0	r≤m	BLS	23	Unsigned
r≥m	BHS/BCC	24	C = 0	r<m	BLO/BCS	25	Unsigned
r=m	BEQ	27	Z = 1	r≠m	BNE	26	Unsigned
r≤m	BLS	23	C + Z = 1	r>m	BHI	22	Unsigned
r<m	BLO/BCS	25	C = 1	r≥m	BHS/BCC	24	Unsigned
Carry	BCS	25	C = 1	No Carry	BCC	24	Simple
Negative	BMI	2B	N = 1	Plus	BPL	2A	Simple
Overflow	BVS	29	V = 1	No Overflow	BVC	28	Simple
r=0	BEQ	27	Z = 1	r≠0	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

For 16-bit offset long branches precede opcode with a $18 page prebyte.

Table A-8. Hexadecimal to ASCII Conversion

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
$00	NUL	$20	SP *space*	$40	@	$60	` *grave*
$01	SOH	$21	!	$41	A	$61	a
$02	STX	$22	“ quote	$42	B	$62	b
$03	ETX	$23	#	$43	C	$63	c
$04	EOT	$24	$	$44	D	$64	d
$05	ENQ	$25	%	$45	E	$65	e
$06	ACK	$26	&	$46	F	$66	f
$07	BEL *beep*	$27	‘ *apost.*	$47	G	$67	g
$08	BS *back sp*	$28	(	$48	H	$68	h
$09	HT *tab*	$29	)	$49	I	$69	i
$0A	LF *linefeed*	$2A	*	$4A	J	$6A	j
$0B	VT	$2B	+	$4B	K	$6B	k
$0C	FF	$2C	, *comma*	$4C	L	$6C	l
$0D	CR *return*	$2D	- *dash*	$4D	M	$6D	m
$0E	SO	$2E	. *period*	$4E	N	$6E	n
$0F	SI	$2F	/	$4F	O	$6F	o
$10	DLE	$30	0	$50	P	$70	p
$11	DC1	$31	1	$51	Q	$71	q
$12	DC2	$32	2	$52	R	$72	r
$13	DC3	$33	3	$53	S	$73	s
$14	DC4	$34	4	$54	T	$74	t
$15	NAK	$35	5	$55	U	$75	u
$16	SYN	$36	6	$56	V	$76	v
$17	ETB	$37	7	$57	W	$77	w
$18	CAN	$38	8	$58	X	$78	x
$19	EM	$39	9	$59	Y	$79	y
$1A	SUB	$3A	:	$5A	Z	$7A	z
$1B	ESCAPE	$3B	;	$5B	[	$7B	{
$1C	FS	$3C	<	$5C	\	$7C	|
$1D	GS	$3D	=	$5D	]	$7D	}
$1E	RS	$3E	>	$5E	^	$7E	~
$1F	US	$3F	?	$5F	_ *under*	$7F	DEL *delete*

A.5 Hexadecimal to Decimal Conversion

To convert a hexadecimal number (up to four hexadecimal digits) to decimal, look up the decimal equivalent of each hexadecimal digit in **Table A-9**. The decimal equivalent of the original hexadecimal number is the sum of the weights found in the table for all hexadecimal digits.

Table A-9. Hexadecimal to/from Decimal Conversion

15 Bit 8				7 Bit 0			
15 12		11 8		7 4		3 0	
4th Hex Digit		3rd Hex Digit		2nd Hex Digit		1st Hex Digit	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,484	E	224	E	14
F	61,440	F	3,840	F	240	F	15

A.6 Decimal to Hexadecimal Conversion

To convert a decimal number (up to 65,535$_{10}$) to hexadecimal, find the largest decimal number in **Table A-9** that is less than or equal to the number you are converting. The corresponding hexadecimal digit is the most significant hexadecimal digit of the result. Subtract the decimal number found from the original decimal number to get the *remaining decimal value*. Repeat the procedure using the remaining decimal value for each subsequent hexadecimal digit.

Appendix B. M68HC11 to CPU12 Upgrade Path

B.1 Introduction

This appendix discusses similarities and differences between the CPU12 and the M68HC11 CPU. In general, the CPU12 is a proper superset of the M68HC11. Significant changes have been made to improve the efficiency and capabilities of the CPU12 without eliminating compatibility and familiarity for the large community of M68HC11 programmers.

B.2 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

B.3 Source Code Compatibility

Every M68HC11 instruction mnemonic and source code statement can be assembled directly with a CPU12 assembler with no modifications.

The CPU12 supports all M68HC11 addressing modes and includes several new variations of indexed addressing mode. CPU12 instructions affect condition code bits in the same way as M68HC11 instructions.

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 according to the result of the decrement or increment.

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

Table B-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 is one byte on M68HC11 but ABY is two bytes. The LEA substitutes are two bytes.

Continued on next page

Table B-1. Translated M68HC11 Mnemonics (Continued)

M68HC11 Mnemonic	Equivalent CPU12 Instruction	Comments
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 take 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 are 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 has 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 the M68HC11. 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 2-page opcode map, rather than the 4-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 CPU12, and some M68HC11 opcodes have been moved to page 1 of the CPU12 opcode map. Object code for each of the moved instructions is one byte smaller than object code for the equivalent M68HC11 instruction. **Table B-2** shows instructions that assemble to one byte less object code on the CPU12.

Table B-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.

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 code for M68HC11 vs. code for CPU12 has also been tested by rewriting several smaller programs from scratch. In these cases, the CPU12 code is typically about 30 percent smaller. These savings are mostly due to improved indexed addressing.

It seems useful to mention the results of size comparisons done on C programs. A C program compiled for the CPU12 is about 30 percent smaller than the same program compiled for the M68HC11. The savings are largely due to better indexing.

B.4 Programmer’s Model and Stacking

The CPU12 programming model and stacking order are identical to those of the M68HC11.

B.5 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 eight 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 arithmetic logic unit (ALU) of the CPU12 (it can perform some 20-bit operations) is used to calculate 16-bit pointers and to speed up math operations.

