MOTOROLA
intelligence everywhere™

digital dna™

S12XCPUV1

Reference Manual

HCS12 & HCS12X Microcontrollers

S12XCPUV1

February 5, 2004

WWW.MOTOROLA.COM/SEMICONDUCTORS

S12XCPUV1

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://motorola.com/semiconductors

Revision History

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

Revision History

Revision Number	Date	Author	Summary of changes
v01.00	5 Feb 2004		Initial version

List of Sections

Section 1. Introduction 19

Section 2. Overview 25

Section 3. Addressing Modes 35

Section 4. Instruction Queue 53

Section 5. Instruction Set Overview 61

Section 6. Instruction Glossary 95

Section 7. Exception Processing. 405

Section 8. Instruction Queue 419

Section 9. Fuzzy Logic Support. 433

Appendix A. Instruction Reference 473

Appendix B. High-Level Language Support 511

List of Sections

Table of Contents

Section 1. Introduction

1.1 Introduction 19
1.2 Features 19
1.3 Symbols and Notation 20
1.3.1 Abbreviations for System Resources 20
1.3.2 Memory and Addressing 21
1.3.3 Operators 22
1.3.4 Definitions 23

Section 2. Overview

2.1 Introduction 25
2.2 Programming Model 25
2.2.1 Accumulators 27
2.2.2 Index Registers 27
2.2.3 Stack Pointer 27
2.2.4 Program Counter 28
2.2.5 Condition Code Register 28
2.2.5.1 **IPL[2:0]** **30**
2.2.5.2 S Control Bit 30
2.2.5.3 **X Mask Bit** **30**
2.2.5.4 **H Status Bit** **31**
2.2.5.5 **I Mask Bit** **32**
2.2.5.6 **N Status Bit** **32**
2.2.5.7 **Z Status Bit** **32**
2.2.5.8 **V Status Bit** **33**
2.2.5.9 **C Status Bit** **33**
2.3 Data Types 33
2.4 Memory Organization 34
2.5 Instruction Queue 34

Section 3. Addressing Modes

3.1 Introduction 35
3.2 Mode Summary 35
3.3 Effective Address 35
3.4 Inherent Addressing Mode 36
3.5 Immediate Addressing Mode 37
3.6 Direct Addressing Mode (HCS12) 38
3.7 Direct Addressing Mode (HCS12X) 38
3.8 Extended Addressing Mode 39
3.9 Relative Addressing Mode 39
3.10 Indexed Addressing Modes 40
3.10.1 5-Bit Constant Offset Indexed Addressing 43
3.10.2 9-Bit Constant Offset Indexed Addressing 43
3.10.3 16-Bit Constant Offset Indexed Addressing 44
3.10.4 16-Bit Constant Indirect Indexed Addressing 44
3.10.5 Auto Pre/Post Decrement/Increment Indexed Addressing . . 45
3.10.6 Accumulator Offset Indexed Addressing 46
3.10.7 Accumulator D Indirect Indexed Addressing 47
3.11 Global Addressing (HCS12X only) 48
3.12 Instructions Using Multiple Modes 48
3.12.1 Move Instructions 48
3.12.2 Bit Manipulation Instructions 49
3.13 Addressing More than 64 Kbytes 50

Section 4. Instruction Queue

4.1 Introduction 53
4.2 Queue Description 53
4.2.1 HCS12 and HCS12X Queue Implementation 54
4.3 Data Movement in the Queue 54
4.3.1 No Movement 54
4.3.2 **Advance and Load from Data Bus 54**
4.4 Changes in Execution Flow 55
4.4.1 Exceptions 55
4.4.2 Subroutines 56

4.4.3 Branches 56
4.4.3.1 Short Branches 57
4.4.3.2 Long Branches 57
4.4.3.3 Bit Condition Branches 58
4.4.3.4 Loop Primitives 58
4.4.4 Jumps 59

Section 5. Instruction Set Overview

5.1 Introduction 61
5.2 Instruction Set Description 61
5.3 Load and Store Instructions 62
5.4 Transfer and Exchange Instructions 63
5.5 Move Instructions 65
5.6 Addition and Subtraction Instructions 66
5.7 Binary-Coded Decimal Instructions 67
5.8 Decrement and Increment Instructions 68
5.9 Compare and Test Instructions 69
5.10 Boolean Logic Instructions 70
5.11 Clear, Complement, and Negate Instructions 71
5.12 Multiplication and Division Instructions 72
5.13 Bit Test and Manipulation Instructions 73
5.14 Shift and Rotate Instructions 74
5.15 Fuzzy Logic Instructions 75
5.15.1 Fuzzy Logic Membership Instruction 75
5.15.2 Fuzzy Logic Rule Evaluation Instructions 75
5.15.3 Fuzzy Logic Weighted Average Instruction 76
5.16 Maximum and Minimum Instructions 78
5.17 Multiply and Accumulate Instruction 78
5.18 Table Interpolation Instructions 79
5.19 Branch Instructions 80
5.19.1 Short Branch Instructions 80
5.19.2 Long Branch Instructions 81
5.19.3 Bit Condition Branch Instructions 83

5.20 Loop Primitive Instructions 84
5.21 Jump and Subroutine Instructions 85
5.22 Interrupt Instructions 86
5.23 Index Manipulation Instructions 88
5.24 Stacking Instructions. 89
5.25 Pointer and Index Calculation Instructions 90
5.26 Condition Code Instructions 91
5.27 Stop and Wait Instructions 92
5.28 Background Mode and Null Operations 93

Section 6. Instruction Glossary

6.1 Introduction 95
6.2 Glossary Information. 95
6.3 Condition Code Changes 97
6.4 Object Code Notation 98
6.5 Source Forms 99
6.6 Cycle-by-Cycle Execution. 102
6.7 Glossary 107

Section 7. Exception Processing

7.1 Introduction 405
7.2 Types of Exceptions 405
7.3 Exception Priority 407
7.4 Resets. 409
7.4.1 Power-On Reset. 409
7.4.2 External Reset 409
7.4.3 COP Reset 410
7.4.4 Clock Monitor Reset 410
7.5 Interrupts. 410
7.5.1 Non-Maskable Interrupt Request ($\overline{\text{XIRQ}}$) 410
7.5.2 Maskable Interrupts 411
7.5.3 Interrupt Recognition 411
7.5.4 External Interrupts 412

7.5.5 Return-from-Interrupt Instruction (RTI) 413
7.6 Unimplemented Opcode Trap. 413
7.7 Software Interrupt Instruction (SWI) 414
7.8 Exception Processing Flow. 414
7.8.1 Vector Fetch 414
7.8.2 Reset Exception Processing 415
7.8.3 Interrupt and Unimplemented Opcode Trap Exception Processing 417

Section 8. Instruction Queue

8.1 Introduction 419
8.2 External Reconstruction of the Queue 419
8.3 Instruction Queue Status Signals 420
8.3.1 HCS12 Timing Detail 421
8.3.2 HCS12X Timing Detail 423
8.3.3 **Null. 424**
8.3.4 **ALD — Advance and Load from Data Bus 425**
8.3.5 **INT —** Interrupt Sequence Start. 425
8.3.6 **SEV — Start Instruction on Even Address 425**
8.3.7 **SOD — Start Instruction on Odd Address 425**
8.4 Queue Reconstruction (for HCS12) 425
8.4.1 Queue Reconstruction Registers (for HCS12) 426
8.4.1.1 **fetch_add Register. 426**
8.4.1.2 **st1_add, st1_dat Registers 427**
8.4.1.3 st2_add, st2_dat Registers 427
8.4.1.4 **st3_add, st3_dat Registers 427**
8.4.2 Reconstruction Algorithm (for HCS12X). 427
8.5 Instruction Tagging (HCS12) 428
8.6 Instruction Tagging (HCS12X) 430

Section 9. Fuzzy Logic Support

9.1 Introduction 433
9.2 Fuzzy Logic Basics 434
9.2.1 Fuzzification (MEM) 436
9.2.2 Rule Evaluation (REV and REVW). 438

9.2.3 Defuzzification (WAV) 440
9.3 Example Inference Kernel 441
9.4 MEM Instruction Details 444
9.4.1 Membership Function Definitions 444
9.4.2 Abnormal Membership Function Definitions 446
9.4.2.1 Abnormal Membership Function Case 1 448
9.4.2.2 Abnormal Membership Function Case 2 449
9.4.2.3 Abnormal Membership Function Case 3 449
9.5 REV and REVW Instruction Details 450
9.5.1 Unweighted Rule Evaluation (REV) 450
9.5.1.1 Set Up Prior to Executing REV 450
9.5.1.2 Interrupt Details 452
9.5.1.3 Cycle-by-Cycle Details for REV 452
9.5.2 Weighted Rule Evaluation (REVW) 456
9.5.2.1 Set Up Prior to Executing REVW 456
9.5.2.2 Interrupt Details 458
9.5.2.3 Cycle-by-Cycle Details for REVW 458
9.6 WAV Instruction Details 461
9.6.1 Set Up Prior to Executing WAV 462
9.6.2 WAV Interrupt Details 462
9.6.3 Cycle-by-Cycle Details for WAV and wavr 463
9.7 Custom Fuzzy Logic Programming 467
9.7.1 Fuzzification Variations 467
9.7.2 Rule Evaluation Variations 470
9.7.3 Defuzzification Variations 471

Appendix A. Instruction Reference

A.1 Introduction 473
A.2 Stack and Memory Layout 474
A.3 Interrupt Vector Locations 475
A.4 Notation Used in Instruction Set Summary 475
A.5 Hexadecimal to Decimal Conversion 509
A.6 Decimal to Hexadecimal Conversion 509

Appendix B. High-Level Language Support

B.1 Introduction 511
B.2 Data Types 511
B.3 Parameters and Variables 512
B.3.1 Register Pushes and Pulls 512
B.3.2 Allocating and Deallocating Stack Space 513
B.3.3 Frame Pointer 513
B.4 Increment and Decrement Operators 514
B.5 Higher Math Functions 514
B.6 Conditional If Constructs 515
B.7 Case and Switch Statements 515
B.8 Pointers 516
B.9 Function Calls 516
B.10 Instruction Set Orthogonality 517

Table of Contents

List of Figures

Figure	Title	Page
2-1	. HCS12 Programming Model	26
2-2	. HCS12X Programming Model	26
6-1	. Example Glossary Page	96
7-1	. Exception Processing Flow Diagram	416
8-1	. Queue Status Signal Timing (HCS12)	421
8-2	. Queue Status Signal Timing (HCS12X)	423
8-3	. Tag Input Timing (HCS12)	428
8-4	. Tag Input Timing (HCS12X)	430
9-1	. Block Diagram of a Fuzzy Logic System	435
9-2	. Fuzzification Using Membership Functions	437
9-3	. Fuzzy Inference Engine	442
9-4	. Defining a Normal Membership Function	445
9-5	. MEM Instruction Flow Diagram	447
9-6	. Abnormal Membership Function Case 1	448
9-7	. Abnormal Membership Function Case 2	449
9-8	. Abnormal Membership Function Case 3	449
9-9	. REV Instruction Flow Diagram	453
9-10	. REVW Instruction Flow Diagram	460
9-11	. WAV and wavr Instruction Flow Diagram (for HCS12)	465
9-12	. WAV and wavr Instruction Flow Diagram (for M68HC12)	466
9-13	. Endpoint Table Handling	469

List of Figures

List of Tables

Table	Title	Page
3-1	. HCS12 & HCS12X Addressing Mode Summary	36
3-2	. Summary of Indexed Operations	42
5-1	. Load and Store Instructions	62
5-2	. Transfer and Exchange Instructions	64
5-3	. Move Instructions	65
5-4	. Addition and Subtraction Instructions	66
5-5	. BCD Instructions	67
5-6	. Decrement and Increment Instructions	68
5-7	. Compare and Test Instructions	69
5-8	. Boolean Logic Instructions	70
5-9	. Clear, Complement, and Negate Instructions	71
5-10	. Multiplication and Division Instructions	72
5-11	. Bit Test and Manipulation Instructions	73
5-12	. Shift and Rotate Instructions	74
5-13	. Fuzzy Logic Instructions	76
5-14	. Minimum and Maximum Instructions	78
5-15	. Multiply and Accumulate Instructions	79
5-16	. Table Interpolation Instructions	80
5-17	. Short Branch Instructions	81
5-18	. Long Branch Instructions	82
5-19	. Bit Condition Branch Instructions	83
5-20	. Loop Primitive Instructions	84
5-21	. Jump and Subroutine Instructions	86
5-22	. Interrupt Instructions	87
5-23	. Index Manipulation Instructions	88
5-24	. Stacking Instructions	89
5-25	. Pointer and Index Calculation Instructions	90
5-26	. Condition Code Instructions	91
5-27	. Stop and Wait Instructions	92

List of Tables

5-28 . Background Mode and Null Operation Instructions 93
7-1 . CPU12 Exception Vector Map 407
7-2 . HCS12 Stacking Order on Entry to Interrupts 412
7-3 . HCS12X Stacking Order on Entry to Interrupts 412
8-1 . IPIPE1 and IPIPE0 Decoding HCS12 422
8-2 . IQSTAT[3:0] on HCS12X 424
8-3 . Tag Pin Function (HCS12) 428
8-4 . Tag Pin Function (HCS12X) 430
A-1 . Instruction Set Summary 479
A-2 . Opcode Map (Sheet 1 of 3) — HCS12 and HCS12X Page 1 Opcodes500
A-3 . Indexed Addressing Mode Postbyte Encoding (xb) 503
A-4 . Indexed Addressing Mode Summary 504
A-5 . Transfer and Exchange Postbyte Encoding 505
A-6 . Loop Primitive Postbyte Encoding (lb) 507
A-7 . Branch/Complementary Branch 507
A-8 . Hexadecimal to ASCII Conversion 508
A-9 . Hexadecimal to/from Decimal Conversion 509

Section 1. Introduction

1.1 Introduction

This manual describes the features and operation of the central processing unit, or CPU12, used in HCS12 and HCS12X microcontrollers. 68HC12, HCS12 and HCS12X represent three generations of 16Bit Controllers with all of them being derived from the industry standard M68HC11. Detailed information for the M68HC12 is provided in the CPU12RM/AD REV 3. This document covers the HCS12 and HCS12X. The term CPU12 is used if the contents applies to both versions.

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 CPU12 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

$\overline{M}$ — 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.

Introduction

Section 2. Overview

2.1 Introduction

This section describes the HCS12 and the HCS12X programming model, register set, the data types used, and basic memory organization.

2.2 Programming Model

The HCS12 programming model, shown in **Figure 2-1**, is the same as that of the M68HC11 CPU. The CPU12 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)
- HCS12 - 8-bit condition code register (CCR)
- HCS12X - 16-bit condition code register (CCRW = CCRH:CCR)

The HCS12X programming model shown in **Figure 2-2** features a 16 Bit condition code register with the lower 8-Bit poprion identical to the HCS12 version.

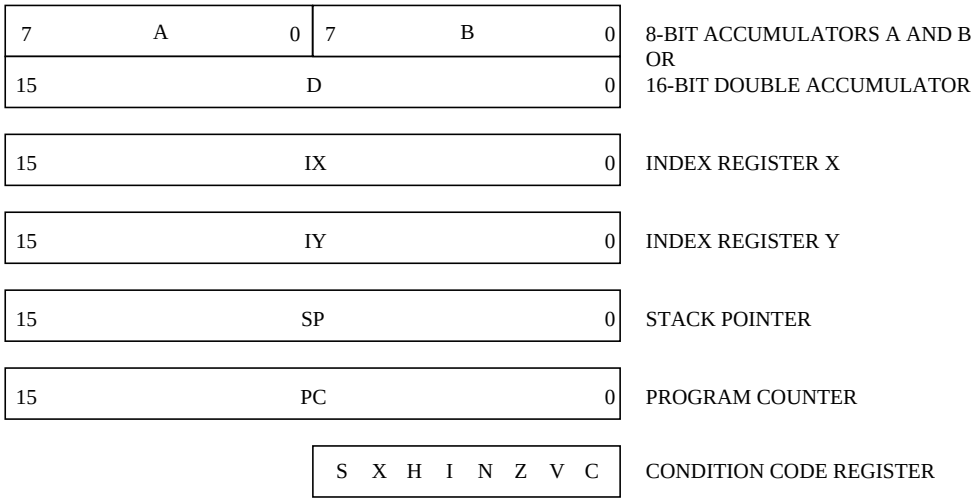


Figure 2-1. HCS12 Programming Model

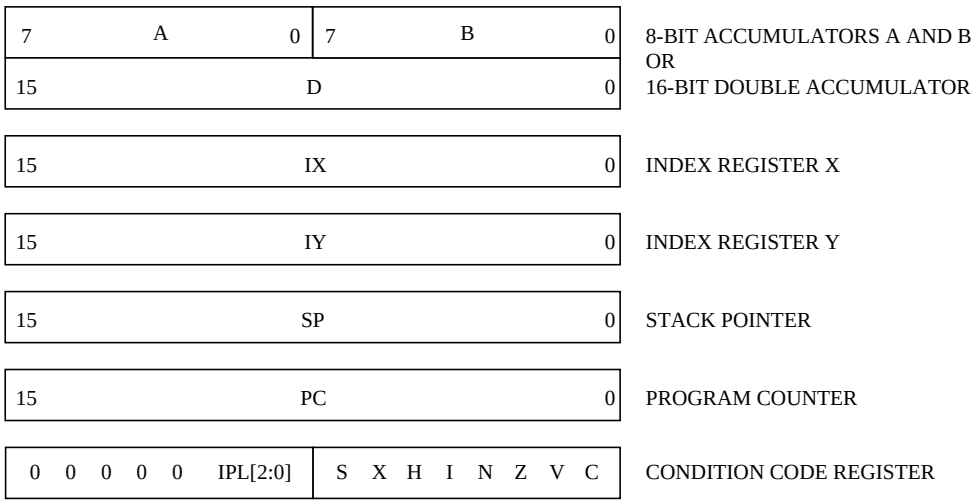


Figure 2-2. HCS12X Programming Model

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 CPU12 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 HCS12 stack frame is the same as that of the M68HC11. HCS12X stack frame has increased by one byte

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
- Interrupt Priority Level (HCS12X only)

The status bits reflect the results of CPU12 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**.

The HCS12X extends this condition code register to a 16-Bit wide register. The lower byte is identical to the HCS12 version. The upper byte holds three bits reflecting the current processing level. Thise bits allow the nesting of interrupts, blocking interrupts of a lower priority.For details on interrupt processing refer to the Interrupt Block Guide.

The upper five bits are reserved for future use and should be written to zero.

2.2.5.1 IPL[2:0]

The IPL bits allow the nesting of interrupts, blocking interrupts of a lower priority. The current IPL is automatically pushed to the stack by the standard interrupt stacking procedure. The new IPL is copied to the CCR from the Priority Level of the highest priority active interrupt request channel. The copying takes place when the interrupt vector is fetched. The previous IPL is automatically restored by executing the RTI instruction.

2.2.5.2 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 CPU12 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.

2.2.5.3 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.

2.2.5.4 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.

2.2.5.5 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, but implementing a nested interrupt management scheme requires great care and seldom improves system performance.

2.2.5.6 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.

2.2.5.7 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).

2.2.5.8 V Status Bit

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

2.2.5.9 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 HCS12 and HCS12X 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 (CPU12) 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 **3.10 Indexed Addressing Modes**. Instructions that use more than one mode are discussed in **3.12 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. HCS12 & HCS12X Addressing Mode Summary

Addressing Mode	Source Format	Abbreviation	Description
Inherent	**INST** (no externally supplied operands)	INH	Operands (if any) are in CPU12 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 CPU12 registers. In either case, the CPU12

does not need to access any memory locations to complete the instruction.

Examples:

```
NOP     ;this instruction has no operands
INX     ;operand is a CPU12 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 CPU12 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 (HCS12)

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 CPU12 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 Direct Addressing Mode (HCS12X)

The Direct Page Register (DIRECT) (refer to Memory Controller Block Guide) determines the position of the direct page within the memory map.The direct addressing mode is based to access operands in the address range $00 through $FF in the direct page. Since these

addresses always begin with the contents of the DIRECT register, 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 DIRECT.

3.8 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.

3.9 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 **3.12 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 CPU12 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.

3.10 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 CPU12 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)

3.10.1 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).

3.10.2 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.

3.10.3 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.

3.10.4 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.

3.10.5 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).

3.10.6 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.

3.10.7 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 CPU12 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.

3.11 Global Addressing (HCS12X only)

The HCS12 Core architecture limits the physical address space available to 64K bytes addr[15:0]. The HCS12X core architecture with the usage of the Global Page Index Register (refer to Memory Controller Block Guide) allows for integrating up to 8 Mbyte of memory addr[22:0] by using the seven global page index bits to page 64K byte blocks into the memory map addr[22:0] is a result of concatenation between GPAGE and addr[15:0].

New instructions started with the label G are created for this usage like (**G**LDAA, **G**STAA,...).

GLDAA : (G(M) ⇒ A) Load Accumulator A from Global Memory

GLDAA has the same addressing mode style like LDAA with the only difference is the memory address (64 KBytes) is presented by the Global memory address (8 MBytes)

This is the case for all Global instructions.

3.12 Instructions Using Multiple Modes

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

3.12.1 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.

In the HCS12 Move instructions do not support indirect modes, 9-bit, or 16-bit offset modes requiring extra extension bytes, while the HCS12X features all addressing modes for the source operand as well as for the destination operand. There are special considerations when using PC-relative addressing with 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 HCS12X, move instructions using PC relative addressing pose a special problem:

- 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.

These case is 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.

.A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB/MOVW instruction.

3.12.2 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 BTAS (Bit Test And Set) works by starting to test bits in memory location M, then set bits in memory location M. To test then set a bit, set the corresponding bit in the mask byte. All other bits in M are unchanged.

BTAS is an atomic instruction and may be used to implement a semaphore.

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.

3.13 Addressing More than 64 Kbytes

Some HCS12 and HCS12X 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 (**3.11 Global Addressing (HCS12X only)**).

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 CPU12 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 HCS12 and HCS12X systems 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.

See specific SoC Guide for more infromation on the memory layout of the particular device.

Addressing Modes

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 (CPU12) 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 (0) 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.

External pins, like IPIPE[1:0] for HCS12 and IQSTAT[3:0] for HCS12X, provide 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 HCS12 and HCS12X 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 CPU12 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 twq 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 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.

4.4 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 CPU12 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 CPU12 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 CPU12 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 CPU12 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 CPU12 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 CPU12 does one program fetch and one

optional fetch to refill the queue[1]. If the branch is taken, the CPU12 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.

Instruction Queue

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 HCS12X. 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 HCS12X 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 CPU12 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) ⇒ A
LDAB	Load B	(M) ⇒ B
LDD	Load D	(M : M + 1) ⇒ (A:B)
LDS	Load SP	(M : M + 1) ⇒ SP_H:SP_L
LDX	Load index register X	(M : M + 1) ⇒ X_H:X_L
LDY	Load index register Y	(M : M + 1) ⇒ 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) ⇒ M
STAB	Store B	(B) ⇒ M
STD	Store D	(A) ⇒ M, (B) ⇒ M + 1
STS	Store SP	(SP_H:SP_L) ⇒ M : M + 1
STX	Store X	(X_H:X_L) ⇒ M : M + 1
STY	Store Y	(Y_H:Y_L) ⇒ 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) ⇒ M_2
MOVW	Move word (16-bit)	(M : M + 1_1) ⇒ 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	$0 \Rightarrow C$
CLI	Clear I bit in CCR	$0 \Rightarrow I$
CLR	Clear memory	$\$00 \Rightarrow M$
CLRA	Clear A	$\$00 \Rightarrow A$
CLRB	Clear B	$\$00 \Rightarrow B$
CLV	Clear V bit in CCR	$0 \Rightarrow V$
COM	One's complement memory	$\$FF - (M) \Rightarrow M$ or $(\overline{M}) \Rightarrow M$
COMA	One's complement A	$\$FF - (A) \Rightarrow A$ or $(\overline{A}) \Rightarrow A$
COMB	One's complement B	$\$FF - (B) \Rightarrow B$ or $(\overline{B}) \Rightarrow B$
NEG	Two's complement memory	$\$00 - (M) \Rightarrow M$ or $(\overline{M}) + 1 \Rightarrow M$
NEGA	Two's complement A	$\$00 - (A) \Rightarrow A$ or $(\overline{A}) + 1 \Rightarrow A$
NEGB	Two's complement B	$\$00 - (B) \Rightarrow B$ 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.

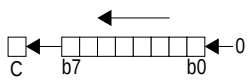
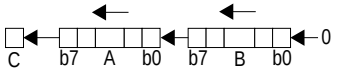
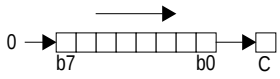
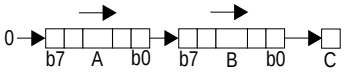
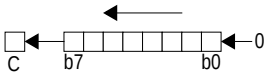
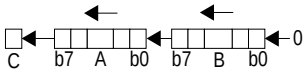
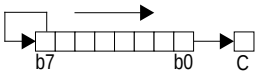
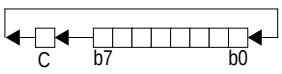
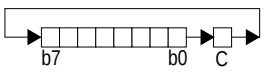
Table 5-11. Bit Test and Manipulation Instructions

Mnemonic	Function	Operation
BCLR	Clear bits in memory	(M) • ($\overline{mm}$) ⇒ M
BITA	Bit test A	(A) • (M)
BITB	Bit test B	(B) • (M)
BSET	Set bits in memory	(M) + (mm) ⇒ 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 μ = 0, else μ = 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) × ((M + 2 : M + 3) – (M : M + 1))] ⇒ 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) × ((M + 1) – (M))] ⇒ 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 ≥ M	C = 0
BLO	Branch if lower	R < M	C = 1
BLS	Branch if lower or same	R ≤ M	C + Z = 1
Signed Branches		
BGE	Branch if greater than or equal	R ≥ M	N ⊕ V = 0
BGT	Branch if greater than	R > M	Z + (N ⊕ V) = 0
BLE	Branch if less than or equal	R ≤ M	Z + (N ⊕ V) = 1
BLT	Branch if less than	R < M	N ⊕ 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 ⊕ V = 0
LBGT	Long branch if greater than	Z + (N ⊕ V) = 0
LBLE	Long branch if less than or equal	Z + (N ⊕ V) = 1
LBLT	Long branch if less than	N ⊕ 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) • (mm) = 0
BRSET	Branch if selected bits set	($\overline{M}$) • (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 ⇒ SP $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address ⇒ PC
CALL	Call subroutine in expanded memory	SP – 2 ⇒ SP $RTN_H:RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ SP – 1 ⇒ SP $(PPAGE) \Rightarrow M_{(SP)}$ Page ⇒ PPAGE Subroutine address ⇒ PC
JMP	Jump	Address ⇒ PC
JSR	Jump to subroutine	SP – 2 ⇒ SP $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ Subroutine address ⇒ PC
RTC	Return from call	$M_{(SP)} \Rightarrow$ PPAGE SP + 1 ⇒ SP $M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ SP + 2 ⇒ SP
RTS	Return from subroutine	$M_{(SP)} : M_{(SP+1)} \Rightarrow PC_H : PC_L$ SP + 2 ⇒ 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 CPU12 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 CPU12 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 CCRH (HCS12X only): 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	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H : CCR$; $(SP) - \$0000 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A$; $(SP) - \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$; $(SP) - \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$; $(SP) - \$0006 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L$; $(SP) - \$0008 \Rightarrow SP$
SWI	Software interrupt	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$
TRAP	Unimplemented opcode interrupt	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$

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; (CCR) ⇒ $M_{(SP)}$
PSHCW	Push CCR_H:CCR	(SP) – 2 ⇒ SP; (CCR_H:CCR) ⇒ $M_{(SP)}$: $M_{(SP+1)}$
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
PULCW	Pull CCR_H:CCR	($M_{(SP)}$: $M_{(SP+1)}$) ⇒ CCR_H:CCR; (SP) + 2 ⇒ SP
PULD	Pull D	($M_{(SP)}$: $M_{(SP+1)}$) ⇒ A : B; (SP) + 2 ⇒ SP

Table 5-24. Stacking Instructions

PULX	Pull X	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X$; $(SP) + 2 \Rightarrow SP$
PULY	Pull Y	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y$; $(SP) + 2 \Rightarrow SP$

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) • (M) ⇒ CCR
CLC	Clear C bit	0 ⇒ C
CLI	Clear I bit	0 ⇒ I
CLV	Clear V bit	0 ⇒ V
ORCC	Logical OR CCR with memory	(CCR) + (M) ⇒ CCR
PSHC	Push CCR onto stack	(SP) – 1⇒ SP; CCR ⇒ $M_{(SP)}$
PSHCW	Push CCR_H:CCR onto stack	(SP) – 2 ⇒ SP; (CCR_H:CCR) ⇒ $M_{(SP)}$:$M_{(SP+1)}$
PULC	Pull CCR from stack	($M_{(SP)}$) ⇒ CCR; (SP) + 1 ⇒ SP
PULCW	Pull CCR_H:CCR from stack	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ CCR_H:CCR; (SP) + 2⇒ SP
SEC	Set C bit	1 ⇒ C
SEI	Set I bit	1 ⇒ I
SEV	Set V bit	1 ⇒ V
TAP	Transfer A to CCR	(A) ⇒ CCR
TPA	Transfer CCR to A	(CCR) ⇒ 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 CPU12 registers and accumulators, then halts all system clocks.

The wait instruction (WAI) stacks a return address and the contents of CPU12 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	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)}\ M_{(SP+1)}$ Stop CPU clocks
WAI	Wait for interrupt	$SP - 2 \Rightarrow SP$; $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP$; $CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$

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 CPU12 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	—

Instruction Set Overview

Section 6. Instruction Glossary

6.1 Introduction

This section is a comprehensive reference to the S12X CPU instruction set. New instructions are denoted with

reverse printing

S12 CPU information is included for a reference point of view.

NOTE: *The glossary contains condition code register (CCR) details for each assembler mnemonic. Unless the high byte is affected, only the low byte of the condition code register is shown.*

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.

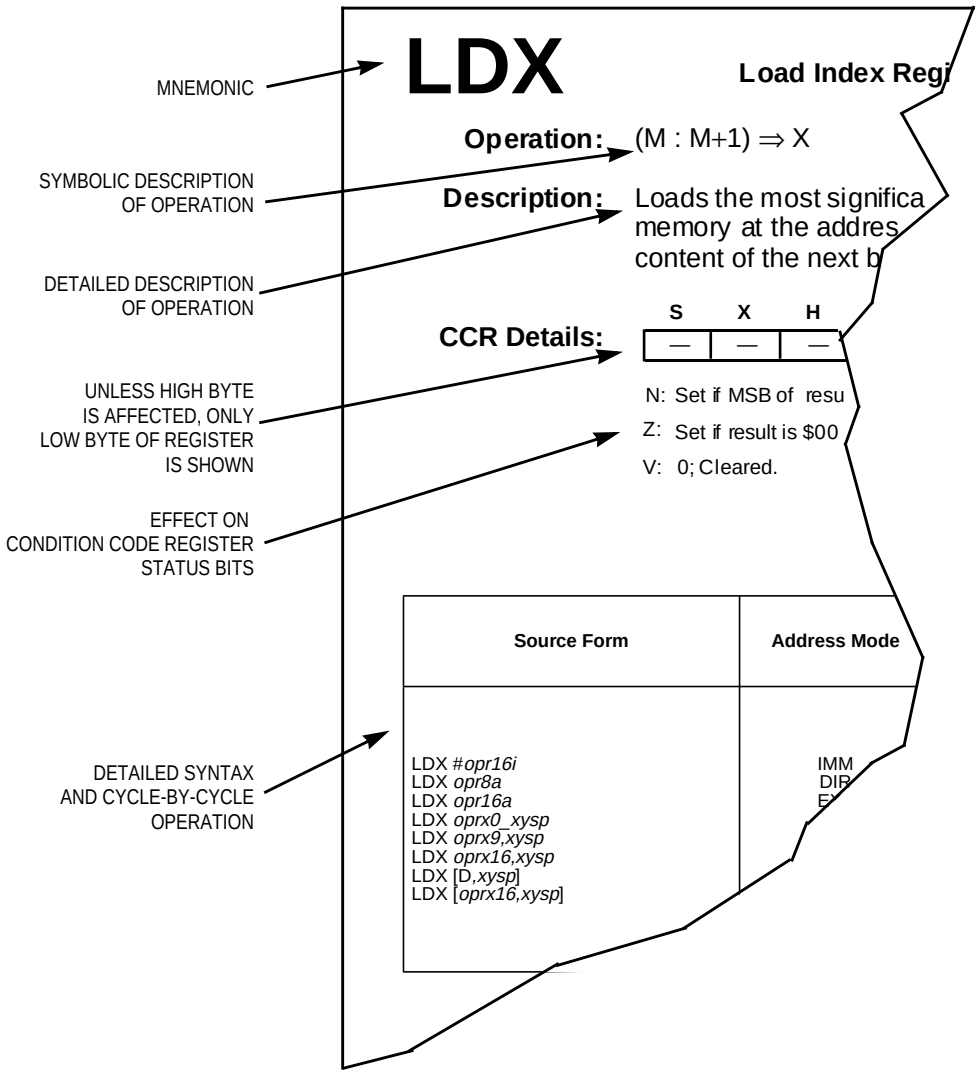


Figure 6-1. Example Glossary Page

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

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

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 CPU12 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 Motorola 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 CPU12 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 CPU12 is not aware of the stretch delays because the clock to the CPU12 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 CPU12 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 CPU12.

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.

l — 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.

NA — Not available

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 CPU12 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 CPU12 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	
			HCS12X	HCS12
ABA	INH	18 06	00	00

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 HCS12X	Access Detail HCS12
ABX *translates to...* LEAX B,X	IDX	1A E5	Pf	Pf

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	
			HCS12X	HCS12
ABY *translates to...* LEAY B,Y	IDX	19 ED	Pf	Pf

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 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	
			HCS12X	HCS12
ADCA #*opr8i*	IMM	89 ii	P	P
ADCA *opr8a*	DIR	99 dd	rPf	rPf
ADCA *opr16a*	EXT	B9 hh ll	rPO	rPO
ADCA *oprx0_xysp*	IDX	A9 xb	rPf	rPf
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	fIfrPf
ADCA [*oprx16,xysp*]	[IDX2]	A9 xb ee ff	fIPrPf	fIPrPf

ADCB Add with Carry to B ADCB

Operation: (B) + (M) + C ⇒ 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 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	
			HCS12X	HCS12
ADCB #*opr8i*	IMM	C9 ii	P	P
ADCB *opr8a*	DIR	D9 dd	rPf	rPf
ADCB *opr16a*	EXT	F9 hh ll	rPO	rPO
ADCB *oprx0_xysp*	IDX	E9 xb	rPf	rPf
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	fIfrPf
ADCB [*oprx16,xysp*]	[IDX2]	E9 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ADDA #*opr8i*	IMM	8B ii	P	P
ADDA *opr8a*	DIR	9B dd	rPf	rPf
ADDA *opr16a*	EXT	BB hh ll	rPO	rPO
ADDA *oprx0_xysp*	IDX	AB xb	rPf	rPf
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	fIfrPf
ADDA [*oprx16,xysp*]	[IDX2]	AB xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ADDB *#opr8i*	IMM	CB ii	P	P
ADDB *opr8a*	DIR	DB dd	rPf	rPf
ADDB *opr16a*	EXT	FB hh ll	rPO	rPO
ADDB *oprx0_xysp*	IDX	EB xb	rPf	rPf
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	fIfrPf
ADDB [*oprx16,xysp*]	[IDX2]	EB xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ADDD #*opr16i*	IMM	C3 jj kk	P0	P0
ADDD *opr8a*	DIR	D3 dd	RPf	RPf
ADDD *opr16a*	EXT	F3 hh ll	RP0	RP0
ADDD *oprx0_xysp*	IDX	E3 xb	RPf	RPf
ADDD *oprx9,xysp*	IDX1	E3 xb ff	RP0	RP0
ADDD *oprx16,xysp*	IDX2	E3 xb ee ff	fRPP	fRPP
ADDD [D,*xysp*]	[D,IDX]	E3 xb	fIfRPF	fIfRPF
ADDD [*oprx16,xysp*]	[IDX2]	E3 xb ee ff	fIPRPf	fIPRPf

ADDX — Add without Carry to X — ADDX

Operation: (X) + (M : M + 1) ⇒ X

Description: Adds the content of index register X to the contents of memory location M : M + 1 and places the result in X.

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 HCS12X	Access Detail HCS12
ADDX *#opr16i*	IMM	18 8B jj kk	OPO	NA
ADDX *opr8a*	DIR	18 9B dd	ORPf	NA
ADDX *opr16a*	EXT	18 BB hh ll	ORPO	NA
ADDX *oprx0_xysp*	IDX	18 AB xb	ORPf	NA
ADDX *oprx9,xysp*	IDX1	18 AB xb ff	ORPO	NA
ADDX *oprx16,xysp*	IDX2	18 AB xb ee ff	OfRPP	NA
ADDX [D,*xysp*]	[D,IDX]	18 AB xb	OfIfRPf	NA
ADDX [*oprx16,xysp*]	[IDX2]	18 AB xb ee ff	OfIPRPf	NA

ADDY — Add without Carry to Y — ADDY

Operation: (Y) + (M : M + 1) ⇒ Y

Description: Adds the content of index register Y to the contents of memory location M : M + 1 and places the result in Y.

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

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

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ADDY *#opr16i*	IMM	18 CB jj kk	OPO	NA
ADDY *opr8a*	DIR	18 DB dd	ORPf	NA
ADDY *opr16a*	EXT	18 FB hh ll	ORPO	NA
ADDY *oprx0_xysp*	IDX	18 EB xb	ORPf	NA
ADDY *oprx9,xysp*	IDX1	18 EB xb ff	ORPO	NA
ADDY *oprx16,xysp*	IDX2	18 EB xb ee ff	OfRPP	NA
ADDY [D,*xysp*]	[D,IDX]	18 EB xb	OfIfRPf	NA
ADDY [*oprx16,xysp*]	[IDX2]	18 EB xb ee ff	OfIPRPf	NA

ADED — Add with Carry to D (A:B) — ADED

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

Description: Adds the content of accumulator A : B to the content of memory location M : M + 1, then adds the value of the C bit and places the result in A : 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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

C: $X15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet X15$
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	
			HCS12X	HCS12
ADED *#opr16i*	IMM	18 C3 jj kk	OPO	NA
ADED *opr8a*	DIR	18 D3 dd	ORPf	NA
ADED *opr16a*	EXT	18 F3 hh ll	ORPO	NA
ADED *oprx0_xysp*	IDX	18 E3 xb	ORPf	NA
ADED *oprx9,xysp*	IDX1	18 E3 xb ff	ORPO	NA
ADED *oprx16,xysp*	IDX2	18 E3 xb ee ff	OfRPP	NA
ADED [D,*xysp*]	[D,IDX]	18 E3 xb	OfIfRPf	NA
ADED [*oprx16,xysp*]	[IDX2]	18 E3 xb ee ff	OfIPRPf	NA