B.5.1 Bus Structures

The CPU12 is a 16-bit processor with 16-bit data paths. Typical HCS12 and 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. HCS12 and 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 the 16-bit data bus. This is similar to the way an M68HC11 can stretch clock cycles to accommodate slow peripherals.

B.5.2 Instruction Queue

The CPU12 has a 2-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 because 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 in 16-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.

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.

B.5.3 Stack Function

Both the M68HC11 and the CPU12 stack nine bytes for interrupts. Since this is an odd number of bytes, there is no practical way to ensure that the stack will stay aligned. To ensure 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 on 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.

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 allows 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. This adjustment causes the X index to point to the last stack location used. The TXS instruction operates similarly, except that it decrements the SP by one during the transfer. CPU12 TSX and TXS instructions are ordinary transfers — the CPU12 stack requires no adjustment.

For ordinary use of the stack, such as pushes, pulls, and even manipulations involving TSX and TXS, there are no differences in the way the M68HC11 and the CPU12 stacks look to a programmer. However, the stack change can affect a program algorithm in two subtle ways.

The LDS #$xxxx instruction is normally used to initialize the stack pointer at the start of a program. In the M68HC11, the address specified in the LDS instruction is the first stack location used. In the CPU12, however, the first stack location used is one address lower than the address specified in the LDS instruction. Since the stack builds downward, M68HC11 programs reassembled for the CPU12 operate normally, but the program stack is 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, change either the initial LDS #$xxxx or the absolute extended address used to read the stack.

B.6 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 1-byte offset.) Analysis of M68HC11 source code indicates that the offset is most frequently zero and 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 (very important for C compilers)
- 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 (M68HC11 only supported positive unsigned 8-bit 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.

Table A-3 and **Table A-4** summarize CPU12 indexed addressing mode capabilities. **Table A-6** shows how the postbyte is encoded.

B.6.1 Constant Offset Indexing

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

The most common constant offset is 0. Offsets of 1, 2, 3, 4 are used fairly often, but with less frequency than 0.

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 M68HC11 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 +32,767) 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.

B.6.2 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.

18 A6 00 18 08 18 08	LDAA 0,Y INY 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 3-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.