ADEX Add with Carry to X ADEX

Operation: (X) + (M : M + 1) + C ⇒ X

Description: Adds the content of index register X to the content of memory location M : M + 1, then adds the value of the C bit and places the result in X.

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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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

C: $X15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet X15$
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	
			HCS12X	HCS12
ADEX *#opr16i*	IMM	18 89 jj kk	OPO	NA
ADEX *opr8a*	DIR	18 99 dd	ORPf	NA
ADEX *opr16a*	EXT	18 B9 hh ll	ORPO	NA
ADEX *oprx0_xysp*	IDX	18 A9 xb	ORPf	NA
ADEX *oprx9,xysp*	IDX1	18 A9 xb ff	ORPO	NA
ADEX *oprx16,xysp*	IDX2	18 A9 xb ee ff	OfRPP	NA
ADEX [D,*xysp*]	[D,IDX]	18 A9 xb	OfIfRPf	NA
ADEX [*oprx16,xysp*]	[IDX2]	18 A9 xb ee ff	OfIPRPf	NA

ADEY Add with Carry to Y ADEY

Operation: (Y) + (M : M + 1) + C ⇒ Y

Description: Adds the content of index register Y to the content of memory location M : M + 1, then adds the value of the C bit and places the result in Y.

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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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

C: $X15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet X15$
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	
			HCS12X	HCS12
ADEY *#opr16i*	IMM	18 C9 jj kk	OPO	NA
ADEY *opr8a*	DIR	18 D9 dd	ORPf	NA
ADEY *opr16a*	EXT	18 F9 hh ll	ORPO	NA
ADEY *oprx0_xysp*	IDX	18 E9 xb	ORPf	NA
ADEY *oprx9,xysp*	IDX1	18 E9 xb ff	ORPO	NA
ADEY *oprx16,xysp*	IDX2	18 E9 xb ee ff	OfRPP	NA
ADEY [D,*xysp*]	[D,IDX]	18 E9 xb	OfIfRPf	NA
ADEY [*oprx16,xysp*]	[IDX2]	18 E9 xb ee ff	OfIPRPf	NA

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 HCS12X	Access Detail HCS12
ANDA #*opr8i*	IMM	84 ii	P	P
ANDA *opr8a*	DIR	94 dd	rPf	rPf
ANDA *opr16a*	EXT	B4 hh ll	rPO	rPO
ANDA *oprx0_xysp*	IDX	A4 xb	rPf	rPf
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	fIfrPf
ANDA [*oprx16,xysp*]	[IDX2]	A4 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ANDB #*opr8i*	IMM	C4 ii	P	P
ANDB *opr8a*	DIR	D4 dd	rPf	rPf
ANDB *opr16a*	EXT	F4 hh ll	rPO	rPO
ANDB *oprx0_xysp*	IDX	E4 xb	rPf	rPf
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	fIfrPf
ANDB [*oprx16,xysp*]	[IDX2]	E4 xb ee ff	fIPrPf	fIPrPf

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, SEI (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	
			HCS12X	HCS12
ANDCC #*opr8i*	IMM	`10 ii`	P	P

ANDX — Logical AND X — ANDX

Operation: (X) • (M : M + 1) ⇒ X

Description: Performs logical AND between the content of memory location M : M + 1 and the content of index register X. The result is placed in X. After the operation is performed, each bit of X is the logical AND of the corresponding bits of M : M + 1 and of X 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ANDX *#opr16i*	IMM	18 84 jj kk	OPO	NA
ANDX *opr8a*	DIR	18 94 dd	ORPf	NA
ANDX *opr16a*	EXT	18 B4 hh ll	ORPO	NA
ANDX *oprx0_xysp*	IDX	18 A4 xb	ORPf	NA
ANDX *oprx9,xysp*	IDX1	18 A4 xb ff	ORPO	NA
ANDX *oprx16,xysp*	IDX2	18 A4 xb ee ff	OfRPP	NA
ANDX [D,*xysp*]	[D,IDX]	18 A4 xb	OfIfRPf	NA
ANDX [*oprx16,xysp*]	[IDX2]	18 A4 xb ee ff	OfIPRPf	NA

ANDY — Logical AND Y — ANDY

Operation: (Y) • (M : M + 1) ⇒ Y

Description: Performs logical AND between the content of memory location M : M + 1 and the content of index register Y. The result is placed in Y. After the operation is performed, each bit of Y is the logical AND of the corresponding bits of M : M + 1 and of Y 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 $0000; cleared otherwise

V: 0; cleared

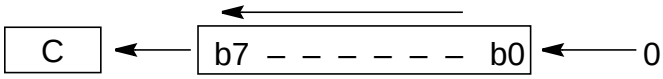
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ANDY *#opr16i*	IMM	18 C4 jj kk	OPO	NA
ANDY *opr8a*	DIR	18 D4 dd	ORPf	NA
ANDY *opr16a*	EXT	18 F4 hh ll	ORPO	NA
ANDY *oprx0_xysp*	IDX	18 E4 xb	ORPf	NA
ANDY *oprx9,xysp*	IDX1	18 E4 xb ff	ORPO	NA
ANDY *oprx16,xysp*	IDX2	18 E4 xb ee ff	OfRPP	NA
ANDY [D,*xysp*]	[D,IDX]	18 E4 xb	OfIfRPf	NA
ANDY [*oprx16,xysp*]	[IDX2]	18 E4 xb ee ff	OfIPRPf	NA

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

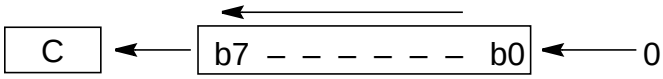
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASL *opr16a*	EXT	`78 hh ll`	`rPw0`	`rPw0`
ASL *oprx0_xysp*	IDX	`68 xb`	`rPw`	`rPw`
ASL *oprx9,xysp*	IDX1	`68 xb ff`	`rPw0`	`rPw0`
ASL *oprx16,xysp*	IDX2	`68 xb ee ff`	`frPwP`	`frPwP`
ASL [D,*xysp*]	[D,IDX]	`68 xb`	`fIfrPw`	`fIfrPw`
ASL [*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. 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 MSB of A was set before the shift; cleared otherwise

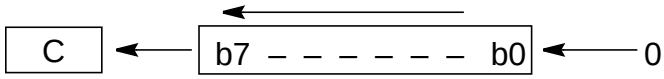
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASLA	INH	48	0	0

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

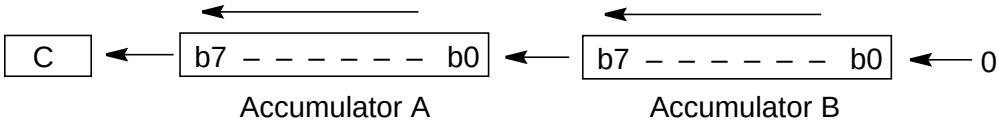
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASLB	INH	58	0	0

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

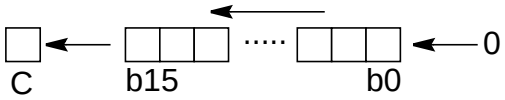
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASLD	INH	59	0	0

ASLW Arithmetic Shift Left W ASLW

Operation:



Description: Shifts all bits of memory location M : M + 1 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 W.

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

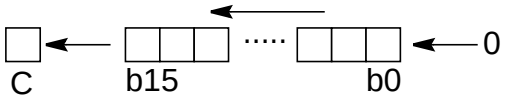
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M15
Set if the MSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASLW *opr16a*	EXT	`18 78 hh ll`	`ORPWO`	NA
ASLW *oprx0_xysp*	IDX	`18 68 xb`	`ORPW`	NA
ASLW *oprx9,xysp*	IDX1	`18 68 xb ff`	`ORPWO`	NA
ASLW *oprx16,xysp*	IDX2	`18 68 xb ee ff`	`OfRPWP`	NA
ASLW [D,*xysp*]	[D,IDX]	`18 68 xb`	`OfIfRPW`	NA
ASLW [*oprx16,xysp*]	[IDX2]	`18 68 xb ee ff`	`OfIPRPW`	NA

ASLX Arithmetic Shift Left Index Register X ASLX

Operation:



Description: Shifts all bits of index register X 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 X.

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

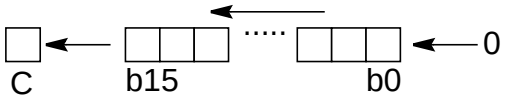
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X15
Set if the MSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASLX	INH	18 48	00	NA

ASLY Arithmetic Shift Left Index Register Y ASLY

Operation:



Description: Shifts all bits of index register Y 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 Y.

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

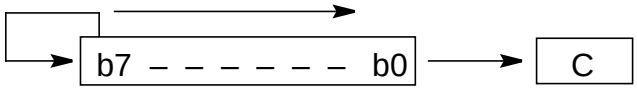
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y15
Set if the MSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ASLY	INH	18 58	00	NA

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

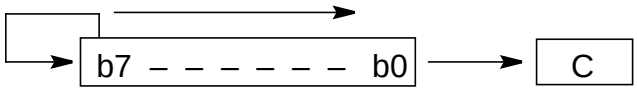
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASR *opr16a*	EXT	`77 hh ll`	`rPw0`	`rPw0`
ASR *oprx0_xysp*	IDX	`67 xb`	`rPw`	`rPw`
ASR *oprx9,xysp*	IDX1	`67 xb ff`	`rPw0`	`rPw0`
ASR *oprx16,xysp*	IDX2	`67 xb ee ff`	`frPwP`	`frPwP`
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

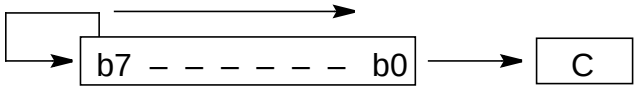
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASRA	INH	47	0	0

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

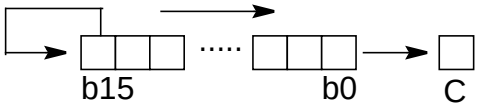
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASRB	INH	57	0	0

ASRW Arithmetic Shift Right W ASRW

Operation:



Description: Shifts all bits of memory location M : M + 1 one place to the right. Bit 15 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 $0000; cleared otherwise

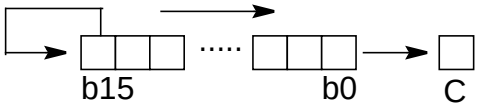
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M : M + 1 was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ASRW *opr16a*	EXT	`18 77 hh ll`	`ORPWO`	NA
ASRW *oprx0_xysp*	IDX	`18 67 xb`	`ORPW`	NA
ASRW *oprx9,xysp*	IDX1	`18 67 xb ff`	`ORPWO`	NA
ASRW *oprx16,xysp*	IDX2	`18 67 xb ee ff`	`OfRPWP`	NA
ASRW [D,*xysp*]	[D,IDX]	`18 67 xb`	`OfIfRPW`	NA
ASRW [*oprx16,xysp*]	[IDX2]	`18 67 xb ee ff`	`OfIPRPW`	NA

ASRX Arithmetic Shift Right Index Register X ASRX

Operation:



Description: Shifts all bits of index register X one place to the right. Bit 15 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 $0000; cleared otherwise

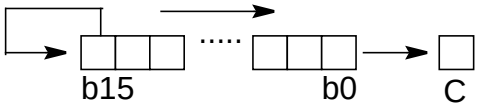
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X0
Set if the LSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASRX	INH	18 47	00	NA

ASRY Arithmetic Shift Right Index Register Y ASRY

Operation:



Description: Shifts all bits of index register Y one place to the right. Bit 15 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 $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: Y0
Set if the LSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ASRY	INH	18 57	00	NA

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 **3.9 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	
			HCS12X	HCS12
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 ⊕ 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

BCLR Clear Bits in Memory BCLR

Operation: (M) • ($\overline{\text{Mask}}$) ⇒ 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 HCS12X	Access Detail HCS12
BCLR *opr8a, msk8*	DIR	4D dd mm	rPw0	rPw0
BCLR *opr16a, msk8*	EXT	1D hh ll mm	rPwP	rPwP
BCLR *oprx0_xysp, msk8*	IDX	0D xb mm	rPw0	rPw0
BCLR *oprx9,xysp, msk8*	IDX1	0D xb ff mm	rPwP	rPwP
BCLR *oprx16,xysp, msk8*	IDX2	0D xb ee ff mm	frPwP0	frPwP0

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 **3.9 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	
			HCS12X	HCS12
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 ⇒ PC

Simple branch

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

See **3.9 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	
			HCS12X	HCS12
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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 CPU12 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 CPU12 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 CPU12 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 CPU12 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 4. 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	
			HCS12X	HCS12
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 CPU12 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 **3.9 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 HCS12X	Access Detail HCS12
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 ⇒ 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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 ⊕ 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

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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
BITA #*opr8i*	IMM	85 ii	P	P
BITA *opr8a*	DIR	95 dd	rPf	rPf
BITA *opr16a*	EXT	B5 hh ll	rPO	rPO
BITA *oprx0_xysp*	IDX	A5 xb	rPf	rPf
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	fIfrPf
BITA [*oprx16,xysp*]	[IDX2]	A5 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
BITB #*opr8i*	IMM	C5 ii	P	P
BITB *opr8a*	DIR	D5 dd	rPf	rPf
BITB *opr16a*	EXT	F5 hh ll	rPO	rPO
BITB *oprx0_xysp*	IDX	E5 xb	rPf	rPf
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	fIfrPf
BITB [*oprx16,xysp*]	[IDX2]	E5 xb ee ff	fIPrPf	fIPrPf

BITX Bit Test X BITX

Operation: (X) • (M : M + 1)

Description: Performs bitwise logical AND on the content of index register X and the content of memory location M : M + 1 and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the index register and the memory location. Neither the content of the index register 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 \$0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
BITX #*opr16i*	IMM	18 85 jj kk	OPO	NA
BITX *opr8a*	DIR	18 95 dd	ORPf	NA
BITX *opr16a*	EXT	18 B5 hh ll	ORPO	NA
BITX *oprx0_xysp*	IDX	18 A5 xb	ORPf	NA
BITX *oprx9,xysp*	IDX1	18 A5 xb ff	ORPO	NA
BITX *oprx16,xysp*	IDX2	18 A5 xb ee ff	OfRPP	NA
BITX [D,*xysp*]	[D,IDX]	18 A5 xb	OfIfRPf	NA
BITX [*oprx16,xysp*]	[IDX2]	18 A5 xb ee ff	OfIPRPf	NA

BITY — Bit Test X — BITY

Operation: (Y) • (M : M + 1)

Description: Performs bitwise logical AND on the content of index register Y and the content of memory location M : M + 1 and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the index register and the memory location. Neither the content of the index register 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
BITY #*opr16i*	IMM	18 C5 jj kk	OPO	NA
BITY *opr8a*	DIR	18 D5 dd	ORPf	NA
BITY *opr16a*	EXT	18 F5 hh ll	ORPO	NA
BITY *oprx0_xysp*	IDX	18 E5 xb	ORPf	NA
BITY *oprx9,xysp*	IDX1	18 E5 xb ff	ORPO	NA
BITY *oprx16,xysp*	IDX2	18 E5 xb ee ff	OfRPP	NA
BITY [D,*xysp*]	[D,IDX]	18 E5 xb	OfIfRPf	NA
BITY [*oprx16,xysp*]	[IDX2]	18 E5 xb ee ff	OfIPRPf	NA

BLE Branch if Less Than or Equal to Zero BLE

Operation: If Z + (N ⊕ V) = 1, then (PC) + $0002 + Rel ⇒ 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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 ⊕ 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

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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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 ⊕ 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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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	rfPPP
BRCLR *oprx16,xysp, msk8, rel8*	IDX2	0F xb ee ff mm rr	PrfPPP	PrfPPP

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 **3.9 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	
			HCS12X	HCS12
BRN *rel8*	REL	21 rr	P	P

BRSET Branch if Bits Set BRSET

Operation: If ($\overline{M}$) • (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 **3.9 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 HCS12X	Access Detail HCS12
BRSET *oprx0_xysp, msk8, rel8*	IDX	0E xb mm rr	rPPP	rPPP
BRSET *oprx9,xysp, msk8, rel8*	IDX1	0E xb ff mm rr	rfPPP	rfPPP
BRSET *oprx16,xysp, msk8, rel8*	IDX2	0E xb ee ff mm rr	PrfPPP	PrfPPP
BRSET *opr8a, msk8, rel8*	DIR	4E dd mm rr	rPPP	rPPP
BRSET *opr16a, msk8, rel8*	EXT	1E hh ll mm rr	rfPPP	rfPPP

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	
			HCS12X	HCS12
BSET *opr8a, msk8*	DIR	4C dd mm	rPwO	rPwO
BSET *opr16a, msk8*	EXT	1C hh ll mm	rPwP	rPwP
BSET *oprx0_xysp, msk8*	IDX	0C xb mm	rPwO	rPwO
BSET *oprx9,xysp, msk8*	IDX1	0C xb ff mm	rPwP	rPwP
BSET *oprx16,xysp, msk8*	IDX2	0C xb ee ff mm	frPwPO	frPwPO

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	
			HCS12X	HCS12
BSR *rel8*	REL	`07 rr`	SPPP	SPPP

BTAS Bit(s) Test and Set in Memory BTAS

Operation: If (M) • (Mask) = 0, then set Z, else clear Z

(M) | (Mask) ⇒ M

Description: Test bits in memory location M, then set bits in memory location M. To test then set a bit, set the corresponding bit in the mask byte. All other bits in M are unchanged. BTAS is an atomic instruction and may be used to implement a semaphore.

NOTE: *The CCR bits are affected by the test operation, (M) AND (Mask), and not the result operation, (M) OR (Mask).*

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

Z: Set if test is $00; cleared otherwise

V: 0; cleared

Source Form	Address Mode(1)	Object Code	Access Detail	
			HCS12X	HCS12
BTAS *opr8, msk8*	DIR	18 35 dd mm	ORPWO	NA
BTAS *opr16a, msk8*	EXT	18 36 hh ll mm	ORPWP	NA
BTAS *oprx0_xysp, msk8*	IDX	18 37 xb mm	ORPWO	NA
BTAS *oprx9,xysp, msk8*	IDX1	18 37 xb ff mm	ORPWP	NA
BTAS *oprx16,xysp, msk8*	IDX2	18 37 xb ee ff mm	OfRPWPO	NA

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

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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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:

(SP) – $0002 ⇒ SP; RTN_H : RTN_L ⇒ $M_{(SP)}$: $M_{(SP+1)}$
(SP) – $0001 ⇒ SP; (PPAGE) ⇒ $M_{(SP)}$
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	
			HCS12X	HCS12
CALL *opr16a, page*	EXT	4A hh ll pg	gnSsPPP	gnSsPPP
CALL *oprx0_xysp, page*	IDX	4B xb pg	gnSsPPP	gnSsPPP
CALL *oprx9,xysp, page*	IDX1	4B xb ff pg	gnSsPPP	gnSsPPP
CALL *oprx16,xysp, page*	IDX2	4B xb ee ff pg	fgnSsPPP	fgnSsPPP
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	
			HCS12X	HCS12
CBA	INH	18 17	00	00

CLC — Clear Carry — CLC

Operation: $0 \Rightarrow$ 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 HCS12X	Access Detail HCS12
CLC *translates to...* ANDCC #$FE	IMM	10 FE	P	P

CLI — Clear Interrupt Mask — CLI

Operation: $0 \Rightarrow$ 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 HCS12X	Access Detail HCS12
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	
			HCS12X	HCS12
CLR *opr16a*	EXT	79 hh ll	PwO	PwO
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	PIfw
CLR [*oprx16,xysp*]	[IDX2]	69 xb ee ff	PIPw	PIPw

CLRA Clear A CLRA

Operation: $0 \Rightarrow A$

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	
			HCS12X	HCS12
CLRA	INH	87	0	0

CLRB Clear B CLRB

Operation: $0 \Rightarrow B$

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	
			HCS12X	HCS12
CLRB	INH	C7	0	0

CLRW Clear Memory CLRW

Operation: $0 \Rightarrow M : M + 1$

Description: All bits in memory location M : M + 1 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	
			HCS12X	HCS12
CLRW *opr16a*	EXT	`18 79 hh ll`	`OPWO`	`NA`
CLRW *oprx0_xysp*	IDX	`18 69 xb`	`OPW`	`NA`