B.6.3 Accumulator Offset Indexing

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

```
C6 05      LDAB     #$5    [2]
CE 10 00   LOOP LDX #$1000 [3]  C6 05      LDAB  #$5       [1]
3A         ABX             [3]  CE 10 00   LDX  #$1000     [2]
A6 00      LDAA     0,X    [4]  A6 E5      LOOP LDAA  B,X  [3]
            |                               |
5A         DECB            [2]  04 31 FB   DBNE  B,LOOP    [3]
26 F7      BNE      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 the loop execution time from 15 cycles to six cycles. This reduction, combined with the 25-MHz bus speed of the HCS12 (M68HC12) Family, can have significant effects.

B.6.4 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.

B.7 Improved Performance

The HCS12 uses a system-on-a-chip (SoC) design methodology and is normally implemented in a 0.25µ FLASH process. HCS12 devices can operate at up to 25 MHz and are designed to be migrated easily to faster, smaller silicon process technologies as they are developed.

The M68HC12 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 of the CPU12 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.

B.7.1 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 ensures 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.

B.7.2 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 B-3 compares the speed of CPU12 and M68HC11 math instructions. The CPU12 requires fewer cycles to perform an operation, and the cycle time is considerably faster than that of the M68HC11.

Table B-3. Comparison of Math Instruction Speeds

Instruction Mnemonic	Math Operation	M68HC11 1 Cycle = 250 ns	M68HC11 With Coprocessor 1 Cycle = 250 ns	CPU12 1 Cycle = 40 ns (125 ns in M68HC12)
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	$32 \times (16 \times 16) \Rightarrow 32$ (signed MAC)	—	20 cycles	12 cycles

The IDIVS instruction is included specifically for C compilers, where word-sized operands are divided to produce a word-sized result (unlike the $32 \div 16 = 16$ 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.

B.7.3 Code Size Reduction

CPU12 assembly language programs written from scratch tend to be 30 percent 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 operations on index pointers are 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.

B.8 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. **Table B-4** is a summary of these new instructions. Subsequent paragraphs discuss significant enhancements.

Table B-4. New M68HC12 Instructions (Sheet 1 of 2)

Mnemonic	Addressing Modes	Brief Functional Description
ANDCC	Immediate	AND CCR with mask (replaces CLC, CLI, and CLV)
BCLR	Extended	Bit(s) clear (added extended mode)
BGND	Inherent	Enter background debug mode, if enabled
BRCLR	Extended	Branch if bit(s) clear (added extended mode)
BRSET	Extended	Branch if bit(s) set (added extended mode)
BSET	Extended	Bit(s) set (added extended mode)
CALL	Extended, indexed	Similar to JSR except also stacks PPAGE value; with RTC instruction, allows easy access to >64-Kbyte space
CPS	Immediate, direct, extended, and indexed	Compare stack pointer
DBNE	Relative	Decrement and branch if equal to zero (looping primitive)
DBEQ	Relative	Decrement and branch if not equal to zero (looping primitive)
EDIV	Inherent	Extended divide Y:D/X = Y(Q) and D(R) (unsigned)
EDIVS	Inherent	Extended divide Y:D/X = Y(Q) and D(R) (signed)
EMACS	Special	Multiply and accumulate $16 \times 16 \Rightarrow 32$ (signed)
EMAXD	Indexed	Maximum of two unsigned 16-bit values
EMAXM	Indexed	Maximum of two unsigned 16-bit values
EMIND	Indexed	Minimum of two unsigned 16-bit values
EMINM	Indexed	Minimum of two unsigned 16-bit values
EMUL	Special	Extended multiply $16 \times 16 \Rightarrow 32$; M(idx) $*$ D $\Rightarrow$ Y:D
EMULS	Special	Extended multiply $16 \times 16 \Rightarrow 32$ (signed); M(idx) $*$ D $\Rightarrow$ Y:D
ETBL	Special	Table lookup and interpolate (16-bit entries)
EXG	Inherent	Exchange register contents
IBEQ	Relative	Increment and branch if equal to zero (looping primitive)
IBNE	Relative	Increment and branch if not equal to zero (looping primitive)
IDIVS	Inherent	Signed integer divide D/X $\Rightarrow$ X(Q) and D(R) (signed)
LBCC	Relative	Long branch if carry clear (same as LBHS)
LBCS	Relative	Long branch if carry set (same as LBLO)
LBEQ	Relative	Long branch if equal (Z=1)
LBGE	Relative	Long branch if greater than or equal to zero
LBGT	Relative	Long branch if greater than zero
LBHI	Relative	Long branch if higher
LBHS	Relative	Long branch if higher or same (same as LBCC)
LBLE	Relative	Long branch if less than or equal to zero
LBLO	Relative	Long branch if lower (same as LBCS)
LBLS	Relative	Long branch if lower or same

Table B-4. New M68HC12 Instructions (Sheet 2 of 2)

Mnemonic	Addressing Modes	Brief Functional Description
LBLT	Relative	Long branch if less than zero
LBMI	Relative	Long branch if minus
LBNE	Relative	Long branch if not equal to zero
LBPL	Relative	Long branch if plus
LBRA	Relative	Long branch always
LBRN	Relative	Long branch never
LBVC	Relative	Long branch if overflow clear
LBVS	Relative	Long branch if overflow set
LEAS	Indexed	Load stack pointer with effective address
LEAX	Indexed	Load X index register with effective address
LEAY	Indexed	Load Y index register with effective address
MAXA	Indexed	Maximum of two unsigned 8-bit values
MAXM	Indexed	Maximum of two unsigned 8-bit values
MEM	Special	Determine grade of fuzzy membership
MINA	Indexed	Minimum of two unsigned 8-bit values
MINM	Indexed	Minimum of two unsigned 8-bit values
MOVB(W)	Combinations of immediate, extended, and indexed	Move data from one memory location to another
ORCC	Immediate	OR CCR with mask (replaces SEC, SEI, and SEV)
PSHC	Inherent	Push CCR onto stack
PSHD	Inherent	Push double accumulator onto stack
PULC	Inherent	Pull CCR contents from stack
PULD	Inherent	Pull double accumulator from stack
REV	Special	Fuzzy logic rule evaluation
REVW	Special	Fuzzy logic rule evaluation with weights
RTC	Inherent	Restore program page and return address from stack used with CALL instruction, allows easy access to >64-Kbyte space
SEX	Inherent	Sign extend 8-bit register into 16-bit register
TBEQ	Relative	Test and branch if equal to zero (looping primitive)
TBL	Inherent	Table lookup and interpolate (8-bit entries)
TBNE	Relative	Test register and branch if not equal to zero (looping primitive)
TFR	Inherent	Transfer register contents to another register
WAV	Special	Weighted average (fuzzy logic support)

B.8.1 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 5-bit signed constant offsets, 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.

B.8.2 Universal Transfer and Exchange

The M68HC11 has only eight 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 produces more compact object code than saving data to memory with pushes or stores.

B.8.3 Loop Construct

The CPU12 instruction set includes a new family of six loop primitive instructions. These instructions decrement, 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.

B.8.4 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 64-Kbyte address space.

B.8.5 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.

B.8.6 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 the M68HC11 required about 250 bytes and executed in about 750 milliseconds. The CPU12 kernel uses about 50 bytes and executes in about 16 microseconds (in a 25-MHz HCS12).

B.8.7 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 of 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.

B.8.8 Extended Bit Manipulation

The M68HC11 CPU allows only 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.

B.8.9 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.

B.8.10 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.

B.8.11 Support for Memory Expansion

Bank switching is a common method of expanding memory beyond the 64-Kbyte limit of a CPU with a 64-Kbyte address space, but there are some known difficulties associated with bank switching. 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 HCS12 and M68HC12 variants include a built-in bank switching scheme that 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 HCS12 and 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 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.

Appendix C. High-Level Language Support

C.1 Introduction

Many programmers are turning to high-level languages such as C as an alternative to coding in native assembly languages. High-level language (HLL) programming can improve productivity and produce code that is more easily maintained than assembly language programs. The most serious drawback to the use of HLL in MCUs has been the relatively large size of programs written in HLL. Larger program ROM size requirements translate into increased system costs.

This appendix identifies CPU12 instructions and addressing modes that provide improved support for high-level language. C language examples are provided to demonstrate how these features support efficient HLL structures and concepts. Since the CPU12 instruction set is a superset of the M68HC11 instruction set, some of the discussions use the M68HC11 as a basis for comparison.

C.2 Data Types

The CPU12 supports the bit-sized data type with bit manipulation instructions which are available in extended, direct, and indexed variations. The char data type is a simple 8-bit value that is commonly used to specify variables in a small microcontroller system because it requires less memory space than a 16-bit integer (provided the variable has a range small enough to fit into eight bits). The 16-bit CPU12 can easily handle 16-bit integer types and the available set of conditional branches (including long branches) allow branching based on signed or unsigned arithmetic results. Some of the higher math functions allow for division and multiplication involving 32-bit values, although it is somewhat less common to use such long values in a microcontroller system.

The CPU12 has special sign extension instructions to allow easy type-casting from smaller data types to larger ones, such as from char to integer. This sign extension is automatically performed when an 8-bit value is transferred to a 16-bit register.

C.3 Parameters and Variables

High-level languages make extensive use of the stack, both to pass variables and for temporary and local storage. It follows that there should be easy ways to push and pull each CPU register, stack pointer based indexing should be allowed, and that direct arithmetic manipulation of the stack pointer value should be allowed. The CPU12 instruction set provided for all of these needs with improved indexed addressing, the addition of an LEAS instruction, and the addition of push and pull instructions for the D accumulator and the CCR.

C.3.1 Register Pushes and Pulls

The M68HC11 has push and pull instructions for A, B, X, and Y, but requires separate 8-bit pushes and pulls of accumulators A and B to stack or unstack the 16-bit D accumulator (the concatenated combination of A:B). The PSHD and PULD instructions allow directly stacking the D accumulator in the expected 16-bit order.

Adding PSHC and PULC improved orthogonality by completing the set of stacking instructions so that any of the CPU registers can be pushed or pulled. These instructions are also useful for preserving the CCR value during a function call subroutine.

C.3.2 Allocating and Deallocating Stack Space

The LEAS instruction can be used to allocate or deallocate space on the stack for temporary variables:

```
LEAS   -10,S    ;Allocate space for 5 16-bit integers
LEAS   10,S     ;Deallocate space for 5 16-bit ints
```

The (de)allocation can even be combined with a register push or pull as in this example:

```
LDX    8,S+     ;Load return value and deallocate
```

X is loaded with the 16-bit integer value at the top of the stack, and the stack pointer is adjusted up by eight to deallocate space for eight bytes worth of temporary storage. Post-increment indexed addressing is used in this example, but all four combinations of pre/post increment/decrement are available (offsets from –8 to +8 inclusive, from X, Y, or SP). This form of indexing can often be used to get an index (or stack pointer) adjustment for free during an indexed operation (the instruction requires no more code space or cycles than a zero-offset indexed instruction).

C.3.3 Frame Pointer

In the C language, it is common to have a frame pointer in addition to the CPU stack pointer. The frame is an area of memory within the system stack which is used for parameters and local storage of variables used within a function subroutine. The following is a description of how a frame pointer can be set up and used.

First, parameters (typically values in CPU registers) are pushed onto the system stack prior to using a JSR or CALL to get to the function subroutine. At the beginning of the called subroutine, the frame pointer of the calling program is pushed onto the stack. Typically, an index register, such as X, is used as the frame pointer, so a PSHX instruction would save the frame pointer from the calling program.

Next, the called subroutine establishes a new frame pointer by executing a TFR S,X. Space is allocated for local variables by executing an LEAS –n,S, where n is the number of bytes needed for local variables.

Notice that parameters are at positive offsets from the frame pointer while locals are at negative offsets. In the M68HC11, the indexed addressing mode uses only positive offsets, so the frame pointer always points to the lowest address of any parameter or local. After the function subroutine finishes, calculations are required to restore the stack pointer to the mid-frame position between the locals and the parameters before returning to the calling program. The CPU12 only requires execution of TFR X,S to deallocate the local storage and return.

The concept of a frame pointer is supported in the CPU12 through a combination of improved indexed addressing, universal transfer/exchange, and the LEA instruction. These instructions work together to achieve more efficient handling of frame pointers. It is important to consider the complete instruction set as a complex system with subtle interrelationships rather than simply examining individual instructions when trying to improve an instruction set. Adding or removing a single instruction can have unexpected consequences.

C.4 Increment and Decrement Operators

In C, the notation + + i or i – – is often used to form loop counters. Within limited constraints, the CPU12 loop primitives can be used to speed up the loop count and branch function.

The CPU12 includes a set of six basic loop control instructions which decrement, increment, or test a loop count register, and then branch if it is either equal to zero or not equal to zero. The loop count register can be A, B, D, X, Y, or SP. A or B could be used if the loop count fits in an 8-bit char variable; the other choices are all 16-bit registers. The relative offset for the loop branch is a 9-bit signed value, so these instructions can be used with loops as long as 256 bytes.

In some cases, the pre- or post-increment operation can be combined with an indexed instruction to eliminate the cost of the increment operation. This is typically done by post-compile optimization because the indexed instruction that could absorb the increment/decrement operation may not be apparent at compile time.

C.5 Higher Math Functions

In the CPU12, subtle characteristics of higher math operations such as IDIVS and EMUL are arranged so a compiler can handle inputs and outputs more efficiently.

The most apparent case is the IDIVS instruction, which divides two 16-bit signed numbers to produce a 16-bit result. While the same function can be accomplished with the EDIVS instruction (a 32 by 16 divide), doing so is much less efficient because extra steps are required to prepare inputs to the EDIVS, and because EDIVS uses the Y index register. EDIVS uses a 32-bit signed numerator and the C compiler would typically want to use a 16-bit value (the size of an integer data type). The 16-bit C value would need to be sign-extended into the upper 16 bits of the 32-bit EDIVS numerator before the divide operation.

Operand size is also a potential problem in the extended multiply operations but the difficulty can be minimized by putting the results in CPU registers. Having higher precision math instructions is not necessarily a requirement for supporting high-level language because these functions can be performed as library functions. However, if an application requires these functions, the code is much more efficient if the MCU can use native instructions instead of relatively large, slow routines.

C.6 Conditional If Constructs

In the CPU12 instruction set, most arithmetic and data manipulation instructions automatically update the condition code register, unlike other architectures that only change condition codes during a few specific compare instructions. The CPU12 includes branch instructions that perform conditional branching based on the state of the indicators in the condition codes register. Short branches use a single byte relative offset that allows branching to a destination within about ±128 locations from the branch. Long branches use a 16-bit relative offset that allows conditional branching to any location in the 64-Kbyte map.