CLR*W oprx9,xysp*	IDX1	`18 69 xb ff`	`OPWO`	`NA`
CLR*W oprx16,xysp*	IDX2	`18 69 xb ee ff`	`OPWP`	`NA`
CLRW [D,*xysp*]	[D,IDX]	`18 69 xb`	`OPIfW`	`NA`
CLRW [*oprx16,xysp*]	[IDX2]	`18 69 xb ee ff`	`OPIPW`	`NA`

CLRX Clear Memory CLRX

Operation: $0 \Rightarrow X$

Description: All bits in index register X 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	
			HCS12X	HCS12
CLRX	INH	18 87	00	NA

CLRY Clear Memory CLRY

Operation: $0 \Rightarrow Y$

Description: All bits in index register Y 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	
			HCS12X	HCS12
CLRY	INH	18 C7	00	NA

CLV — Clear Two's Complement Overflow Bit — CLV

Operation: $0 \Rightarrow$ 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	
			HCS12X	HCS12
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 HCS12X	Access Detail HCS12
CMPA #*opr8i*	IMM	81 ii	P	P
CMPA *opr8a*	DIR	91 dd	rPf	rPf
CMPA *opr16a*	EXT	B1 hh ll	rPO	rPO
CMPA *oprx0_xysp*	IDX	A1 xb	rPf	rPf
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	fIfrPf
CMPA [*oprx16*,*xysp*]	[IDX2]	A1 xb ee ff	fIPrPf	fIPrPf

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 HCS12X	Access Detail HCS12
CMPB #*opr8i*	IMM	C1 ii	P	P
CMPB *opr8a*	DIR	D1 dd	rPf	rPf
CMPB *opr16a*	EXT	F1 hh ll	rPO	rPO
CMPB *oprx0_xysp*	IDX	E1 xb	rPf	rPf
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	fIfrPf
CMPB [*oprx16*,*xysp*]	[IDX2]	E1 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
COM *opr16a*	EXT	71 hh ll	rPw0	rPw0
COM *oprx0_xysp*	IDX	61 xb	rPw	rPw
COM *oprx9,xysp*	IDX1	61 xb ff	rPw0	rPw0
COM *oprx16,xysp*	IDX2	61 xb ee ff	frPwP	frPwP
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	
			HCS12X	HCS12
COMA	INH	41	0	0

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	
			HCS12X	HCS12
COMB	INH	51	0	0

COMW Complement Memory COMW

Operation: $(\overline{M : M + 1}) = \$FFFF - (M : M + 1) \Rightarrow M : M + 1$

Description: Replaces the content of memory location M : M + 1 with its one's complement. Each bit of M : M + 1 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 $0000; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
COMW *opr16a*	EXT	`18 71 hh ll`	`ORPWO`	NA
COMW *oprx0_xysp*	IDX	`18 61 xb`	`ORPW`	NA
COMW *oprx9,xysp*	IDX1	`18 61 xb ff`	`ORPWO`	NA
COMW *oprx16,xysp*	IDX2	`18 61 xb ee ff`	`OfRPWP`	NA
COMW [D,*xysp*]	[D,IDX]	`18 61 xb`	`OfIfRPW`	NA
COMW [*oprx16,xysp*]	[IDX2]	`18 61 xb ee ff`	`OfIPRPW`	NA

COMX Complement Index Register X COMX

Operation: $(\overline{X}) = (\$FFFF - X) \Rightarrow X$

Description: Replaces the content of index register X with its one's complement. Each bit of X 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 $0000; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
COMX	INH	18 41	00	NA

COMY Complement Index Register Y COMY

Operation: $(\overline{Y}) = (\$FFFF - Y) \Rightarrow Y$

Description: Replaces the content of index register Y with its one's complement. Each bit of Y 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 $0000; cleared otherwise

V: 0; cleared

C: 1; set (for M6800 compatibility)

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
COMY	INH	18 51	00	NA

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	
			HCS12X	HCS12
CPD #*opr16i*	IMM	8C jj kk	P0	P0
CPD *opr8a*	DIR	9C dd	RPf	RPf
CPD *opr16a*	EXT	BC hh ll	RP0	RP0
CPD *oprx0_xysp*	IDX	AC xb	RPf	RPf
CPD *oprx9,xysp*	IDX1	AC xb ff	RP0	RP0
CPD *oprx16,xysp*	IDX2	AC xb ee ff	fRPP	fRPP
CPD [D,*xysp*]	[D,IDX]	AC xb	fIfRPf	fIfRPf
CPD [*oprx16,xysp*]	[IDX2]	AC xb ee ff	fIPRPf	fIPRPf

CPED — Compare D to Memory with Borrow — CPED

Operation: (A : B) – ((M : M + 1) + C)

Description: Compares the content of 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) + C) 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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 plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
CPED #*opr16i*	IMM	18 8C jj kk	OPO	NA
CPED *opr8a*	DIR	18 9C dd	ORPf	NA
CPED *opr16a*	EXT	18 BC hh ll	ORPO	NA
CPED *oprx0_xysp*	IDX	18 AC xb	ORPf	NA
CPED *oprx9,xysp*	IDX1	18 AC xb ff	ORPO	NA
CPED *oprx16,xysp*	IDX2	18 AC xb ee ff	OfRPP	NA
CPED [D,*xysp*]	[D,IDX]	18 AC xb	OfIfRPf	NA
CPED [*oprx16,xysp*]	[IDX2]	18 AC xb ee ff	OfIPRPf	NA

CPES Compare SP to Memory with Borrow CPES

Operation: (SP) – ((M : M + 1) + C)

Description: Compares the content of stack pointer SP 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) + C) from SP without modifying either 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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 plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
CPES #*opr16i*	IMM	`18 8F jj kk`	`OPO`	NA
CPES *opr8a*	DIR	`18 9F dd`	`ORPf`	NA
CPES *opr16a*	EXT	`18 BF hh ll`	`ORPO`	NA
CPES *oprx0_xysp*	IDX	`18 AF xb`	`ORPf`	NA
CPES *oprx9,xysp*	IDX1	`18 AF xb ff`	`ORPO`	NA
CPES *oprx16,xysp*	IDX2	`18 AF xb ee ff`	`OfRPP`	NA
CPES [D,*xysp*]	[D,IDX]	`18 AF xb`	`OfIfRPf`	NA
CPES [*oprx16,xysp*]	[IDX2]	`18 AF xb ee ff`	`OfIPRPf`	NA

CPEX Compare X to Memory with Borrow CPEX

Operation: (X) – ((M : M + 1) + C)

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) + C) from X without modifying either 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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 plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
CPEX #*opr16i*	IMM	18 8E jj kk	OPO	NA
CPEX *opr8a*	DIR	18 9E dd	ORPf	NA
CPEX *opr16a*	EXT	18 BE hh ll	ORPO	NA
CPEX *oprx0_xysp*	IDX	18 AE xb	ORPf	NA
CPEX *oprx9,xysp*	IDX1	18 AE xb ff	ORPO	NA
CPEX *oprx16,xysp*	IDX2	18 AE xb ee ff	OfRPP	NA
CPEX [D,*xysp*]	[D,IDX]	18 AE xb	OfIfRPf	NA
CPEX [*oprx16,xysp*]	[IDX2]	18 AE xb ee ff	OfIPRPf	NA

CPEY Compare Y to Memory with Borrow CPEY

Operation: (Y) – ((M : M + 1) + C)

Description: Compares the content of index register Y 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) + C) 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

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 plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
CPEY #*opr16i*	IMM	18 8D jj kk	OPO	NA
CPEY *opr8a*	DIR	18 9D dd	ORPf	NA
CPEY *opr16a*	EXT	18 BD hh ll	ORPO	NA
CPEY *oprx0_xysp*	IDX	18 AD xb	ORPf	NA
CPEY *oprx9,xysp*	IDX1	18 AD xb ff	ORPO	NA
CPEY *oprx16,xysp*	IDX2	18 AD xb ee ff	OfRPP	NA
CPEY [D,*xysp*]	[D,IDX]	18 AD xb	OfIfRPf	NA
CPEY [*oprx16,xysp*]]	[IDX2]	18 AD xb ee ff	OfIPRPf	NA

CPS — Compare Stack Pointer — CPS

Operation: (SP) – (M : M + 1)

Description: Compares the content of the stack pointer 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 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 HCS12X	Access Detail HCS12
CPS #*opr16i*	IMM	8F jj kk	PO	PO
CPS *opr8a*	DIR	9F dd	RPf	RPf
CPS *opr16a*	EXT	BF hh ll	RPO	RPO
CPS *oprx0_xysp*	IDX	AF xb	RPf	RPf
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	fIfRPf
CPS [*oprx16,xysp*]	[IDX2]	AF xb ee ff	fIPRPf	fIPRPf

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	
			HCS12X	HCS12
CPX #*opr16i*	IMM	8E jj kk	P0	P0
CPX *opr8a*	DIR	9E dd	RPf	RPf
CPX *opr16a*	EXT	BE hh ll	RP0	RP0
CPX *oprx0_xysp*	IDX	AE xb	RPf	RPf
CPX *oprx9,xysp*	IDX1	AE xb ff	RP0	RP0
CPX *oprx16,xysp*	IDX2	AE xb ee ff	fRPP	fRPP
CPX [D,*xysp*]	[D,IDX]	AE xb	fIfRPf	fIfRPf
CPX [*oprx16,xysp*]	[IDX2]	AE xb ee ff	fIPRPf	fIPRPf

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 HCS12X	Access Detail HCS12
CPY #*opr16i*	IMM	8D jj kk	PO	PO
CPY *opr8a*	DIR	9D dd	RPf	RPf
CPY *opr16a*	EXT	BD hh ll	RPO	RPO
CPY *oprx0_xysp*	IDX	AD xb	RPf	RPf
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	fIfRPf
CPY [*oprx16,xysp*]	[IDX2]	AD xb ee ff	fIPRPf	fIPRPf

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 HCS12X	Access Detail HCS12
DAA	INH	18 07	0f0	0f0

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	
			HCS12X	HCS12
DBEQ *abdxys, rel9*	REL	`04 lb rr`	PPP/PP0	PPP/PP0

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	
			HCS12X	HCS12
DBNE *abdxys, rel9*	REL	`04 lb rr`	PPP/PP0	PPP/PP0

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	Access Detail	
			HCS12X	HCS12
DEC *opr16a*	EXT	73 hh ll	rPw0	rPw0
DEC *oprx0_xysp*	IDX	63 xb	rPw	rPw
DEC *oprx9,xysp*	IDX1	63 xb ff	rPw0	rPw0
DEC *oprx16,xysp*	IDX2	63 xb ee ff	frPwP	frPwP
DEC [D,*xysp*]	[D,IDX]	63 xb	fIfrPw	fIfrPw
DEC [*oprx16,xysp*]	[IDX2]	63 xb ee ff	fIPrPw	fIPrPw

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	
			HCS12X	HCS12
DECA	INH	43	0	0

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	
			HCS12X	HCS12
DECB	INH	53	0	0

DECW Decrement Memory DECW

Operation: (M : M + 1) – $0001 ⇒ M : M + 1

Description: Subtract one from the content of memory location M : M + 1.

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 $0000; 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 : M + 1) was $8000 before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
DECW *opr16a*	EXT	`18 73 hh ll`	`ORPWO`	NA
DECW *oprx0_xysp*	IDX	`18 63 xb`	`ORPW`	NA
DECW *oprx9,xysp*	IDX1	`18 63 xb ff`	`ORPWO`	NA
DECW *oprx16,xysp*	IDX2	`18 63 xb ee ff`	`OfRPWP`	NA
DECW [D,*xysp*]	[D,IDX]	`18 63 xb`	`OfIfRPW`	NA
DECW [*oprx16,xysp*]	[IDX2]	`18 63 xb ee ff18`	`OfIPRPW`	NA

DECX Decrement X DECX

Operation: (X) – $0001 ⇒ X

Description: Subtract one from the content of index register X.

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 $0000; 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 : M + 1) was $8000 before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
DECX	INH	18 43	00	NA

DECY Decrement Y DECY

Operation: (Y) – $0001 ⇒ Y

Description: Subtract one from the content of index register Y.

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 $0000; 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 : M + 1) was $8000 before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
DECY	INH	18 53	00	NA

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	
			HCS12X	HCS12
DES *translates to...* LEAS –1,SP	IDX	1B 9F	Pf	Pf

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	
			HCS12X	HCS12
DEX	INH	09	0	0

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	
			HCS12X	HCS12
DEY	INH	03	0	0

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 CPU12 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 > $FFFF; cleared otherwise Undefined after division by zero

C: Set if divisor was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
EDIV	INH	11	fffffffff0	fffffffff0

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 CPU12 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	
			HCS12X	HCS12
EDIVS	INH	18 14	0fffffffff0	0fffffffff0

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	
			HCS12X	HCS12
EMACS *opr16a*	Special	`18 12 hh ll`	`ORRORRWWP`	`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	
			HCS12X	HCS12
EMAXD *oprx0_xysp*	IDX	18 1A xb	ORPf	ORPf
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	OfIfRPf
EMAXD [*oprx16,xysp*]	[IDX2]	18 1A xb ee ff	OfIPRPf	OfIPRPf

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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
EMIND *oprx0_xysp*	IDX	18 1B xb	ORPf	ORPf
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	OfIfRPf
EMIND [*oprx16,xysp*]	[IDX2]	18 1B xb ee ff	OfIPRPf	OfIPRPf

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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
EMUL	INH	13	0	ff0

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	
			HCS12X	HCS12
EMULS	INH	18 13	0f0	0f0
			If followed by another Page 2 instruction	
			0ff0	0ff0

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	
			HCS12X	HCS12
EORA #*opr8i*	IMM	88 ii	P	P
EORA *opr8a*	DIR	98 dd	rPf	rPf
EORA *opr16a*	EXT	B8 hh ll	rPO	rPO
EORA *oprx0_xysp*	IDX	A8 xb	rPf	rPf
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	fIfrPf
EORA [*oprx16,xysp*]	[IDX2]	A8 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
EORB #*opr8i*	IMM	C8 ii	P	P
EORB *opr8a*	DIR	D8 dd	rPf	rPf
EORB *opr16a*	EXT	F8 hh ll	rPO	rPO
EORB *oprx0_xysp*	IDX	E8 xb	rPf	rPf
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	fIfrPf
EORB [*oprx16,xysp*]	[IDX2]	E8 xb ee ff	fIPrPf	fIPrPf

EORX Exclusive OR X EORX

Operation: (X) ⊕ (M : M + 1) ⇒ X

Description: Performs the logical exclusive OR between the content of index register X and the content of memory location M : M + 1. The result is placed in X. Each bit of X after the operation is the logical exclusive OR of the corresponding bits of M : M + 1 and X 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
EORX #*opr16i*	IMM	18 88 jj kk	OPO	NA
EORX *opr8a*	DIR	18 98 dd	ORPf	NA
EORX *opr16a*	EXT	18 B8 hh ll	ORPO	NA
EORX *oprx0_xysp*	IDX	18 A8 xb	ORPf	NA
EORX *oprx9,xysp*	IDX1	18 A8 xb ff	ORPO	NA
EORX *oprx16,xysp*	IDX2	18 A8 xb ee ff	OfRPP	NA
EORX [D,*xysp*]	[D,IDX]	18 A8 xb	OfIfRPf	NA
EORX [*oprx16,xysp*]	[IDX2]	18 A8 xb ee ff	OfIPRPf	NA

EORY Exclusive OR Y EORY

Operation: (Y) ⊕ (M : M + 1) ⇒ Y

Description: Performs the logical exclusive OR between the content of index register Y and the content of memory location M : M + 1. The result is placed in Y. Each bit of Y after the operation is the logical exclusive OR of the corresponding bits of M : M + 1 and Y 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
EORY #*opr16i*	IMM	18 C8 jj kk	OPO	NA
EORY *opr8a*	DIR	18 D8 dd	ORPf	NA
EORY *opr16a*	EXT	18 F8 hh ll	ORPO	NA
EORY *oprx0_xysp*	IDX	18 E8 xb	ORPf	NA
EORY *oprx9,xysp*	IDX1	18 E8 xb ff	ORPO	NA
EORY *oprx16,xysp*	IDX2	18 E8 xb ee ff	OfRPP	NA
EORY [D,*xysp*]	[D,IDX]	18 E8 xb	OfIfRPf	NA
EORY [*oprx16,xysp*]	[IDX2]	18 E8 xb ee ff	OfIPRPf	NA

ETBL Extended Table Lookup and Interpolate ETBL

Operation: $(M : M + 1) + [(B) \times ((M + 2 : M + 3) - (M : M + 1))] \Rightarrow D$

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ETBL *oprx0_xysp*	IDX	18 3F xb	ORRffffP	ORRfffffffP

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, 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	
			HCS12X	HCS12
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. Values are in hexadecimal.

EXG

Exchange Register Contents (Continued)

EXG

	MS⇒	8	9	A	B	C	D	E	F
⇓LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR_L⇔ A EXG CCR,A *EXG CCRL,A*	$TMP3_L$ ⇒ A $00:A ⇒ TMP3 EXG A, TMP3	B ⇔ A EXG D,A	X_L ⇒ A $00:A ⇒ X EXG X,A	Y_L ⇒ A $00:A ⇒ Y EXG Y,A	SP_L ⇒ A $00:A ⇒ SP EXG SP,A
1	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR_L ⇔ B EXG CCR,B *EXG CCRL,B*	$TMP3_L$ ⇒ B $FF:B ⇒ TMP3 EXG B,TMP3	B ⇒ B $FF ⇒ A EXG D,B	X_L ⇒ B $FF:B ⇒ X EXG X,B	Y_L ⇒ B $FF:B ⇒ Y EXG Y,B	SP_L ⇒ B $FF:B ⇒ SP EXG SP,B
2	CCR	A ⇔ CCR_L EXG A, CCR *EXG A,CCRL*	B ⇔ CCR_L EXG B,CCR *EXG B,CCRL*	CCR_L ⇔ CCR_L EXG CCR,CCR *EXG CCRL,CCRL*	$TMP3_L$ ⇒ CCR_L $FF:$CCR_L$ ⇒ TMP3 EXG, TMP3,CCR *EXG TMP3,CCRL*	B ⇒ CCR_L $FF:$CCR_L$ ⇒ D EXG D,CCR *EXG D,CCRL*	X_L ⇒ CCR_L $FF:$CCR_L$ ⇒ X EXG X,CCR *EXG X,CCRL*	Y_L ⇒ CCR_L $FF:$CCR_L$ ⇒ Y EXG Y,CCR *EXG Y,CCRL*	SP_L ⇒ CCR_L $FF:$CCR_L$ ⇒ SP EXG SP,CCR *EXG SP,CCRL*
3	TMP2	$00:A ⇒ TMP2 $TMP2_L$ ⇒ A EXG A,TMP2	$00:B ⇒ TMP2 $TMP2_L$ ⇒ B EXG B,TMP2	$00:$CCR_L$ ⇒ TMP2 $TMP2_L$ ⇒ CCR EXG CCR,TMP2	TMP3 ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP2 EXG D,TMP2	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
4	D	$00:A ⇒ D EXG A,D	$00:B ⇒ D EXG B,D	$00:$CCR_L$ ⇒ D B ⇒ CCR_L EXG CCR,D *EXG CCRL,D*	TMP3 ⇔ D EXG TMP3,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
5	X	$00:A ⇒ X X_L ⇒ A EXG A,X	$00:B ⇒ X X_L ⇒ B EXG B,X	$00:$CCR_L$ ⇒ X X_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
6	Y	$00:A ⇒ Y Y_L ⇒ A EXG A,Y	$00:B ⇒ Y Y_L ⇒ B EXG B,Y	$00:$CCR_L$ ⇒ Y Y_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
7	SP	$00:A ⇒ SP SP_L ⇒ A EXG A,SP	$00:B ⇒ SP SP_L ⇒ B EXG B,SP	$00:$CCR_L$ ⇒ SP SP_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP
8	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR_H ⇔ A EXG CCRH,A	$TMP3_H$ ⇔ A EXG TMP3H,A	B ⇔ A EXG D,A	X_H ⇔ A EXG XH,A	Y_H ⇔ A EXG YH,A	SP_H ⇔ A EXG SPH,A
9	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR_L ⇔ B EXG CCRL,B	$TMP3_L$ ⇔ B EXG TMP3L,B	$FF ⇒ A, B ⇒ B EXG D,B	X_L ⇔ B EXG XL,B	Y_L ⇔ B EXG YL,B	SP_L ⇔ B EXG SPL,B
A	CCR	A ⇔ CCR_H EXG A,CCRH	B ⇔ CCR_L EXG B,CCRL	$CCR_{H:L}$ ⇔ $CCR_{H:L}$ EXG CCRW,CCRW	TMP3 ⇔ $CCR_{H:L}$ EXG TMP3,CCRW	D ⇔ $CCR_{H:L}$ EXG D,CCRW	X ⇔ $CCR_{H:L}$ EXG X,CCRW	Y ⇔ $CCR_{H:L}$ EXG Y,CCRW	SP ⇔ $CCR_{H:L}$ EXG, SP,CCRW
B	TMPx	A ⇔ $TMP2_H$ EXG A,TMP2H	B ⇔ $TMP2_L$ EXG B,TMP2L	$CCR_{H:L}$⇔ TMP2 EXG CCRW,TMP2	TMP3 ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP1 EXG D,TMP1	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
C	D	$00:A ⇒ D EXG A,D	$00:B ⇒ D EXG B,D	$CCR_{H:L}$ ⇔ D EXG CCRW,D	TMP1 ⇔ D EXG TMP1,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
D	X	A ⇔ X_H EXG A,XH	B ⇔ X_L EXG B,XL	$CCR_{H:L}$ ⇔ X EXG CCRW,X	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
E	Y	A ⇔ Y_H EXG A,YH	B ⇔ Y_L EXG B,YL	$CCR_{H:L}$ ⇔ Y EXG CCRW,Y	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
F	SP	A ⇔ SP_H EXG A,SPH	B ⇔ SP_L EXG B,SPL	$CCR_{H:L}$ ⇔ SP EXG CCRW,SP	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the S12X.

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 ≤ 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	
			HCS12X	HCS12
FDIV	INH	18 11	0fffffffff0	0fffffffff0

GLDAA — Load Accumulator A from Global Memory — GLDAA

Operation: G(M) ⇒ A