C.7 Case and Switch Statements

Case and switch statements (and computed GOTOs) can use PC-relative indirect addressing to determine which path to take. Depending upon the situation, cases can use either the constant offset variation or the accumulator D offset variation of indirect indexed addressing.

C.8 Pointers

The CPU12 supports pointers by allowing direct arithmetic operations on the 16-bit index registers (LEAS, LEAX, and LEAY instructions) and by allowing indexed indirect addressing modes.

C.9 Function Calls

Bank switching is a fairly common way of adapting a CPU with a 16-bit address bus to accommodate more than 64 Kbytes of program memory space. One of the most significant drawbacks of this technique has been the requirement to mask (disable) interrupts while the bank page value was being changed. Another problem is that the physical location of the bank page register can change from one MCU derivative to another (or even due to a change to mapping controls by a user program). In these situations, an operating system program has to keep track of the physical location of the page register. The CPU12 addresses both of these problems with the uninterruptible CALL and return-from-call (RTC) instructions.

The CALL instruction is similar to a JSR instruction, except that the programmer supplies a destination page value as part of the instruction. When CALL executes, the old page value is saved on the stack and the new page value is written to the bank page register. Since the CALL instruction is uninterruptible, this eliminates the need to separately mask off interrupts during the context switch.

The CPU12 has dedicated signal lines that allow the CPU to access the bank page register without having to use an address in the normal 64-Kbyte address space. This eliminates the need for the program to know where the page register is physically located.

The RTC instruction is similar to the RTS instruction, except that RTC uses the byte of information that was saved on the stack by the corresponding CALL instruction to restore the bank page register to its old value. Although a CALL/RTC pair can be used to access any function subroutine regardless of the location of the called routine (on the current bank page or a different page), it is most efficient to access some subroutines with JSR/RTS instructions when the called subroutine is on the current page or in an area of memory that is always visible in the 64-Kbyte map regardless of the bank page selection.

Push and pull instructions can be used to stack some or all the CPU registers during a function call. The CPU12 can push and pull any of the CPU registers A, B, CCR, D, X, Y, or SP.

C.10 Instruction Set Orthogonality

One helpful aspect of the CPU12 instruction set, orthogonality, is difficult to quantify in terms of direct benefit to an HLL compiler. Orthogonality refers to the regularity of the instruction set. A completely orthogonal instruction set would allow any instruction to operate in any addressing mode, would have identical code sizes and execution times for similar operations on different registers, and would include both signed and unsigned versions of all mathematical instructions. Greater regularity of the instruction set makes it possible to implement compilers more efficiently, because operation is more consistent, and fewer special cases must be handled.

Index

A

ABA instruction 100
Abbreviations for system resources 16
ABX instruction 101
ABY instruction 102
Access details 94–99, 379
Accumulator offset indexed addressing mode 41
Accumulator offset indexed indirect addressing mode 40
Accumulators 22
  A 21, 35
  B 21, 35
  D 21, 35
ADCA instruction 103
ADCB instruction 104
ADDA instruction 105
ADDB instruction 106
ADDD instruction 107
Addition instructions 59
ADDR mnemonic 19
Addressing modes 29
  Direct 32
  Extended 33
  Immediate 31
  Indexed 22, 34
  Inherent 31
  Relative 33
ANDA instruction 108
ANDB instruction 109
ANDCC instruction 110
Arithmetic shift 115
ASL instruction 111
ASLA instruction 112
ASLB instruction 113
ASLD instruction 114

ASR instruction . . . 115
ASRA instruction . . . 116
ASRB instruction . . . 117
Asserted . . . 19
Auto increment . . . 39

B

Background debug mode . . . 86
  Instruction . . . 86, 123
Base index register . . . 37–41
BCC instruction . . . 118
BCD instructions . . . 60, 161
BCLR instruction . . . 119
BCS instruction . . . 120
BEQ instruction . . . 121
BGE instruction . . . 122
BGND instruction . . . 86, 123
BGT instruction . . . 124
BHI instruction . . . 125
BHS instruction . . . 126
Binary-coded decimal instructions . . . 60, 161
Bit manipulation instructions . . . 65, 119, 140, 423, 425
  Mask operand . . . 43, 119, 137, 139, 140
  Multiple addressing modes . . . 43
Bit test instructions . . . 65, 76, 127, 128, 137, 139
BITA instruction . . . 127
BITB instruction . . . 128
Bit-condition branches . . . 76, 137, 139
BLE instruction . . . 129
BLO instruction . . . 130
BLS instruction . . . 131
BLT instruction . . . 132
BMI instruction . . . 133
BNE instruction . . . 134
Boolean logic instructions . . . 63
  AND . . . 108, 109, 110
  Complement . . . 154, 155, 156
  Exclusive OR . . . 179, 180
  Inclusive OR . . . 243, 244, 245
  Negate . . . 239, 240, 241
BPL instruction . . . 135
BRA instruction . . . 136
Branch instructions . . . 33, 51–53, 73, 429
  Bit-condition . . . 53, 76, 137, 139

Long 52, 53, 75, 421
Loop primitive 53, 77, 400
Offset values 74, 75, 76, 77
Offsets 34
Short 52, 53, 74
Signed 73–75
Simple 73–75
Subroutine 78, 141
Summary of complementary branches 118, 196
Taken/not-taken cases 52, 99
Unary 73–75
Unsigned 73–75
Branch offset 33–34
BRCLR instruction 137
BRN instruction 138
BRSET instruction 139
BSET instruction 140
BSR instruction 50, 141
Bus cycles 95
Bus structure 407
BVC instruction 142
BVS instruction 143
Byte moves 58, 236
Byte order in memory 27
Byte-sized instructions 52

C

C 111
C status bit 26, 66, 118, 120
CALL instruction . . . . . 44–??, 50, 78, 144, 424, 430
Case statements 429
CBA instruction 145
CCR *(see Condition codes register)*
Changes in execution flow 49–53
CLC instruction 146
Clear instructions 63
Clear memory 148
Cleared 19
CLI instruction 147
Clock monitor reset 314
CLR instruction 148
CLRA instruction 149
CLRB instruction 150
CLV instruction 151

CMPA instruction 152
CMPB instruction 153
Code size. 416
COM instruction 154
COMA instruction 155
COMB instruction 156
Compare instructions. 62
Complement instructions 63
Computer operating properly (COP) watchdog. 314
Condition codes instructions 84, 110, 245, 248, 254, 291, 297, 405, 423
Condition codes register 21, 23–27
- C status bit 26, 66, 118, 120
- H status bit 25, 161
- I mask bit 25, 110, 147, 276, 306, 313, 315
- Manipulation 84, 110, 245, 276
- N status bit 26
- S control bit 24, 282
- V status bit. 26
- X mask bit 25, 182, 254, 270, 282, 291, 296, 306, 313, 315
- Z status bit. 26, 121, 134

Conditional 16-bit read cycle 98, 379
Conditional 8-bit read cycle 98, 379
Conditional 8-bit write cycle 98, 379
Conserving power 85, 282, 306
Constant indirect indexed addressing mode 38
Constant offset indexed addressing mode. 37, 38
COP reset 314
CPD instruction. 157
CPS instruction. 158
CPX instruction. 159
CPY instruction. 160
Cycle code letters. 95, 379
Cycle counts 415
Cycle-by-cycle operation. 94, 379

D

DAA instruction. 161
DATA mnemonic 19
Data types 27, 425
DBEQ instruction 162, 400
DBNE instruction 163, 400
DEC instruction. 164
DECA instruction 165
DECB instruction 166

Decrement instructions . . . 61
Defuzzification . . . 344, 364–368
DES instruction . . . 167
DEX instruction . . . 168
DEY instruction . . . 169
Direct addressing mode . . . 32
Division instructions . . . 64, 428
  16-bit fractional . . . 183
  16-bit integer . . . 186, 187
  32-bit extended . . . 170, 171
Double accumulator . . . 21, 22

E

EDIV instruction . . . 170
EDIVS instruction . . . 171
Effective address . . . 29, 35, 83, 220, 221, 222, 417, 426–428
EMACS instruction . . . 71, 172
EMAXD instruction . . . 173
EMAXM instruction . . . 174, 338
EMIND instruction . . . 175, 338
EMINM instruction . . . 176
EMUL instruction . . . 177
EMULS instruction . . . 178
Enabling maskable interrupts . . . 25, 147
EORA instruction . . . 179
EORB instruction . . . 180
ETBL instruction . . . 72, 181, 338
Even bytes . . . 27
Exceptions . . . 50, 311
  Interrupts . . . 315
  Maskable interrupts . . . 315, 317
  Non-maskable interrupts . . . 315
  Priority . . . 312
  Processing flow . . . 318
  Resets . . . 311, 313–314
  Software interrupts . . . 79, 289, 318
  Unimplemented opcode trap . . . 311, 313, 317
  Vectors . . . 311, 318
Exchange instructions . . . 57, 182, 417, 420
  Postbyte encoding . . . 399
Execution cycles . . . 95
Execution time . . . 94
EXG instruction . . . 182
Expanded memory . . . 44, 50, 424, 430

Bank switching . . . 44
Instructions . . . 44, 78, 144, 269
Page registers . . . 44
Subroutines . . . 78, 430
Extended addressing mode . . . 33
Extended division . . . 64
Extension byte . . . 34
External interrupts . . . 317
External queue reconstruction . . . 323
HCS12 queue reconstruction . . . 328
HCS12 reconstruction algorithm . . . 330
HCS12 timing detail . . . 324
M68HC12 queue reconstruction . . . 331
M68HC12 reconstruction algorithm . . . 332
M68HC12 timing detail . . . 325
External reset . . . 314

F

Fast math . . . 415
f-cycle (free cycle) . . . 95, 379
FDIV instruction . . . 64, 183
Fractional division . . . 64, 183
Frame pointer . . . 427, 428
Free cycle . . . 95, 379
Fuzzy logic . . . 337–374
Antecedents . . . 342, 372
Consequents . . . 342, 372
Custom programming . . . 369
Defuzzification . . . 68, 344, 364–367
Fuzzification . . . 67, 340, 369
Inference kernel . . . 339, 345
Inputs . . . 372
Instructions . . . 67, 68, 233, 258–262, 307, 337, 347–367, 422
Interrupts . . . 361, 365–368
Knowledge base . . . 340, 342, 372
Membership functions . . . 67, 233, 338, 340, 347–352, 369–371
Outputs . . . 68, 372
Rule evaluation . . . 67, 258–262, 342, 353–363, 372
Rules . . . 340, 342, 372
Sets . . . 339
Tabular membership functions . . . 72, 369
Weighted average . . . 68, 307, 337, 344, 364–367

G

g-cycle (read PPAGE) 95, 379
General purpose accumulators 21
Global interrupt mask 26, 313

H

H status bit 25, 161
Highest priority interrupt 313
High-level language 425–431
  Addressing modes 425, 427, 429
  Condition codes register 429
  Expanded memory 430
  Instructions 425
  Loop primitives 428
  Stack 426, 427

I

I mask bit 25, 110, 147, 276, 313
IBEQ instruction 184, 400
IBNE instruction 185, 400
I-cycle (16-bit read indirect) 95, 379
i-cycle (8-bit read indirect) 95, 379
IDIV instruction 186
IDIVS instruction 187, 428
Immediate addressing mode 31
INC instruction 188
INCA instruction 189
INCB instruction 190
Increment instructions 61
Index calculation instructions 83, 417
Index manipulation instructions 81
Index registers 21, 81, 83, 427
  PC (as an index register) 23, 36, 37, 94
  SP (as an index register) 22, 36, 37, 94
  X 22, 36, 94
  Y 22, 36, 94