Description: Loads the content of global memory location M into accumulator A. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GLDAA *opr8a*	DIR	18 96 dd	OrPf	NA
GLDAA *opr16a*	EXT	18 B6 hh ll	OrPO	NA
GLDAA *oprx0_xysp*	IDX	18 A6 xb	OrPf	NA
GLDAA *oprx9,xysp*	IDX1	18 A6 xb ff	OrPO	NA
GLDAA *oprx16,xysp*	IDX2	18 A6 xb ee ff	OfrPP	NA
GLDAA [D,*xysp*]	[D,IDX]	18 A6 xb	OfIfrPf	NA
GLDAA [*oprx16,xysp*]	[IDX2]	18 A6 xb ee ff	OfIPrPf	NA

GLDAB — Load Accumulator B from Global Memory — GLDAB

Operation: G(M) ⇒ B

Description: Loads the content of global memory location M into accumulator B. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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 HCS12X	Access Detail HCS12
GLDAB *opr8a*	DIR	18 D6 dd	OrPf	NA
GLDAB *opr16a*	EXT	18 F6 hh ll	OrPO	NA
GLDAB *oprx0_xysp*	IDX	18 E6 xb	OrPf	NA
GLDAB *oprx9,xysp*	IDX1	18 E6 xb ff	OrPO	NA
GLDAB *oprx16,xysp*	IDX2	18 E6 xb ee ff	OfrPP	NA
GLDAB [D,*xysp*]	[D,IDX]	18 E6 xb	OfIfrPf	NA
GLDAB [*oprx16,xysp*]	[IDX2]	18 E6 xb ee ff	OfIPrPf	NA

GLDD — Load Double Accumulator D (A : B) from Global Memory — GLDD

Operation: G(M : M + 1) ⇒ A : B

Description: Loads the content of global memory location M : M + 1 into double accumulator D. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GLDD *opr8a*	DIR	18 DC dd	`ORPf`	NA
GLDD *opr16a*	EXT	18 FC hh ll	`ORPO`	NA
GLDD *oprx0_xysp*	IDX	18 EC xb	`ORPf`	NA
GLDD *oprx9,xysp*	IDX1	18 EC xb ff	`ORPO`	NA
GLDD *oprx16,xysp*	IDX2	18 EC xb ee ff	`OfRPP`	NA
GLDD [D,*xysp*]	[D,IDX]	18 EC xb	`OfIfRPf`	NA
GLDD [*oprx16,xysp*]	[IDX2]	18 EC xb ee ff	`OfIPRPf`	NA

GLDS — Load Stack Pointer from Global Memory — GLDS

Operation: G(M : M + 1) ⇒ SP

Description: Loads the content of global memory location M : M + 1 into stack pointer SP. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GLDS *opr8a*	DIR	`18 DF dd`	`ORPf`	NA
GLDS *opr16a*	EXT	`18 FF hh ll`	`ORPO`	NA
GLDS *oprx0_xysp*	IDX	`18 EF xb`	`ORPf`	NA
GLDS *oprx9,xysp*	IDX1	`18 EF xb ff`	`ORPO`	NA
GLDS *oprx16,xysp*	IDX2	`18 EF xb ee ff`	`OfRPP`	NA
GLDS [D,*xysp*]	[D,IDX]	`18 EF xb`	`OfIfRPf`	NA
GLDS [*oprx16,xysp*]	[IDX2]	`18 EF xb ee ff`	`OfIPRPf`	NA

GLDX — Load Stack Index Register X from Global Memory — GLDX

Operation: G(M : M + 1) ⇒ X

Description: Loads the content of global memory location M : M + 1 into index register X. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GLDX *opr8a*	DIR	`18 DE dd`	`ORPf`	NA
GLDX *opr16a*	EXT	`18 FE hh ll`	`ORPO`	NA
GLDX *oprx0_xysp*	IDX	`18 EE xb`	`ORPf`	NA
GLDX *oprx9,xysp*	IDX1	`18 EE xb ff`	`ORPO`	NA
GLDX *oprx16,xysp*	IDX2	`18 EE xb ee ff`	`OfRPP`	NA
GLDX [D,*xysp*]	[D,IDX]	`18 EE xb`	`OfIfRPf`	NA
GLDX [*oprx16,xysp*]	[IDX2]	`18 EE xb ee ff`	`OfIPRPf`	NA

GLDY — Load Stack Index Register Y from Global Memory — GLDY

Operation: G(M : M + 1) ⇒ Y

Description: Loads the content of global memory location M : M + 1 into index register Y. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GLDY *opr8a*	DIR	`18 DD dd`	`ORPf`	NA
GLDY *opr16a*	EXT	`18 FD hh ll`	`ORPO`	NA
GLDY *oprx0_xysp*	IDX	`18 ED xb`	`ORPf`	NA
GLDY *oprx9,xysp*	IDX1	`18 ED xb ff`	`ORPO`	NA
GLDY *oprx16,xysp*	IDX2	`18 ED xb ee ff`	`OfRPP`	NA
GLDY [D,*xysp*]	[D,IDX]	`18 ED xb`	`OfIfRPf`	NA
GLDY [*oprx16,xysp*]	[IDX2]	`18 ED xb ee ff`	`OfIPRPf`	NA

GSTAA — Store Accumulator A to Global Memory — GSTAA

Operation: (A) ⇒ G(M)

Description: Stores the content of accumulator A into global memory location M. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTAA *opr8a*	DIR	18 5A dd	OPw	NA
GSTAA *opr16a*	EXT	18 7A hh ll	OPwO	NA
GSTAA *oprx0_xysp*	IDX	18 6A xb	OPw	NA
GSTAA *oprx9,xysp*	IDX1	18 6A xb ff	OPwO	NA
GSTAA *oprx16,xysp*	IDX2	18 6A xb ee ff	OPwP	NA
GSTAA [D,*xysp*]	[D,IDX]	18 6A xb	OPIfw	NA
GSTAA [*oprx16,xysp*]	[IDX2]	18 6A xb ee ff	OPIPw	NA

GSTAB — Store Accumulator B to Global Memory — GSTAB

Operation: (B) ⇒ G(M)

Description: Stores the content of accumulator B into global memory location M. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTAB *opr8a*	DIR	18 5B dd	OPw	NA
GSTAB *opr16a*	EXT	18 7B hh ll	OPwO	NA
GSTAB *oprx0_xysp*	IDX	18 6B xb	OPw	NA
GSTAB *oprx9,xysp*	IDX1	18 6B xb ff	OPwO	NA
GSTAB *oprx16,xysp*	IDX2	18 6B xb ee ff	OPwP	NA
GSTAB [D,*xysp*]	[D,IDX]	18 6B xb	OPIfw	NA
GSTAB [*oprx16,xysp*]	[IDX2]	18 6B xb ee ff	OPIPw	NA

GSTD — Store Double Accumulator to Global Memory — GSTD

Operation: (A) ⇒ G(M), (B) ⇒ G(M + 1)

Description: Stores the content of double accumulator D into global memory location M : M + 1. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTD *opr8a*	DIR	18 5C dd	OPW	NA
GSTD *opr16a*	EXT	18 7C hh ll	OPWO	NA
GSTD *oprx0_xysp*	IDX	18 6C xb	OPW	NA
GSTD *oprx9,xysp*	IDX1	18 6C xb ff	OPWO	NA
GSTD *oprx16,xysp*	IDX2	18 6C xb ee ff	OPWP	NA
GSTD [D,*xysp*]	[D,IDX]	18 6C xb	OPIfW	NA
GSTD [*oprx16,xysp*]	[IDX2]	18 6C xb ee ff	OPIPW	NA

GSTS — Store Stack Pointer to Global Memory — GSTS

Operation: (SP) ⇒ G(M : M + 1)

Description: Stores the content of stack pointer SP into global memory location M : M+ 1. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTS *opr8a*	DIR	`18 5F dd`	`OPW`	`NA`
GSTS *opr16a*	EXT	`18 7F hh ll`	`OPWO`	`NA`
GSTS *oprx0_xysp*	IDX	`18 6F xb`	`OPW`	`NA`
GSTS *oprx9,xysp*	IDX1	`18 6F xb ff`	`OPWO`	`NA`
GSTS *oprx16,xysp*	IDX2	`18 6F xb ee ff`	`OPWP`	`NA`
GSTS [D,*xysp*]	[D,IDX]	`18 6F xb`	`OPIfW`	`NA`
GSTS [*oprx16,xysp*]	[IDX2]	`18 6F xb ee ff`	`OPIPW`	`NA`

GSTX — Store Index Register X to Global Memory — GSTX

Operation: (X) ⇒ G(M :M + 1)

Description: Stores the content of index register X into global memory location M : M + 1. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTX *opr8a*	DIR	18 5E dd	OPW	NA
GSTX *opr16a*	EXT	18 7E hh ll	OPWO	NA
GSTX *oprx0_xysp*	IDX	18 6E xb	OPW	NA
GSTX *oprx9,xysp*	IDX1	18 6E xb ff	OPWO	NA
GSTX *oprx16,xysp*	IDX2	18 6E xb ee ff	OPWP	NA
GSTX [D,*xysp*]	[D,IDX]	18 6E xb	OPIfW	NA
GSTX [*oprx16,xysp*]	[IDX2]	18 6E xb ee ff	OPIPW	NA

GSTY — Store Index Register Y to Global Memory — GSTY

Operation: (Y) ⇒ G(M : M + 1)

Description: Stores the content of index register Y into global memory location M. The condition codes are set according to the data.

A global memory reference appends the contents of the GPAGE register to the most significant byte of the effective address to form a 23-bit address.

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	
			HCS12X	HCS12
GSTY *opr8a*	DIR	18 5D dd	OPW	NA
GSTY *opr16a*	EXT	18 7D hh ll	OPWO	NA
GSTY *oprx0_xysp*	IDX	18 6D xb	OPW	NA
GSTY *oprx9,xysp*	IDX1	18 6D xb ff	OPWO	NA
GSTY *oprx16,xysp*	IDX2	18 6D xb ee ff	OPWP	NA
GSTY [D,*xysp*]	[D,IDX]	18 6D xb	OPIfW	NA
GSTY [*oprx16,xysp*]	[IDX2]	18 6D xb ee ff	OPIPW	NA

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	
			HCS12X	HCS12
IBEQ *abdxys, rel9*	REL	`04 lb rr`	PPP/PP0	PPP/PP0

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	
			HCS12X	HCS12
IBNE *abdxys, rel9*	REL	`04 lb rr`	PPP/PPO	PPP/PPO

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: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet ... \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$
Set if denominator was $0000; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
IDIV	INH	18 10	Offffffffff0	Offffffffff0

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 ($7FFFF).

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	
			HCS12X	HCS12
IDIVS	INH	18 15	0fffffffff0	0fffffffff0

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	
			HCS12X	HCS12
INC *opr16a*	EXT	72 hh ll	rPw0	rPw0
INC *oprx0_xysp*	IDX	62 xb	rPw	rPw
INC *oprx9,xysp*	IDX1	62 xb ff	rPw0	rPw0
INC *oprx16,xysp*	IDX2	62 xb ee ff	frPwP	frPwP
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	
			HCS12X	HCS12
INCA	INH	42	0	0

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	
			HCS12X	HCS12
INCB	INH	52	0	0

INCW Increment Memory INCW

Operation: (M : M +1) + $0001 ⇒ M : M + 1

Description: Add one to the content of memory location M : M + 1.

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 $0000; 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 : M + 1) was $7FFF before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
INCW *opr16a*	EXT	`18 72 hh ll`	`ORPWO`	NA
INCW *oprx0_xysp*	IDX	`18 62 xb`	`ORPW`	NA
INCW *oprx9,xysp*	IDX1	`18 62 xb ff`	`ORPWO`	NA
INCW *oprx16,xysp*	IDX2	`18 62 xb ee ff`	`OfRPWP`	NA
INCW [D,*xysp*]	[D,IDX]	`18 62 xb`	`OfIfRPW`	NA
INCW [*oprx16,xysp*]	[IDX2]	`18 62 xb ee ff`	`OfIPRPW`	NA

INCX Increment Index Register X INCX

Operation: (X) + $0001 ⇒ X

Description: Add one to the content of index register X.

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 $0000; 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 : M + 1) was $7FFF before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
INCX	INH	18 42	00	NA

INCY Increment Index Register Y INCY

Operation: (Y) + $0001 ⇒ Y

Description: Add one to the content of index register Y.

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 $0000; 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 : M + 1) was $7FFF before the operation.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
INCY	INH	18 52	00	NA

INS — Increment Stack Pointer — INS

Operation: (SP) + $0001 ⇒ SP

Description: Add one to the stack pointer 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	
			HCS12X	HCS12
INS *translates to...* LEAS 1,SP	IDX	1B 81	Pf	Pf

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	
			HCS12X	HCS12
INX	INH	08	0	0

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	
			HCS12X	HCS12
INY	INH	02	0	0

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	
			HCS12X	HCS12
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
$RTN_H : RTN_L ⇒ M_{(SP)} : M_{(SP + 1)}$
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.

For SP relative auto pre/post decrement/increment indexed addressing modes, the effective address of the jump is calculated firsts, then SP adjustments associated with the stacking operation.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
JSR *opr8a*	DIR	17 dd	SPPP	SPPP
JSR *opr16a*	EXT	16 hh ll	SPPP	SPPP
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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 ⊕ V = 0, (PC) + $0004 + Rel ⇒ 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 CPU12 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 **3.9 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 HCS12X	Access Detail HCS12
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 CPU12 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 **3.9 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 HCS12X	Access Detail HCS12
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 ⇒ 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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 ⊕ 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

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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 CPU12 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 **3.9 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 HCS12X	Access Detail HCS12
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 ⇒ 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 CPU12 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 **3.9 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 HCS12X	Access Detail HCS12
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 CPU12 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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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 HCS12X	Access Detail HCS12
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 **3.9 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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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 **3.9 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	
			HCS12X	HCS12
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
–	–	–	–	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	
			HCS12X	HCS12
LDAA #*opr8i*	IMM	86 ii	P	P
LDAA *opr8a*	DIR	96 dd	rPf	rPf
LDAA *opr16a*	EXT	B6 hh ll	rPO	rPO
LDAA *oprx0_xysp*	IDX	A6 xb	rPf	rPf
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	fIfrPf
LDAA [*oprx16,xysp*]	[IDX2]	A6 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
LDAB #*opr8i*	IMM	C6 ii	P	P
LDAB *opr8a*	DIR	D6 dd	rPf	rPf
LDAB *opr16a*	EXT	F6 hh ll	rPO	rPO
LDAB *oprx0_xysp*	IDX	E6 xb	rPf	rPf
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	fIfrPf
LDAB [*oprx16*,*xysp*]	[IDX2]	E6 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
LDD #*opr16i*	IMM	CC jj kk	PO	PO
LDD *opr8a*	DIR	DC dd	RPf	RPf
LDD *opr16a*	EXT	FC hh ll	RPO	RPO
LDD *oprx0_xysp*	IDX	EC xb	RPf	RPf
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	fIfRPf
LDD [*oprx16,xysp*]	[IDX2]	EC xb ee ff	fIPRPf	fIPRPf

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 : M + 1, and loads the least significant byte of the SP with the content of the next byte of memory at M : 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	
			HCS12X	HCS12
LDS #*opr16i*	IMM	CF jj kk	PO	PO
LDS *opr8a*	DIR	DF dd	RPf	RPf
LDS *opr16a*	EXT	FF hh ll	RPO	RPO
LDS *oprx0_xysp*	IDX	EF xb	RPf	RPf
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	fIfRPf
LDS [*oprx16,xysp*]	[IDX2]	EF xb ee ff	fIPRPf	fIPRPf

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	
			HCS12X	HCS12
LDX #*opr16i*	IMM	CE jj kk	P0	P0
LDX *opr8a*	DIR	DE dd	RPf	RPf
LDX *opr16a*	EXT	FE hh ll	RP0	RP0
LDX *oprx0_xysp*	IDX	EE xb	RPf	RPf
LDX *oprx9,xysp*	IDX1	EE xb ff	RP0	RP0
LDX *oprx16,xysp*	IDX2	EE xb ee ff	fRPP	fRPP
LDX [D,*xysp*]	[D,IDX]	EE xb	fIfRPf	fIfRPf
LDX [*oprx16,xysp*]	[IDX2]	EE xb ee ff	fIPRPf	fIPRPf

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	
			HCS12X	HCS12
LDY #*opr16i*	IMM	CD jj kk	PO	PO
LDY *opr8a*	DIR	DD dd	RPf	RPf
LDY *opr16a*	EXT	FD hh ll	RPO	RPO
LDY *oprx0_xysp*	IDX	ED xb	RPf	RPf
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	fIfRPf
LDY [*oprx16,xysp*]	[IDX2]	ED xb ee ff	fIPRPf	fIPRPf

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 **3.10 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	
			HCS12X	HCS12
LEAS *oprx0_xysp*	IDX	1B xb	Pf	Pf
LEAS *oprx9,xysp*	IDX1	1B xb ff	P0	P0
LEAS *oprx16,xysp*	IDX2	1B xb ee ff	PP	PP

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 **3.10 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	
			HCS12X	HCS12
LEAX *oprx0_xysp*	IDX	1A xb	Pf	Pf
LEAX *oprx9,xysp*	IDX1	1A xb ff	P0	P0
LEAX *oprx16,xysp*	IDX2	1A xb ee ff	PP	PP

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 **3.10 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
–	–	–	–	–	–	–	–

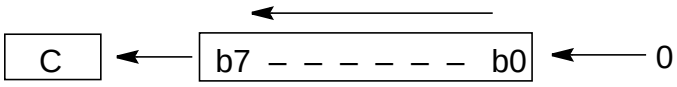
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LEAY *oprx0_xysp*	IDX	19 xb	Pf	Pf
LEAY *oprx9,xysp*	IDX1	19 xb ff	P0	P0
LEAY *oprx16,xysp*	IDX2	19 xb ee ff	PP	PP

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

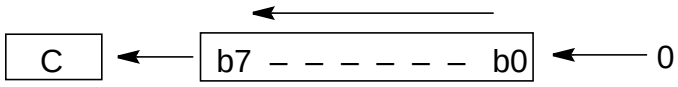
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSL *opr16a*	EXT	`78 hh ll`	`rPw0`	`rPw0`
LSL *oprx0_xysp*	IDX	`68 xb`	`rPw`	`rPw`
LSL *oprx9,xysp*	IDX1	`68 xb ff`	`rPw0`	`rPw0`
LSL *oprx16,xysp*	IDX2	`68 xb ee ff`	`frPwP`	`frPwP`
LSL [D,*xysp*]	[D,IDX]	`68 xb`	`fIfrPw`	`fIfrPw`
LSL [*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

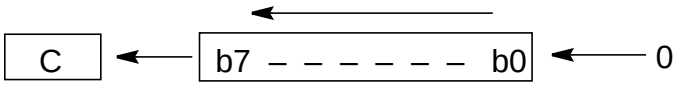
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSLA	INH	48	0	0

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

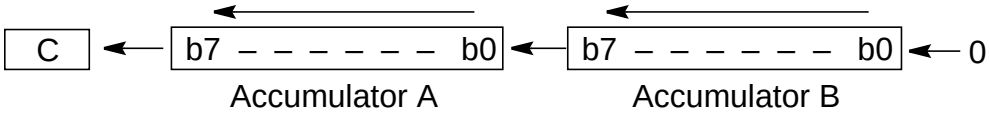
Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSLB	INH	58	0	0

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

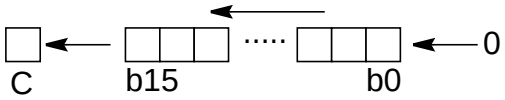
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSLD	INH	59	0	0

LSLW — Logical Shift Left W (Same as ASLW) — LSLW

Operation:



Description: Shifts all bits of memory location M : M + 1 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 W.

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

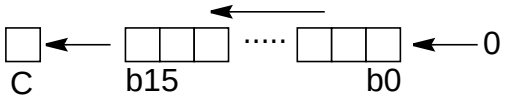
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M15
Set if the MSB of M was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
LSLW *opr16a*	EXT	18 78 hh ll	ORPWO	NA
LSLW *oprx0_xysp*	IDX	18 68 xb	ORPW	NA
LSLW *oprx9,xysp*	IDX1	18 68 xb ff	ORPWO	NA
LSLW *oprx16,xysp*	IDX2	18 68 xb ee ff	OfRPWP	NA
LSLW [D,*xysp*]	[D,IDX]	18 68 xb	OfIfRPW	NA
LSLW [*oprx16,xysp*]	[IDX2]	18 68 xb ee ff	OfIPRPW	NA

LSLX — Logic Shift Left Index Register X (Same as ASLX) — LSLX

Operation:



Description: Shifts all bits of index register X 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 X.

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

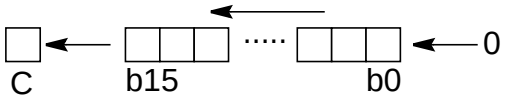
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X15
Set if the MSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSLX	INH	18 48	00	NA

LSLY — Logical Shift Left Index Register Y (Same as ASLY) — LSLY

Operation:



Description: Shifts all bits of index register Y 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 Y.

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

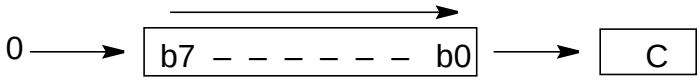
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y15
Set if the MSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
LSLY	INH	18 58	00	NA

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

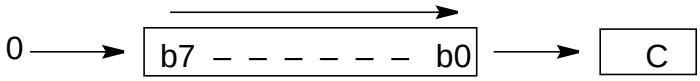
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSR *opr16a*	EXT	74 hh ll	rPw0	rPw0
LSR *oprx0_xysp*	IDX	64 xb	rPw	rPw
LSR *oprx9,xysp*	IDX1	64 xb ff	rPw0	rPw0
LSR *oprx16,xysp*	IDX2	64 xb ee ff	frPwP	frPwP
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

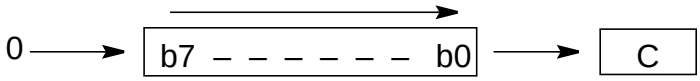
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRA	INH	44	0	0

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

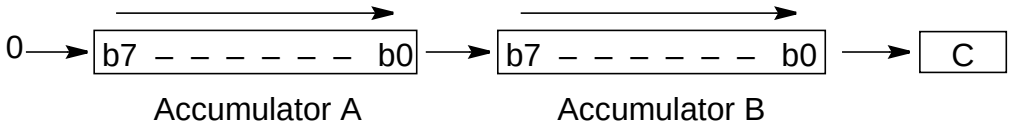
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRB	INH	54	0	0

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

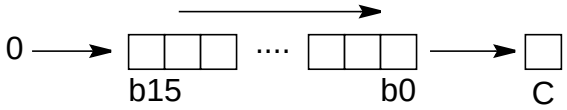
V: D0
Set if, after the shift operation, C is set; cleared otherwise

C: D0
Set if the LSB of D was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRD	INH	49	0	0

LSRW Logical Shift Right Memory LSRW

Operation:



Description: Shifts all bits of memory location M : M + 1 one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of M : M + 1.

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

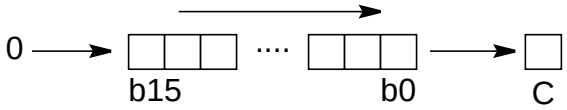
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M : M + 1 was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRW *opr16a*	EXT	`18 74 hh ll`	`ORPW0`	NA
LSRW *oprx0_xysp*	IDX	`18 64 xb`	`ORPW`	NA
LSRW *oprx9,xysp*	IDX1	`18 64 xb ff`	`ORPW0`	NA
LSRW *oprx16,xysp*	IDX2	`18 64 xb ee ff`	`OfRPWP`	NA
LSRW [D,*xysp*]	[D,IDX]	`18 64 xb`	`OfIfRPW`	NA
LSRW [*oprx16,xysp*]	[IDX2]	`18 64 xb ee ff`	`OfIPRPW`	NA

LSRX — Logical Shift Index Register X to Right — LSRX

Operation:



Description: Shifts all bits of index register X one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of X.

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

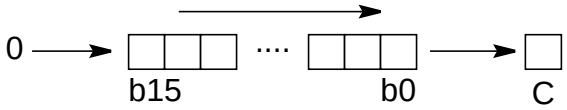
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X0
Set if the LSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRX	INH	18 44	00	NA

LSRY — Logical Shift Index Register Y to Right — LSRY

Operation:



Description: Shifts all bits of index register Y one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of Y.

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: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (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: Y0
Set if the LSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
LSRY	INH	18 54	00	NA

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 HCS12X	Access Detail HCS12
MAXA *oprx0_xysp*	IDX	18 18 xb	OrPf	OrPf
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	OfIfrPf
MAXA [*oprx16,xysp*]	[IDX2]	18 18 xb ee ff	OfIPrPf	OfIPrPf

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	
			HCS12X	HCS12
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 $\Rightarrow M_{(Y)}$
(Y) + $0001 $\Rightarrow$ Y
(X) + $0004 $\Rightarrow$ 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)
- Slope_1 — The slope of the leading side (at M_{X+1})
- Point_2 — The x-axis position of the rightmost point (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	HCS12
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	
			HCS12X	HCS12
MINA *oprx0_xysp*	IDX	18 19 xb	OrPf	OrPf
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	OfIfrPf
MINA [*oprx16,xysp*]	[IDX2]	18 19 xb ee ff	OfIPrPf	OfIPrPf

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	
			HCS12X	HCS12
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

Immediate-to-Memory Byte Move (8 Bit)

MOVB

Operation: # ⇒ M

Description: Moves the immediate value # to memory location M.

Move byte instructions specify the source first and destination second in the object code for an immediate value source and an extended addressing mode destination. Move byte instructions using immediate values for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for HCS12 and HC12 compatibility.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB #*oprx8i, opr16a*[1]	EXT	18 0B ii hh ll	PwP	OPwP
MOVB *#opr8i, oprx0_xysp*[1]	IDX	18 08 **xb**[2] ii	PwO	OPwO
MOVB *#opr8i, oprx9_xysp*[1]	IDX1	18 08 **xb**[2] ff ii	PwP	NA
MOVB *#opr8i, oprx16_xysp*[1]	IDX2	18 08 **xb**[2] ee ff ii	PPwO	NA
MOVB *#opr8i, [D_xysp]*[1]	[D,IDX]	18 08 **xb**[2] ii	PIOw	NA
MOVB *#opr8i, [oprx16_xysp]*[1]	[IDX2]	18 08 **xb**[2] ee ff ii	PIOwP	NA

1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

MOVB

Memory-to-Memory Byte Move
EXT Source (8 Bit)

MOVB

Operation: $(M_1) \Rightarrow M_2$
EXT Source ⇒ Address Mode Destination

Description: Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for an extended addressing mode source and an extended addressing mode destination. Move byte instructions using extended addressing for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for HCS12 and HC12 compatibility.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB *opr16a, opr16a*[1]	EXT	18 0C hh ll hh ll	PrPwO	OrPwPO
MOVB *opr16a, oprx0_xysp*[1]	IDX	18 09 **xb**[2] hh ll	PrPw	OPrPw
MOVB *opr16a, oprx9_xysp*[1]	IDX1	18 09 **xb**[2] ff hh ll	PrPwO	NA
MOVB *opr16a, oprx16_xysp*[1]	IDX2	18 09 **xb**[2] ee ff hh ll	PPrPw	NA
MOVB *opr16a, [D_xysp]*[1]	[D,IDX]	18 09 **xb**[2] hh ll	PrIPw	NA
MOVB *opr16a, [oprx16_xysp]*[1]	[IDX2]	18 09 **xb**[2] ee ff hh ll	PPrIPw	NA

1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

MOVB

Memory-to-Memory Byte Move
IDX Source (8 Bit)

MOVB

Operation: $(M_1) \Rightarrow M_2$
IDX Source ⇒ Address Mode Destination

Description: Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form[(1)]	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB *oprx0_xysp, opr16a*[1]	EXT	18 0D xb hh ll	rPPw	OrPwP
MOVB *oprx0_xysp, oprx0_xysp*[1]	IDX	18 0A xb xb	rPOw	OrPwO
MOVB *oprx0_xysp, oprx9_xysp*[1]	IDX1	18 0A xb xb ff	rPPw	NA
MOVB *oprx0_xysp, oprx16_xysp*[1]	IDX2	18 0A xb xb ee ff	rPOPw	NA
MOVB *oprx0_xysp, [D_xysp]*[1]	[D,IDX]	18 0A xb xb	rPIOw	NA
MOVB *oprx0_xysp, [oprx16_xysp]*[1]	[IDX2]	18 0A xb xb ee ff	rPPIOw	NA

1. The first operand in the source code statement specifies the source for the move.

MOVB — Memory-to-Memory Byte Move IDX1 Source (8 Bit) — MOVB

Operation: $(M_1) \Rightarrow M_2$
IDX1 Source ⇒ Address Mode Destination

Description: Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB *oprx9_xysp, opr16a*[1]	EXT	18 0D xb ff hh ll	PrOPw	NA
MOVB *oprx9_xysp, oprx0_xysp*[1]	IDX	18 0A xb ff xb	PrOOw	NA
MOVB *oprx9_xysp, oprx9_xysp*[1]	IDX1	18 0A xb ff xb ff	PrOPw	NA
MOVB *oprx9_xysp, oprx16_xysp*[1]	IDX2	18 0A xb ff xb ee ff	PrOOPw	NA
MOVB *oprx9_xysp, [D_xysp]*[1]	[D,IDX]	18 0A xb ff xb	PrOIOw	NA
MOVB *oprx9_xysp, [oprx16_xysp]*[1]	[IDX2]	18 0A xb ff xb ee ff	PrOPIOw	NA

1. The first operand in the source code statement specifies the source for the move.

MOVB — Memory-to-Memory Byte Move IDX2 Source (8 Bit) — MOVB

Operation:

$(M_1) \Rightarrow M_2$
IDX2 Source ⇒ Address Mode Destination

Description:

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB *oprx16_xysp, opr16a*[1]	EXT	18 0D xb ee ff hh ll	PrPPw	NA
MOVB *oprx16_xysp, oprx0_xysp*[1]	IDX	18 0A xb ee ff xb	PrPOw	NA
MOVB *oprx16_xysp, oprx9_xysp*[1]	IDX1	18 0A xb ee ff xb ff	PrPPw	NA
MOVB *oprx16_xysp, oprx16_xysp*[1]	IDX2	18 0A xb ee ff xb ee ff	PrPOPw	NA
MOVB *oprx16_xysp, [D_xysp]*[1]	[D,IDX]	18 0A xb ee ff xb	PrPIOw	NA
MOVB *oprx16_xysp, [oprx16_xysp]*[1]	[IDX2]	18 0A xb ee ff xb ee ff	PrPPIO	NA

1. The first operand in the source code statement specifies the source for the move.

MOVB — Memory-to-Memory Byte Move [D,IDX] Source (8 Bit) — MOVB

Operation: $(M_1) \Rightarrow M_2$
[D,IDX] Source ⇒ Address Mode Destination

Description: Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVB *[D_xysp], opr16a*[1]	EXT	18 0D xb hh ll	IPrfPw	NA
MOVB *[D_xysp], oprx0_xysp*[1]	IDX	18 0A xb xb	IPrfOw	NA
MOVB *[D_xysp], oprx9_xysp*[1]	IDX1	18 0A xb xb ff	IPrfPw	NA
MOVB *[D_xysp], oprx16_xysp*[1]	IDX2	18 0A xb xb ee ff	IPrfOPw	NA
MOVB *[D_xysp], [D_xysp]*[1]	[D,IDX]	18 0A xb xb	IPrfIOw	NA
MOVB *[D_xysp], [oprx16_xysp]*[1]	[IDX2]	18 0A xb xb ee ff	IPrfPIOw	NA

1. The first operand in the source code statement specifies the source for the move.

MOVB — Memory-to-Memory Byte Move [IDX2] Source (8 Bit) — MOVB

Operation: $(M_1) \Rightarrow M_2$
[IDX2] Source ⇒ Address Mode Destination

Description: Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
MOVB *[oprx16_xysp], opr16a*[1]	EXT	18 0D xb ee ff hh ll	PIPrfPw	NA
MOVB *[oprx16_xysp], oprx0_xysp*[1]	IDX	18 0A xb ee ff xb	PIPrfOw	NA
MOVB *[oprx16_xysp], oprx9_xysp*[1]	IDX1	18 0A xb ee ff xb ff	PIPrfPw	NA
MOVB *[oprx16_xysp], oprx16_xysp*[1]	IDX2	18 0A xb ee ff xb ee ff	PIPrfOPw	NA
MOVB *[oprx16_xysp], [D_xysp]*[1]	[D,IDX]	18 0A xb ee ff xb	PIPrfIOw	NA
MOVB *[oprx16_xysp], [oprx16_xysp]*[1]	[IDX2]	18 0A xb ee ff xb ee ff	PIPrfPIOw	NA

1. The first operand in the source code statement specifies the source for the move.

MOVW — Immediate-to-Memory Word Move (16 Bit) — MOVW

Operation: $\# \Rightarrow M : M + 1_2$

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 word instructions specify the source first and destination second in the object code for an immediate value source and an extended addressing mode destination. Move word instructions using immediate values for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for HCS12 and HC12 compatibility.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW #*oprx16i, opr16a*[1]	EXT	18 03 jj kk hh ll	PWPO	OPWPO
MOVW #*opr16i, oprx0_xysp*[1]	IDX	18 00 xb[2] jj kk	PWP	OPPW
MOVW #*opr16i, oprx9_xysp*[1]	IDX1	18 00 xb[2] ff jj kk	PPWO	NA
MOVW #*opr16i, oprx16_xysp*[1]	IDX2	18 00 xb[2] ee ff jj kk	PPWP	NA
MOVW #*opr16i, [D_xysp]*[1]	[D,IDX]	18 00 xb[2] jj kk	PIPW	NA
MOVW #*opr16i, [oprx16_xysp]*[1]	[IDX2]	18 00 xb[2] ee ff jj kk	PIPWP	NA

1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

MOVW — Memory-to-Memory Word Move EXT Source (16 Bit) — MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$
EXT Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for an extended addressing mode source and an extended addressing mode destination. Move word instructions using extended addressing for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for HCS12 and HC12 compatibility.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *opr16a, opr16a*[1]	EXT	`18 04 hh ll hh ll`	PRPWO	ORPWPO
MOVW *opr16a, oprx0_xysp*[1]	IDX	`18 01 xb`[2] `hh ll`	PRPW	OPRPW
MOVW *opr16a, oprx9_xysp*[1]	IDX1	`18 01 xb`[2] `ff hh ll`	PRPWO	NA
MOVW *opr16a, oprx16_xysp*[1]	IDX2	`18 01 xb`[2] `ee ff hh ll`	PPRPW	NA
MOVW *opr16a, [D_xysp]*[1]	[D,IDX]	`18 01 xb`[2] `hh ll`	PRIPW	NA
MOVW *opr16a, [oprx16_xysp]*[1]	[IDX2]	`18 01 xb`[2] `ee ff hh ll`	PPRIPW	NA

1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

MOVW — Memory-to-Memory Word Move IDX Source (16 Bit) — MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$
IDX Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *oprx0_xysp, opr16a*[1]	EXT	18 05 xb hh ll	RPPW	ORPWP
MOVW *oprx0_xysp, oprx0_xysp*[1]	IDX	18 02 xb xb	RPOW	ORPWO
MOVW *oprx0_xysp, oprx9_xysp*[1]	IDX1	18 02 xb xb ff	RPPW	NA
MOVW *oprx0_xysp, oprx16_xysp*[1]	IDX2	18 02 xb xb ee ff	RPOPW	NA
MOVW *oprx0_xysp, [D_xysp]*[1]	[D,IDX]	18 02 xb xb	RPIOW	NA
MOVW *oprx0_xysp, [oprx16_xysp]*[1]	[IDX2]	18 02 xb xb ee ff	RPPIOW	NA

1. The first operand in the source code statement specifies the source for the move.

MOVW Memory-to-Memory Word Move IDX1 Source (16 Bit) MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$
IDX1 Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *oprx9_xysp, opr16a*[1]	EXT	`18 05 xb ff hh ll`	PROPW	NA
MOVW *oprx9_xysp, oprx0_xysp*[1]	IDX	`18 02 xb ff xb`	PROOW	NA
MOVW *oprx9_xysp, oprx9_xysp*[1]	IDX1	`18 02 xb ff xb ff`	PROPW	NA
MOVW *oprx9_xysp, oprx16_xysp*[1]	IDX2	`18 02 xb ff xb ee ff`	PROOPW	NA
MOVW *oprx9_xysp, [D_xysp]*[1]	[D,IDX]	`18 02 xb ff xb`	PROIOW	NA
MOVW *oprx9_xysp, [oprx16_xysp]*[1]	[IDX2]	`18 02 xb ff xb ee ff`	PROPIOW	NA

1. The first operand in the source code statement specifies the source for the move.

MOVW — Memory-to-Memory Word Move IDX2 Source (16 Bit) — MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$
IDX2 Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *oprx16_xysp, opr16a*[1]	EXT	18 05 xb ee ff hh ll	PRPPW	NA
MOVW *oprx16_xysp, oprx0_xysp*[1]	IDX	18 02 xb ee ff xb	PRPOW	NA
MOVW *oprx16_xysp, oprx9_xysp*[1]	IDX1	18 02 xb ee ff xb ff	PRPPW	NA
MOVW *oprx16_xysp, oprx16_xysp*[1]	IDX2	18 02 xb ee ff xb ee ff	PRPOPW	NA
MOVW *oprx16_xysp, [D_xysp]*[1]	[D,IDX]	18 02 xb ee ff xb	PRPIOW	NA
MOVW *oprx16_xysp, [oprx16_xysp]*[1]	[IDX2]	18 02 xb ee ff xb ee ff	PRPPIO	NA

1. The first operand in the source code statement specifies the source for the move.

MOVW — Memory-to-Memory Word Move [D,IDX] Source (16 Bit) — MOVW

Operation:

$(M : M + 1_1) \Rightarrow M : M + 1_2$
[D,IDX] Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *[D_xysp], opr16a*[1]	EXT	18 05 xb hh ll	IPRfPW	NA
MOVW *[D_xysp], oprx0_xysp*[1]	IDX	18 02 xb xb	IPRfOW	NA
MOVW *[D_xysp], oprx9_xysp*[1]	IDX1	18 02 xb xb ff	IPRfPW	NA
MOVW *[D_xysp], oprx16_xysp*[1]	IDX2	18 02 xb xb ee ff	IPRfOPW	NA
MOVW *[D_xysp], [D_xysp]*[1]	[D,IDX]	18 02 xb xb	IPRfIOW	NA
MOVW *[D_xysp], [oprx16_xysp]*[1]	[IDX2]	18 02 xb xb ee ff	IPRfPIOW	NA

1. The first operand in the source code statement specifies the source for the move.

MOVW — Memory-to-Memory Word Move [IDX2] Source (16 Bit) — MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$
[IDX2] Source ⇒ Address Mode Destination

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 word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated fist and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1
IDX2: +2
[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details:

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Source Form(1)	Destination Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
MOVW *[oprx16_xysp], opr16a*[1]	EXT	18 05 xb ee ff hh ll	PIPRfPW	
MOVW *[oprx16_xysp], oprx0_xysp*[1]	IDX	18 02 xb ee ff xb	PIPRfOW	NA
MOVW *[oprx16_xysp], oprx9_xysp*[1]	IDX1	18 02 xb ee ff xb ff	PIPRfPW	NA
MOVW *[oprx16_xysp], oprx16_xysp*[1]	IDX2	18 02 xb ee ff xb ee ff	PIPRfOPW	NA
MOVW *[oprx16_xysp], [D_xysp]*[1]	[D,IDX]	18 02 xb ee ff xb	PIPRfIOW	NA
MOVW *[oprx16_xysp], [oprx16_xysp]*[1]	[IDX2]	18 02 xb ee ff xb ee ff	PIPRfPIOW	NA
				NA

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	
			HCS12X	HCS12
MUL	INH	12	0	0

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.

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	
			HCS12X	HCS12
NEG *opr16a*	EXT	`70 hh ll`	`rPwO`	`rPwO`
NEG *oprx0_xysp*	IDX	`60 xb`	`rPw`	`rPw`
NEG *oprx9,xysp*	IDX1	`60 xb ff`	`rPwO`	`rPwO`
NEG *oprx16,xysp*	IDX2	`60 xb ee ff`	`frPwP`	`frPwP`
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.

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	
			HCS12X	HCS12
NEGA	INH	40	0	0

NEGB — Negate B — NEGB

Operation: $0 - (B) = (\overline{B}) + 1 \Rightarrow B$

Description: Replaces the content of accumulator B with its two's complement.

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	
			HCS12X	HCS12
NEGB	INH	50	0	0

NEGW Two's Complement Negate NEGW

Operation: $0 - (M : M + 1) \Rightarrow M : M + 1$ equivalent to $(\overline{M : M + 1}) + 1 \Rightarrow M : M + 1$

Description: Replaces the content of memory location M : M + 1 with its two's complement.

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: $R15 \bullet \overline{R14} \bullet \overline{R13} \bullet \overline{R12} \bullet \overline{R11} \bullet \overline{R10} \bullet \overline{R9} \bullet \overline{R8} \bullet \overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
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 : M + 1) = $8000

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when (M : M + 1) = $0000.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
NEGW *opr16a*	EXT	`18 70 hh ll`	`ORPWO`	NA
NEGW *oprx0_xysp*	IDX	`18 60 xb`	`ORPW`	NA
NEGW *oprx9,xysp*	IDX1	`18 60 xb ff`	`ORPWO`	NA
NEGW *oprx16,xysp*	IDX2	`18 60 xb ee ff`	`OfRPWP`	NA
NEGW [D,*xysp*]	[D,IDX]	`18 60 xb`	`OfIfRPW`	NA
NEGW [*oprx16,xysp*]	[IDX2]	`18 60 xb ee ff`	`OfIPRPW`	NA

NEGX — Negate Index Register X — NEGX

Operation: 0 – (X) ⇒ X equivalent to $(\overline{X})$ + 1 ⇒ X

Description: Replaces the content of index register X with its two's complement.

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: $R15 \bullet \overline{R14} \bullet \overline{R13} \bullet \overline{R12} \bullet \overline{R11} \bullet \overline{R10} \bullet \overline{R9} \bullet \overline{R8} \bullet \overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
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 : M + 1) = $8000

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when (M : M + 1) = $0000.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
NEGX	INH	18 40	OO	NA

NEGY — Negate Index Register Y — NEGY

Operation: 0 – (Y) ⇒ Y equivalent to $(\overline{Y})$ + 1 ⇒ Y

Description: Replaces the content of index register Y with its two's complement.