Indexed addressing modes 22, 34–43, 397, 410–414
  16-bit constant indirect 38
  16-bit constant offset 38
  5-bit constant offset 37
  9-bit constant offset 37
  Accumulator direct 41

Accumulator offset 40
Auto increment/decrement indexing 39
Base index register 37–41
Extension byte 34
Limitations for BIT and MOV instructions 119, 137, 139, 140, 236, 237
Postbyte 36
Postbyte encoding 34, 397

Inference kernel, fuzzy logic 345

Inherent addressing mode 31

INS instruction 191

Instruction pipe, see *Instruction queue*

Instruction queue 28, 47, 323, 408
Buffer 48
Data movement 48
Debugging 323
Reconstruction 323–334
Stages 48, 323
Status registers 329, 332
Status signals 48, 324–333

Instruction set 55, 87, 381

Integer division 64, 186–187

Interrupt instructions 79

Interrupts 315–319
Enabling and disabling 25, 147, 276, 315
External 317
I mask bit 25, 147, 276, 316
Instructions 79, 80, 147, 270, 276, 289, 298
Low-power stop 85, 282
Maskable 25, 315
Non-maskable 25, 311–313, 315
Recognition 316
Return 25, 26, 80, 270, 317
Service routines 316
Software 79, 289, 318
Stacking order 316, 376
Vectors 311, 317, 318
Wait instruction 85, 306
X mask bit 25, 282, 306, 316

INX instruction 192

INY instruction 193

J

JMP instruction 53, 194

JSR instruction 50, 195

Jump instructions 53, 78

K

Knowledge base 340

L

Label 91
LBCC instruction 196
LBCS instruction 197
LBEQ instruction 198
LBGE instruction 199
LBGT instruction 200
LBHI instruction 201
LBHS instruction 202
LBLE instruction 203
LBLO instruction 204
LBLS instruction 205
LBLT instruction 206
LBMI instruction 207
LBNE instruction 208
LBPL instruction 209
LBRA instruction 210
LBRN instruction 211
LBVC instruction 212
LBVS instruction 213
LDAA instruction 214
LDAB instruction 215
LDD instruction 216
LDS instruction 217
LDX instruction 218
LDY instruction 219
LEAS instruction 220, 427, 429
Least significant byte 19
Least significant word 19
LEAX instruction 221, 429
LEAY instruction 222, 429
Legal label 91
Literal expression 91
Load instructions 56
Logic level one 19
Logic level zero 19
Loop primitive instructions 53, 77, 400, 420, 428
    Offset values 77

Postbyte encoding 400
Low-power stop 85, 282
LSL instruction 66, 223
LSLA instruction 224
LSLB instruction 225
LSLD instruction 226
LSR instruction 227
LSRA instruction 228
LSRB instruction 229
LSRD instruction 230

M

M68HC11 compatibility 29, 403–424
M68HC11 instruction mnemonics 404
Maskable interrupts 25, 315
MAXA instruction 231
Maximum instructions 70, 421
16-bit 173, 174
8-bit 231, 232
MAXM instruction 232, 338
MEM instruction 67, 233, 337, 347–352
Membership functions 340, 347–352
Memory and addressing symbols 17
MINA instruction 234, 338
Minimum instructions 70, 421
16-bit 175, 176
8-bit 234, 235
MINM instruction 235
Misaligned instructions 52
Mnemonic 88
Most significant byte 19
Most significant word 19
MOVB instruction 236
Move instructions 58, 236, 237, 417, 420
Destination 41
Multiple addressing modes 41
PC relative addressing 41
Reference index register 41
Source 41
MOVW instruction 237
MUL instruction 238
Multiple addressing modes
Bit manipulation instructions 43
Move instructions 41

Multiplication instructions . . . 64
16-bit . . . 177, 178
8-bit. . . . 238
Multiply and accumulate instructions . . . 71, 172, 307, 373

N

N status bit . . . 26
n-cycle (write PPAGE). . . . 96, 379
NEG instruction . . . 239
NEGA instruction . . . 240
Negate instructions . . . 63
Negated . . . 19
Negative integers . . . 27
NEGB instruction . . . 241
Non-maskable interrupts . . . 25, 313, 315
NOP instruction . . . 86, 242
Notation
Branch taken/not taken . . . 99, 379
Changes in CCR bits . . . 89
Cycle-by-cycle operation. . . . 94
Memory and addressing . . . 17
Object code . . . 90
Operators. . . . 18, 377
Source forms. . . . 91
System resources . . . 16
Null operation instruction . . . 86, 242
Numeric range of branch offsets . . . 34, 74–77

O

Object code notation . . . 90
O-cycle (optional program word fetch) . . . 52, 96, 379
Odd bytes . . . 27
Offset
Branch . . . 33–34
Index. . . . 34–38
Opcode map . . . 395–396
Operators. . . . 18, 377
Optional cycles . . . 52, 96, 379
ORAA instruction . . . 243
ORAB instruction . . . 244
ORCC instruction . . . 245
Orthogonality. . . . 431

P

Page 2 prebyte 52, 96, 396
P-cycle (program word fetch) 96, 379
Pipeline 28
Pointer calculation instructions 83, 220, 221, 222
Pointers 429
Postbyte encoding
  Exchange instructions 182, 399
  Indexed addressing instructions 36
  Indexed addressing modes 36, 397
  Loop primitive instructions 400
  Transfer instructions 278, 296, 399
Post-decrement indexed addressing mode 39
Post-increment indexed addressing mode. 39
Power conservation 85, 282, 306
Power-on reset 314
Prebyte. 52, 96, 396
Pre-decrement indexed addressing mode 39
Pre-increment indexed addressing mode 39
Priority, exception. 312
Program counter. 21, 23, 35, 123
Program word access cycle 96, 379
Programming model 15, 21, 407
Pseudo-non-maskable interrupt. 313
PSHA instruction 246
PSHB instruction 247
PSHC instruction 248
PSHD instruction 249, 426
PSHX instruction 250
PSHY instruction 251
PULA instruction. 252
PULB instruction. 253
PULC instruction 254, 426
PULD instruction 255, 426
Pull instructions 430
PULX instruction. 256
PULY instruction. 257
Push instructions 430

Q

Queue reconstruction 323
  HCS12 queue reconstruction. 328
  HCS12 reconstruction algorithm 330

HCS12 timing detail . . . 324
M68HC12 queue reconstruction . . . 331
M68HC12 reconstruction algorithm . . . 332
M68HC12 timing detail . . . 325

R

R-cycle (16-bit data read) . . . 97, 379
r-cycle (8-bit data read) . . . 97, 379
Read 16-bit data cycle . . . 97, 379