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: $R15 \bullet \overline{R14} \bullet \overline{R13} \bullet \overline{R12} \bullet \overline{R11} \bullet \overline{R10} \bullet \overline{R9} \bullet \overline{R8} \bullet \overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
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 : M + 1) = $8000

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when (M : M + 1) = $0000.

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
NEGY	INH	18 50	OO	NA

NOP Null Operation NOP

Operation: No operation

Description: This single-byte instruction increments the PC and does nothing else. No other CPU12 registers are affected. NOP is typically used to produce a time delay, although some software disciplines discourage CPU12 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	
			HCS12X	HCS12
NOP	INH	A7	0	0

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 HCS12X	Access Detail HCS12
ORAA #*opr8i*	IMM	8A ii	P	P
ORAA *opr8a*	DIR	9A dd	rPf	rPf
ORAA *opr16a*	EXT	BA hh ll	rPO	rPO
ORAA *oprx0_xysp*	IDX	AA xb	rPf	rPf
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	fIfrPf
ORAA [*oprx16,xysp*]	[IDX2]	AA xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ORAB #*opr8i*	IMM	CA ii	P	P
ORAB *opr8a*	DIR	DA dd	rPf	rPf
ORAB *opr16a*	EXT	FA hh ll	rPO	rPO
ORAB *oprx0_xysp*	IDX	EA xb	rPf	rPf
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	fIfrPf
ORAB [*oprx16,xysp*]	[IDX2]	EA xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
ORCC #*opr8i*	IMM	`14 ii`	P	P

ORX — Logic OR X with Memory — ORX

Operation: (X) | (M : M + 1) ⇒ X

Description: Performs bitwise logical inclusive OR between the content of index register X and the content of memory location M : M + 1 and places the result in X. Each bit of X after the operation is the logical inclusive OR of the corresponding bits of M : M + 1 and of X 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ORX #*opr16i*	IMM	18 8A jj kk	OPO	NA
ORX *opr8a*	DIR	18 9A dd	ORPf	NA
ORX *opr16a*	EXT	18 BA hh ll	ORPO	NA
ORX *oprx0_xysp*	IDX	18 AA xb	ORPf	NA
ORX *oprx9,xysp*	IDX1	18 AA xb ff	ORPO	NA
ORX *oprx16,xysp*	IDX2	18 AA xb ee ff	OfRPP	NA
ORX [D,*xysp*]	[D,IDX]	18 AA xb	OfIfRPf	NA
ORX [*oprx16,xysp*]	[IDX2]	18 AA xb ee ff	OfIPRPf	NA

ORY — Logic OR Y with Memory — ORY

Operation: (Y) | (M : M + 1) ⇒ Y

Description: Performs bitwise logical inclusive OR between the content of index register Y and the content of memory location M : M + 1 and places the result in Y. Each bit of Y after the operation is the logical inclusive OR of the corresponding bits of M : M + 1 and of Y 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 $0000; cleared otherwise

V: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ORY #*opr16i*	IMM	18 CA jj kk	OP	NA
ORY *opr8a*	DIR	18 DA dd	ORPf	NA
ORY *opr16a*	EXT	18 FA hh ll	ORPO	NA
ORY *oprx0_xysp*	IDX	18 EA xb	ORPf	NA
ORY *oprx9,xysp*	IDX1	18 EA xb ff	ORPO	NA
ORY *oprx16,xysp*	IDX2	18 EA xb ee ff	OfRPP	NA
ORY [D,*xysp*]	[D,IDX]	18 EA xb	OfIfRPf	NA
ORY [*oprx16,xysp*]	[IDX2]	18 EA xb ee ff	OfIPRPf	NA

PSHA Push A onto Stack PSHA

Operation: (SP) – $0001 ⇒ SP
(A) ⇒ $M_{(SP)}$

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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHA	INH	36	0s	0s

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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHB	INH	37	0s	0s

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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHC	INH	39	0s	0s

PSHCW Push CCR onto Stack PSHCW

Operation: (SP) – 2 ⇒ SP; (CCR_H:CCR_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$

Description: Stacks the content of the condition codes register. The stack pointer is decremented by two. 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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 registers just before returning from the subroutine.

CCR Details:

0	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
0	0	0	0	0	–	–	–	–	–	–	–	–	–

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
PSHCW	INH	18 39	00S	NA

PSHD Push Double Accumulator onto Stack PSHD

Operation: (SP) – $0002 ⇒ SP
(A : B) ⇒ $M_{(SP)}$: $M_{(SP+1)}$

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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHD	INH	3B	0S	0S

PSHX Push Index Register X onto Stack PSHX

Operation: (SP) – $0002 ⇒ SP
$(X_H : X_L) ⇒ M_{(SP)} : M_{(SP+1)}$

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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHX	INH	34	0S	0S

PSHY — Push Index Register Y onto Stack — PSHY

Operation: (SP) – $0002 ⇒ 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 CPU12 registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU12 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	
			HCS12X	HCS12
PSHY	INH	35	0S	0S

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 CPU12 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	
			HCS12X	HCS12
PULA	INH	32	uf0	uf0

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 CPU12 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	
			HCS12X	HCS12
PULB	INH	33	uf0	uf0

PULC Pull Condition Code Register from Stack PULC

Operation: $(M_{(SP)}) \Rightarrow CCR$
$(SP) + \$0001 \Rightarrow SP$

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 CPU12 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	
			HCS12X	HCS12
PULC	INH	38	uf0	uf0

PULCW Pull Condition Code Register from Stack PULCW

Operation: $(M_{(SP)}:M_{(SP+1)}) \Rightarrow CCR_H:CCR_L$; (SP) + 2 ⇒ SP

Description: The condition code register 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 CPU12 registers that were pushed onto the stack before subroutine execution.

CCR Details:

0	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
0	0	0	0	0	Δ	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	
			HCS12X	HCS12
PULCW	INH	18 38	0Uf0	NA

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 CPU12 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	
			HCS12X	HCS12
PULD	INH	3A	Uf0	Uf0

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 CPU12 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	
			HCS12X	HCS12
PULX	INH	30	Uf0	Uf0

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 CPU12 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	
			HCS12X	HCS12
PULY	INH	31	Uf0	Uf0

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) HCS12X	Access Detail(1) HCS12
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

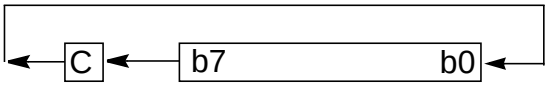
H, N, Z, and C may be altered by this instruction

Source Form	Address Mode	Object Code	Access Detail[1] HCS12X	Access Detail[1] HCS12
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(t,Tx)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

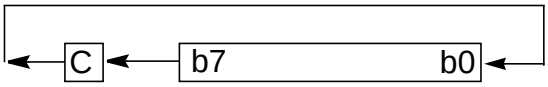
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ROL *opr16a*	EXT	`75 hh ll`	`rPw0`	`rPw0`
ROL *oprx0_xysp*	IDX	`65 xb`	`rPw`	`rPw`
ROL *oprx9,xysp*	IDX1	`65 xb ff`	`rPw0`	`rPw0`
ROL *oprx16,xysp*	IDX2	`65 xb ee ff`	`frPwP`	`frPwP`
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

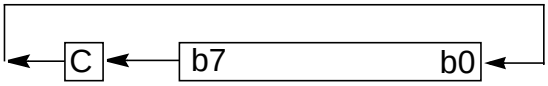
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ROLA	INH	45	0	0

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

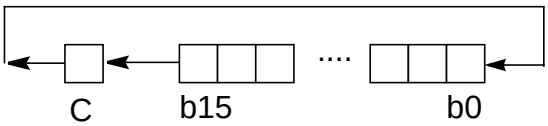
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ROLB	INH	55	0	0

ROLW Rotate Memory Left through Carry ROLW

Operation:



Description: Shifts all bits of memory location M : M + 1 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 : M + 1. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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

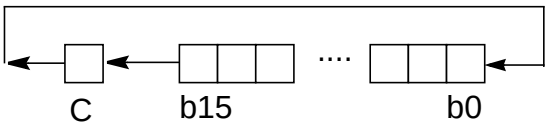
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M15
Set if the MSB of M : M + 1 was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ROLW *opr16a*	EXT	18 75 hh ll	ORPW0	NA
ROLW *oprx0_xysp*	IDX	18 65 xb	ORPW	NA
ROLW *oprx9,xysp*	IDX1	18 65 xb ff	ORPW0	NA
ROLW *oprx16,xysp*	IDX2	18 65 xb ee ff	OfRPWP	NA
ROLW [D,*xysp*]	[D,IDX]	18 65 xb	OfIfRPW	NA
ROLW [*oprx16,xysp*]	[IDX2]	18 65 xb ee ff	fOIPRPW	NA

ROLX — Rotate X Left through Carry — ROLX

Operation:



Description: Shifts all bits of index register X 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 X. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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

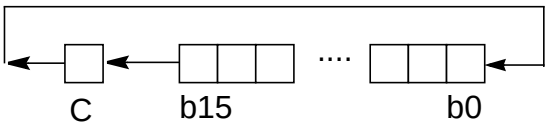
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X15
Set if the MSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ROLX	INH	18 45	00	NA

ROLY Rotate Y Left through Carry ROLY

Operation:



Description: Shifts all bits of index register Y 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 Y. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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

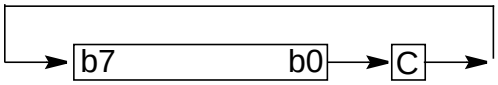
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y15
Set if the MSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
ROLY	INH	18 55	00	NA

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

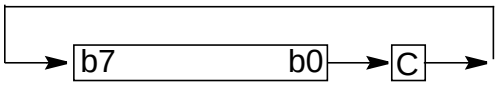
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
ROR *opr16a*	EXT	`76 hh ll`	`rPw0`	`rPw0`
ROR *oprx0_xysp*	IDX	`66 xb`	`rPw`	`rPw`
ROR *oprx9,xysp*	IDX1	`66 xb ff`	`rPw0`	`rPw0`
ROR *oprx16,xysp*	IDX2	`66 xb ee ff`	`frPwP`	`frPwP`
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

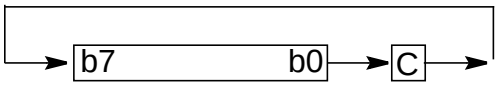
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
RORA	INH	46	0	0

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

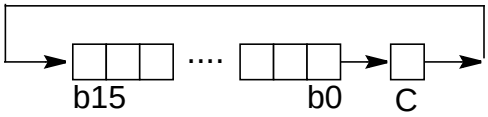
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

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

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
RORB	INH	56	0	0

RORW Rotate Memory Right through Carry RORW

Operation:



Description: Shifts all bits of memory location M : M + 1 one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of M : M + 1. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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

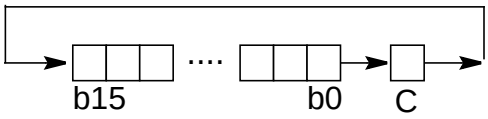
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M : M + 1 was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
RORW *opr16a*	EXT	`18 76 hh ll`	`ORPW0`	NA
RORW *oprx0_xysp*	IDX	`18 66 xb`	`ORPW`	NA
RORW *oprx9,xysp*	IDX1	`18 66 xb ff`	`ORPW0`	NA
RORW *oprx16,xysp*	IDX2	`18 66 xb ee ff`	`OfRPWP`	NA
RORW [D,*xysp*]	[D,IDX]	`18 66 xb`	`OfIfRPW`	NA
RORW [*oprx16,xysp*]	[IDX2]	`18 66 xb ee ff`	`OfIPRPW`	NA

RORX Rotate X Right through Carry RORX

Operation:



Description: Shifts all bits of index register X one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of X. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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

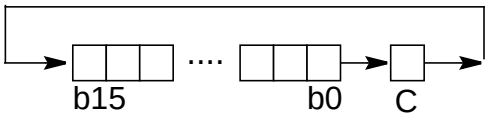
V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X0
Set if the LSB of X was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
RORX	INH	18 46	00	NA

RORY Rotate Y Right through Carry RORY

Operation:



Description: Shifts all bits of index register Y one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of Y. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

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: Y0
Set if the LSB of Y was set before the shift; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
RORY	INH	18 56	00	NA

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	
			HCS12X	HCS12
RTC	INH	0A	uUnfPPP	uUnfPPP

RTI — Return from Interrupt — RTI

Operation:

$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H : CCR_L$; (SP) + $0002 ⇒ SP
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A$; (SP) + $0002 ⇒ SP
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$; (SP) + $0004 ⇒ SP
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$; (SP) – $0002 ⇒ SP
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L$; (SP) + $0004 ⇒ 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	
			HCS12X	HCS12
RTI (with interrupt pending)	INH	0B	UUUUUPPP UUUUUfVfPPP	UUUUUPPP UUUUUfVfPPP

RTS — Return from Subroutine — RTS

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$; (SP) + $0002 ⇒ 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	
			HCS12X	HCS12
RTS	INH	3D	UfPPP	UfPPP

SBA Subtract Accumulators SBA

Operation: (A) – (B) ⇒ 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	
			HCS12X	HCS12
SBA	INH	18 16	00	00

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	
			HCS12X	HCS12
SBCA #*opr8i*	IMM	82 ii	P	P
SBCA *opr8a*	DIR	92 dd	rPf	rPf
SBCA *opr16a*	EXT	B2 hh ll	rPO	rPO
SBCA *oprx0_xysp*	IDX	A2 xb	rPf	rPf
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	fIfrPf
SBCA [*oprx16,xysp*]	[IDX2]	A2 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
SBCB #*opr8i*	IMM	C2 ii	P	P
SBCB *opr8a*	DIR	D2 dd	rPf	rPf
SBCB *opr16a*	EXT	F2 hh ll	rPO	rPO
SBCB *oprx0_xysp*	IDX	E2 xb	rPf	rPf
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	fIfrPf
SBCB [*oprx16*,*xysp*]	[IDX2]	E2 xb ee ff	fIPrPf	fIPrPf

SBED Subtract with Borrow from D SBED

Operation: (D) – (M : M + 1) – C ⇒ D

Description: Subtracts the content of memory location M : M + 1 and the value of the C status bit from the content of accumulator D. The result is placed 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

V: $B15 \bullet \overline{M15} \bullet \overline{R15} + \overline{B15} \bullet M15 \bullet R15$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{B15}$
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	
			HCS12X	HCS12
SBED #*opr16i*	IMM	18 83 jj kk	OPO	NA
SBED *opr8a*	DIR	18 93 dd	ORPf	NA
SBED *opr16a*	EXT	18 B3 hh ll	ORPO	NA
SBED *oprx0_xysp*	IDX	18 A3 xb	ORPf	NA
SBED *oprx9,xysp*	IDX1	18 A3 xb ff	ORPO	NA
SBED *oprx16,xysp*	IDX2	18 A3 xb ee ff	OfRPP	NA
SBED [D,*xysp*]	[D,IDX]	18 A3 xb	OfIfRPf	NA
SBED [*oprx16,xysp*]	[IDX2]	18 A3 xb ee ff	OfIPRPf	NA

SBEX Subtract with Borrow from X SBEX

Operation: (X) – (M : M + 1) – C ⇒ X

Description: Subtracts the content of memory location M : M + 1 and the value of the C status bit from the content of index register X. The result is placed in X. 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

V: $B15 \bullet \overline{M15} \bullet \overline{R15} + \overline{B15} \bullet M15 \bullet R15$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{B15}$
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	
			HCS12X	HCS12
SBEX #*opr16i*	IMM	18 82 jj kk	OPO	NA
SBEX *opr8a*	DIR	18 92 dd	ORPf	NA
SBEX *opr16a*	EXT	18 B2 hh ll	ORPO	NA
SBEX *oprx0_xysp*	IDX	18 A2 xb	ORPf	NA
SBEX *oprx9,xysp*	IDX1	18 A2 xb ff	ORPO	NA
SBEX *oprx16,xysp*	IDX2	18 A2 xb ee ff	OfRPP	NA
SBEX [D,*xysp*]	[D,IDX]	18 A2 xb	OfIfRPf	NA
SBEX [*oprx16,xysp*]	[IDX2]	18 A2 xb ee ff	OfIPRPf	NA

SBEY Subtract with Borrow from Y SBEY

Operation: (Y) – (M : M + 1) – C ⇒ Y

Description: Subtracts the content of memory location M : M + 1 and the value of the C status bit from the content of index register Y. The result is placed in Y. 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: The zero bit is set if the result is $0000 AND the zero bit was set before the instruction

V: $B15 \bullet \overline{M15} \bullet \overline{R15} + \overline{B15} \bullet M15 \bullet R15$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{B15}$
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	
			HCS12X	HCS12
SBEY #*opr16i*	IMM	18 C2 jj kk	OPO	NA
SBEY *opr8a*	DIR	18 D2 dd	ORPf	NA
SBEY *opr16a*	EXT	18 F2 hh ll	ORPO	NA
SBEY *oprx0_xysp*	IDX	18 E2 xb	ORPf	NA
SBEY *oprx9,xysp*	IDX1	18 E2 xb ff	ORPO	NA
SBEY *oprx16,xysp*	IDX2	18 E2 xb ee ff	OfRPP	NA
SBEY [D,*xysp*]	[D,IDX]	18 E2 xb	OfIfRPf	NA
SBEY [*oprx16,xysp*]	[IDX2]	18 E2 xb ee ff	OfIPRPf	NA

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	
			HCS12X	HCS12
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 CPU12 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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
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) ⇒ r3
If r1 bit 7 = 1, then $FF : (r1) ⇒ r3

If r2 bit 15 = 0, then $0000 ⇒ r3
If r2 bit 15 = 1, then $FFFF ⇒ r3

Description:

This instruction is an alternate mnemonic for the TFR r1,r3 or TFR r2,r3 instructions, where r1 is an 8-bit register, and r2 and r3 are16-bit registers. The result in r3 is the 16-bit sign extended representation of the original two's complement number in r1 or r2. The content of r1 or r2 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 HCS12X	Access Detail HCS12
SEX *abc,dxys*	INH	B7 eb	P	P
SEX *d,xy*	INH	B7 eb	P	NA

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. Values are in hexadecimal.

	MS ⇒	0	1	2	4
⇓ LS		A	B	CCR	D
3	TMP2	sex:A ⇒ TMP2 SEX A,TMP2	sex:B ⇒ TMP2 SEX B,TMP2	sex:CCR_L ⇒ TMP2 *SEX CCR,TMP2* *SEX CCRL,TMP2*	NA
4	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR_L ⇒ D SEX CCR_L,D *SEX CCRL,D*	NA
5	X	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR_L ⇒ X SEX CCR,X *SEX CCRL,X*	NA
6	Y	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR_L ⇒ Y SEX CCR,Y *SEX CCRL,Y*	NA
7	SP	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR_L ⇒ SP SEX CCR,SP *SEX CCRL,SP*	NA