Read 8-bit data cycle . . . 97, 379
Read indirect pointer cycle . . . 95, 379
Read indirect PPAGE value cycle . . . 95, 379
Read PPAGE cycle . . . 95, 379
Register designators . . . 91
Relative addressing mode . . . 33
Relative offset . . . 33
Resets . . . 311, 313
Clock monitor . . . 314
COP . . . 314
External . . . 314
Power-on . . . 314
Return from call . . . 269
Return from interrupt . . . 270
Return from subroutine . . . 271
REV instruction . . . 67, 258–259, 337, 342, 353–358, 372
REVW instruction . . . 67, 260–262, 337, 342, 359–363, 372
ROL instruction . . . 263
ROLA instruction . . . 264
ROLB instruction . . . 265
ROR instruction . . . 266
RORA instruction . . . 267
RORB instruction . . . 268
Rotate instructions . . . 66
RTC instruction . . . 44, 50, 78, 269, 424, 430
RTI instruction . . . 26, 80, 270, 317
RTS instruction . . . 51, 271

S

S control bit . . . 24, 282
SBA instruction . . . 272
SBCA instruction . . . 273
SBCB instruction . . . 274
S-cycle (16-bit stack write) . . . 97, 379

s-cycle (8-bit stack write) . . . 97, 379
SEC instruction . . . 275
SEI instruction . . . 276
Service routine . . . 311
Set . . . 19
Setting memory bits . . . 140
SEV instruction . . . 277
SEX instruction . . . 57, 278
Shift instructions . . . 66
    Arithmetic . . . 115
Sign extension instruction . . . 57, 278, 426
Signed branches . . . 73–75
Signed integers . . . 27
Signed multiplication . . . 64
Simple branches . . . 73–75
Software interrupts . . . 289
Source code compatibility . . . 15, 404
Source form notation . . . 91
STAA instruction . . . 279
STAB instruction . . . 280
Stack . . . 22, 409, 410
Stack 16-bit data cycle . . . 97, 379
Stack 8-bit data cycle . . . 97, 379
Stack operation instructions . . . 82
Stack pointer . . . 21, 22, 35, 426
    Compatibility with HC11 . . . 409–410
    Initialization . . . 22, 410
    Manipulation . . . 82
    Stacking order . . . 316, 376
Stack pointer instructions . . . 82, 423, 426
Standard CPU12 address space . . . 27
STD instruction . . . 281
STOP continue . . . 282
STOP disable . . . 24, 282
STOP instruction . . . 24, 85, 282
Store instructions . . . 56
STS instruction . . . 283
STX instruction . . . 284
STY instruction . . . 285
SUBA instruction . . . 286
SUBB instruction . . . 287
SUBD instruction . . . 288
Subroutine instructions . . . 78
Subroutines . . . 50, 430
    Expanded memory . . . 50, 78, 144, 269, 430

Instructions . . . 78, 141, 144, 195, 430
Return . . . 269, 271
Subtraction instructions . . . 59
SWI instruction . . . 79, 289, 318
Switch statements . . . 429
Symbols and notation . . . 16, 377

T

TAB instruction . . . 290
Table interpolation instructions . . . 72, 181, 294, 422
Tabular membership functions . . . 369–371
TAP instruction . . . 291
TBA instruction . . . 292
TBEQ instruction . . . 293, 400
TBL instruction . . . 72, 294, 338, 369–370
TBNE instruction . . . 295, 400
T-cycle (16-bit conditional read) . . . 98, 379
t-cycle (8-bit conditional read) . . . 98, 379
Termination of interrupt service routines . . . 80, 270, 317
Termination of subroutines . . . 269, 271
Test instructions . . . 62
TFR instruction . . . 296
TPA instruction . . . 297
Transfer instructions . . . 57, 417, 420
Postbyte encoding . . . 399
TRAP instruction . . . 79, 298, 317, 396
TST instruction . . . 299
TSTA instruction . . . 300
TSTB instruction . . . 301
TSX instruction . . . 302
TSY instruction . . . 303
Twos-complement form . . . 27
TXS instruction . . . 304
Types of instructions
Addition and Subtraction . . . 59
Background and null . . . 86
Binary-coded decimal . . . 60
Bit test and manipulation . . . 65
Boolean logic . . . 63
Branch . . . 73
Clear, complement, and negate . . . 63
Compare and test . . . 62
Condition code . . . 84
Decrement and increment . . . 61

- Fuzzy logic . . . 67
- Index manipulation . . . 81
- Interrupt . . . 79
- Jump and subroutine . . . 78
- Load and store . . . 56
- Loop primitives . . . 77
- Maximum and minimum . . . 70
- Move . . . 58
- Multiplication and division . . . 64
- Multiply and accumulate . . . 71
- Pointer and index calculation . . . 83
- Shift and rotate . . . 66
- Sign extension . . . 57
- Stacking . . . 82
- Stop and wait . . . 85
- Table interpolation . . . 72
- Transfer and exchange . . . 57

TYS instruction . . . 305

U

U-cycle (16-bit stack read) . . . 98, 379
u-cycle (8-bit stack read) . . . 97, 379
Unary branches . . . 73–75
Unimplemented opcode trap . . . 79, 298, 311, 313, 396
Unsigned branches . . . 73–75
Unsigned multiplication . . . 64
Unstack 16-bit data cycle . . . 98, 379
Unstack 8-bit data cycle . . . 97, 379
Unweighted rule evaluation . . . 258–259, 342, 353–358, 372

V

V status bit . . . 26, 84
V-cycle (vector fetch) . . . 98, 379
Vector fetch cycle . . . 98, 379
Vectors, exception . . . 311, 318

W

WAI instruction . . . 85, 306
Wait instruction . . . 85, 306
Watchdog . . . 314
WAV instruction . . . 68, 307, 337, 344, 364–368

  - HCS12 . . . 366
  - M68HC12 . . . 367
- wavr pseudo-instruction . . . 365–368
  - HCS12 . . . 366
  - M68HC12 . . . 367
- W-cycle (16-bit data write) . . . 97, 379
- w-cycle (8-bit data write) . . . 97, 379
- Weighted average . . . 307
- Weighted rule evaluation . . . 260–262, 342, 353–355, 359–363, 372
- Word moves . . . 58, 237
- Write 16-bit data cycle . . . 97, 379
- Write 8-bit data cycle . . . 97, 379
- Write PPAGE cycle . . . 96, 379

X

- X mask bit . . . 25, 182, 254, 270, 282, 291, 296, 306
- x-cycle (8-bit conditional write) . . . 98, 379
- XGDX instruction . . . 308
- XGDY instruction . . . 309

Z

- Z status bit . . . 26, 121, 134
- Zero-page addressing . . . 32

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.