C	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	NA	NA
D	X	NA	NA	NA	sex:D ⇒ X SEX D,X
E	Y	NA	NA	NA	sex:D ⇒ Y SEX D,Y

Note: Encodings in the shaded area (LS = C–E) are only available on the S12X.

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	
			HCS12X	HCS12
STAA *opr8a*	DIR	5A dd	Pw	Pw
STAA *opr16a*	EXT	7A hh ll	PwO	PwO
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	PIfw
STAA [*oprx16,xysp*]	[IDX2]	6A xb ee ff	PIPw	PIPw

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	
			HCS12X	HCS12
STAB *opr8a*	DIR	5B dd	Pw	Pw
STAB *opr16a*	EXT	7B hh ll	PwO	PwO
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	PIfw
STAB [*oprx16,xysp*]	[IDX2]	6B xb ee ff	PIPw	PIPw

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	
			HCS12X	HCS12
STD *opr8a*	DIR	5C dd	PW	PW
STD *opr16a*	EXT	7C hh ll	PWO	PWO
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	PIfW
STD [*oprx16,xysp*]	[IDX2]	6C xb ee ff	PIPW	PIPW

STOP — Stop Processing — STOP

Operation:

(SP) – $0002 ⇒ SP; $RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; B : A ⇒ $(M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
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 CPU12 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 CPU12 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 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 HCS12X	Access Detail HCS12
STOP (entering STOP)	INH	18 3E	00SSSSSf	00SSSSSf
(exiting STOP)			fVfPPP	fVfPPP
(continue)			ff	ff
(if STOP disabled)			00	00

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	
			HCS12X	HCS12
STS *opr8a*	DIR	5F dd	PW	PW
STS *opr16a*	EXT	7F hh ll	PWO	PWO
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	PIfW
STS [*oprx16,xysp*]	[IDX2]	6F xb ee ff	PIPW	PIPW

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	
			HCS12X	HCS12
STX *opr8a*	DIR	5E dd	PW	PW
STX *opr16a*	EXT	7E hh ll	PWO	PWO
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	PIfW
STX [*oprx16,xysp*]	[IDX2]	6E xb ee ff	PIPW	PIPW

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	
			HCS12X	HCS12
STY *opr8a*	DIR	5D dd	PW	PW
STY *opr16a*	EXT	7D hh ll	PWO	PWO
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	PIfW
STY [*oprx16,xysp*]	[IDX2]	6D xb ee ff	PIPW	PIPW

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	
			HCS12X	HCS12
SUBA #*opr8i*	IMM	80 ii	P	P
SUBA *opr8a*	DIR	90 dd	rPf	rPf
SUBA *opr16a*	EXT	B0 hh ll	rPO	rPO
SUBA *oprx0_xysp*	IDX	A0 xb	rPf	rPf
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	fIfrPf
SUBA [*oprx16,xysp*]	[IDX2]	A0 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
SUBB #*opr8i*	IMM	C0 ii	P	P
SUBB *opr8a*	DIR	D0 dd	rPf	rPf
SUBB *opr16a*	EXT	F0 hh ll	rPO	rPO
SUBB *oprx0_xysp*	IDX	E0 xb	rPf	rPf
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	fIfrPf
SUBB [*oprx16,xysp*]	[IDX2]	E0 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
SUBD #*opr16i*	IMM	83 jj kk	PO	PO
SUBD *opr8a*	DIR	93 dd	RPf	RPf
SUBD *opr16a*	EXT	B3 hh ll	RPO	RPO
SUBD *oprx0_xysp*	IDX	A3 xb	RPf	RPf
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	fIfRPf
SUBD [*oprx16,xysp*]	[IDX2]	A3 xb ee ff	fIPRPf	fIPRPf

SUBX Subtract Memory from X SUBX

Operation: (X) – (M : M + 1) ⇒ X

Description: Subtracts the content of memory location M : M + 1 from the content of index register X and places the result in X. 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 index register; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
SUBX #*opr16i*	IMM	18 80 jj kk	OPO	NA
SUBX *opr8a*	DIR	18 90 dd	ORPf	NA
SUBX *opr16a*	EXT	18 B0 hh ll	ORPO	NA
SUBX *oprx0_xysp*	IDX	18 A0 xb	ORPf	NA
SUBX *oprx9,xysp*	IDX1	18 A0 xb ff	ORPO	NA
SUBX *oprx16,xysp*	IDX2	18 A0 xb ee ff	OfRPP	NA
SUBX [D,*xysp*]	[D,IDX]	18 A0 xb	OfIfRPf	NA
SUBX [*oprx16,xysp*]	[IDX2]	18 A0 xb ee ff	OfIPRPf	NA

SUBY — Subtract Memory from Y — SUBY

Operation: (Y) – (M : M + 1) ⇒ Y

Description: Subtracts the content of memory location M : M + 1 from the content of index register Y and places the result in Y. 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 index register; cleared otherwise

Source Form	Address Mode	Object Code	Access Detail	
			HCS12X	HCS12
SUBY #*opr16i*	IMM	18 C0 jj kk	OPO	NA
SUBY *opr8a*	DIR	18 D0 dd	ORPf	NA
SUBY *opr16a*	EXT	18 F0 hh ll	ORPO	NA
SUBY *oprx0_xysp*	IDX	18 E0 xb	ORPf	NA
SUBY *oprx9,xysp*	IDX1	18 E0 xb ff	ORPO	NA
SUBY *oprx16,xysp*	IDX2	18 E0 xb ee ff	OfRPP	NA
SUBY [D,*xysp*]	[D,IDX]	18 E0 xb	OfIfRPf	NA
SUBY [*oprx16,xysp*]	[IDX2]	18 E0 xb ee ff	OfIPRPf	NA

SWI — Software Interrupt — SWI

Operation:

(SP) – $0002 ⇒ SP; $RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; B : A⇒ $(M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
1 ⇒ I
(SWI Vector) ⇒ 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	
			HCS12X	HCS12
SWI	INH	3F	VSPSSPSSP(1)	VSPSSPSsP(1)

1. The CPU12 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	
			HCS12X	HCS12
TAB	INH	18 0E	00	00

TAP

Transfer from Accumulator A to Condition Code Register

TAP

Operation: (A) $\Rightarrow$ CCR_L

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, SEI.

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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
TBA	INH	18 0F	00	00

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	
			HCS12X	HCS12
TBEQ *abdxys,rel9*	REL	`04 lb rr`	PPP/PP0	PPP/PP0

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) × ((M+1) – (M))] ⇒ 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

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	
			HCS12X	HCS12
TBL *oprx0_xysp*	IDX	18 3D xb	ORfffP	ORfffP

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 HCS12X	Access Detail HCS12
TBNE *abdxys,rel9*	REL	`04 lb rr`	PPP/PP0	PPP/PP0

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 Motorola 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	
			HCS12X	HCS12
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. Values are in hexadecimal.

TFR

Transfer Register Content to Another Register (Continued)

TFR

	MS⇒	0	1	2	3	4	5	6	7
⇓LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR_L ⇒ A TFR CCR,A TFR CCRL,A	$TMP3_L$ ⇒ A TFR TMP3,A TFR TMP3L,A	B ⇒ A TFR D,A	X_L ⇒ A TFR X, A TFR XL,A	Y_L ⇒ A TFR Y,A TFR YL,A	SP_L ⇒ A TFR SP,A TFR SPL,A
1	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR_L ⇒ B TFR CCR,B TFR CCRL,B	$TMP3_L$ ⇒ B TFR TMP3,B TFR TMP3L,B	B ⇒ B TFR D,B	X_L ⇒ B TFR X, B TFR XL,B	Y_L ⇒ B TFR Y,B TFR YL,B	SP_L ⇒ B TFR SP,B TFR SPL,B
2	CCR	A ⇒ CCR TFR A,CCR TFR A,CCRL	B ⇒ CCR TFR B,CCR TFR B,CCRL	CCR_L ⇒ CCR_L TFR CCR,CCR TFR CCRL,CCRL	$TMP3_L$ ⇒ CCR TFR TMP3,CCR TFR TMP3L,CCRL	B ⇒ CCR TFR D,CCR TFR D,CCRL	X_L ⇒ CCR TFR X,CCR TFR XL,CCRL	Y_L ⇒ CCR TFR Y,CCR TFR YL,CCRL	SP_L ⇒ CCR TFR SP,CCR TFR SPL,CCRL
3	TMP2	sex:A ⇒ TMP2 SEX A,TMP2	sex:B ⇒ TMP2 SEX B,TMP2	sex:CCR_L ⇒ TMP2 SEX CCR,TMP2 SEX CCRL,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP2 TFR D,TMP2	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
4	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR_L ⇒ D SEX CCR_L,D SEX CCRL,D	TMP3 ⇒ D TFR TMP3,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
5	X	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR_L ⇒ X SEX CCR,X SEX CCRL,X	TMP3 ⇒ X TFR TMP3,X	D ⇒ X TFR D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
6	Y	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR_L ⇒ Y SEX CCR,Y SEX CCRL,Y	TMP3 ⇒ Y TFR TMP3,Y	D ⇒ Y TFR D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
7	SP	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR_L ⇒ SP SEX CCR,SP SEX CCRL,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP
8	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR_H ⇒ A TFR CCRH,A	$TMP3_H$ ⇒ A TFR TMP3H,A	B ⇒ A TFR D,A	X_H ⇒ A TFR XH, A	Y_H ⇒ A TFR YH,A	SP_H ⇒ A TFR SPH,A
9	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR_L ⇒ B TFR CCRL,B	$TMP3_L$ ⇒ B TFR TMP3L,B	B ⇒ B TFR D,B	X_L ⇒ B TFR XL, B	Y_L ⇒ B TFR YL,B	SP_L ⇒ B TFR SPL,B
A	CCR	A ⇒ CCR_H TFR A,CCRH	B ⇒ CCR_L TFR B,CCRL	CCRW ⇒ CCRW TFR CCRW,CCRW	TMP3 ⇒ $CCR_{H:L}$ TFR TMP3,CCRW	D ⇒ $CCR_{H:L}$ TFR D,CCRW	X ⇒ $CCR_{H:L}$ TFR X,CCRW	Y ⇒ $CCR_{H:L}$ TFR Y,CCRW	SP ⇒ $CCR_{H:L}$ TFR SP,CCRW
B	TMPx	A ⇒ $TMP2_H$ TFR A,TMP2H	B ⇒ $TMP2_L$ TFR B,TMP2L	$CCR_{H:L}$ ⇒ TMP2 TFR CCRW,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP1 TFR D,TMP1	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
C	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	$CCR_{H:L}$ ⇒ D TFR CCRW,D	TMP1 ⇒ D TFR TMP1,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
D	X	A ⇒ X_H TFR A,XH	B ⇒ X_L TFR B,XL	$CCR_{H:L}$ ⇒ X TFR CCRW,X	TMP3 ⇒ X TFR TMP3,X	sex:D ⇒ X SEX D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
E	Y	A ⇒ Y_H TFR A,YH	B ⇒ Y_L TFR B,YL	$CCR_{H:L}$ ⇒ Y TFR CCRW,Y	TMP3 ⇒ Y TFR TMP3,Y	sex:D ⇒ Y SEX D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
F	SP	A ⇒ SP_H TFR A,SPH	B ⇒ SP_L TFR B,SPL	$CCR_{H:L}$ ⇒ SP TFR CCRW,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the S12X.

TPA

Transfer from Condition Code Register to Accumulator A

TPA

Operation: $(CCR_L) \Rightarrow A$

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	
			HCS12X	HCS12
TPA *translates to...* TFR CCR,A	INH	B7 20	P	P

TRAP Unimplemented Opcode Trap TRAP

Operation:

(SP) – $0002 ⇒ SP; $RTN_H : RTN_L ⇒ (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $Y_H : Y_L ⇒ (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $X_H : X_L ⇒ (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; B : A ⇒ $(M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $CCR_H : CCR_L ⇒ (M_{(SP)} : M_{(SP+1)})$
1 ⇒ I
(Trap Vector) ⇒ 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 CPU12 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 HCS12X	Access Detail HCS12
TRAP *trapnum*	INH	$18 tn(1)	0VSPSSPSSP	0VSPSSPSSP

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	
			HCS12X	HCS12
TST *opr16a*	EXT	F7 hh ll	rPO	rPO
TST *oprx0_xysp*	IDX	E7 xb	rPf	rPf
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	fIfrPf
TST [*oprx16,xysp*]	[IDX2]	E7 xb ee ff	fIPrPf	fIPrPf

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	
			HCS12X	HCS12
TSTA	INH	97	0	0

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	
			HCS12X	HCS12
TSTB	INH	D7	0	0

TSTW Test Memory for Zero or Minus TSTW

Operation: (M : M + 1) – 0

Description: Subtracts $0000 from the content of memory location M : M + 1 and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying M : M + 1.

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 $0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
TSTW *opr16a*	EXT	18 F7 hh ll	ORPO	NA
TSTW *oprx0_xysp*	IDX	18 E7 xb	ORPf	NA
TSTW *oprx9,xysp*	IDX1	18 E7 xb ff	ORPO	NA
TSTW *oprx16,xysp*	IDX2	18 E7 xb ee ff	OfRPP	NA
TSTW [D,*xysp*]	[D,IDX]	18 E7 xb	OfIfRPf	NA
TSTW [*oprx16,xysp*]	[IDX2]	18 E7 xb ee ff	OfIPRPf	NA

TSTX Test X for Zero or Minus TSTX

Operation: (X) – 0

Description: Subtracts $0000 from the content of index register X and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying X.

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 $0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
TSTX	INH	18 97	00	NA

TSTY Test Y for Zero or Minus TSTY

Operation: (Y) – 0

Description: Subtracts $0000 from the content of index register Y and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying Y.

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 $0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Source Form	Address Mode	Object Code	Access Detail HCS12X	Access Detail HCS12
TSTY	INH	18 D7	00	NA

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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
TYS *translates to...* TFR Y,SP	INH	B7 67	P	P

WAI — Wait for Interrupt — WAI

Operation:

(SP) – $0002 ⇒ SP; $RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; B : A ⇒ $(M_{(SP)} : M_{(SP+1)})$
(SP) – $0002 ⇒ SP; $CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
Stop CPU12 Clocks

Description:

Puts the CPU12 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 CPU12 then enters a wait state for an integer number of bus clock cycles. During the wait state, CPU12 clocks are stopped, but other MCU clocks can continue to run. The CPU12 leaves the wait state when it senses an interrupt that has not been masked.

Upon leaving the wait state, the CPU12 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	
			HCS12X	HCS12
WAI (before interrupt)	INH	3E	OSSSSSf	OSSSSSf
WAI (when interrupt comes)			fVfPPP	fVfPPP

Although the WAI instruction itself does not alter the condition codes, the interrupt that causes the CPU12 to resume processing also causes the I mask bit (and the X mask bit, if the interrupt was $\overline{XIRQ}$) 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 $SOW_{[15:0]}$, $SOP_{[15:0]}$, $00:$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] HCS12X	Access Detail[1] HCS12
WAV	Special	18 3C	Of(frr,ffff)O	Of(frr,ffff)O
			(replace comma if interrupted)	
			SSS + UUUrr	SSS + 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	
			HCS12X	HCS12
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	
			HCS12X	HCS12
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
  - System reset :
    - $\overline{\text{RESET}}$ pin
    - Power On reset
    - Low Voltage reset
    - Illegal address reset (S12X only)
  - Clock monitor reset
  - COP watchdog reset
- An unimplemented opcode trap
- A software interrupt instruction (SWI) or BDM vector request
- Non-maskable (X-bit) interrupts
- Non-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(2)	Source
$FFFE	System reset
$FFFC	Clock Monitor reset
$FFFA	COP Watchdog reset
(Vector Base(3) + $F8)	Unimplemented opcode trap
(Vector Base +$F6)	Software interrupt instruction (SWI) or BDM vector request
(Vector Base +$F4)	$\overline{\text{XIRQ}}$ signal
(Vector Base +$F2)	$\overline{\text{IRQ}}$ signal
(Vector Base +$F0...$10)	Device-specific interrupt sources (HCS12X)
$FF80–$FFF1	Device-specific interrupt sources (HCS12)

1. See SoC Guide and Interrupt Block Guide for further details
2. 16 bits Vector Address based
3. The Vector Base is a 16 bit address which is accumulated from :
   a) A fixed value $FF00 (HCS12)
   b) Contents of Interrupt Vector Base Register (upper byte) and $00 (lower byte) (HCS12X).

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 Motorola use. Refer to SoC Guide for more information. The HCS12X devices typically allow the user to configure the start adress of the vector table. Refer to the Interrupt Module Block Guide for further 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. System Reset 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)

System 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 CPU12 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. Typically, 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

HCS12 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 CPU12 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.

There are four possible sources of reset are:

- System reset :
  - $\overline{\text{RESET}}$ pin
  - Power On reset (POR)
  - Low Voltage reset
  - Illegal address reset (S12X only)
- Clock monitor reset
- COP watchdog 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. Typically, 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 CPU12 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 CPU12 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 CPU12 registers are stacked as shown in **Table 7-2** for HCS12 and as shown in **Table 7-3** for the HCS12X.

Table 7-2. HCS12 Stacking Order on Entry to Interrupts

Memory Location	CPU12 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

Table 7-3. HCS12X Stacking Order on Entry to Interrupts

Memory Location	CPU12 Registers
SP + 8	RTN_H : RTN_L
SP + 6	Y_H : Y_L
SP + 4	X_H : X_L
SP + 2	B : A
SP	CCR_H:CCR_L

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, 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 for HCS12 and SP = SP -10 for HCS12X). 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 HCS12 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 CPU12 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 HCS12X has opcodes in all 256 positions in the page 1 opcode map, and 227 of the 256 positions on page 2 of the opcode map are used. If the CPU12 attempts to execute one of the 29 unused opcodes on page 2, an unimplemented opcode trap occurs. The 29 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. 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 CPU12 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) if not placed else where using the Interrupt Vector Base Register (HCS12X only). The CPU12 cannot use the fetched vector until the third cycle of the exception processing sequence.

During the vector fetch cycle, the CPU12 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 CPU12 does not provide this address).

After the vector fetch, the CPU12 selects one of the three alternate execution paths, depending upon the cause of the exception.

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 CPU12 starts executing the instruction at the head of the instruction queue.

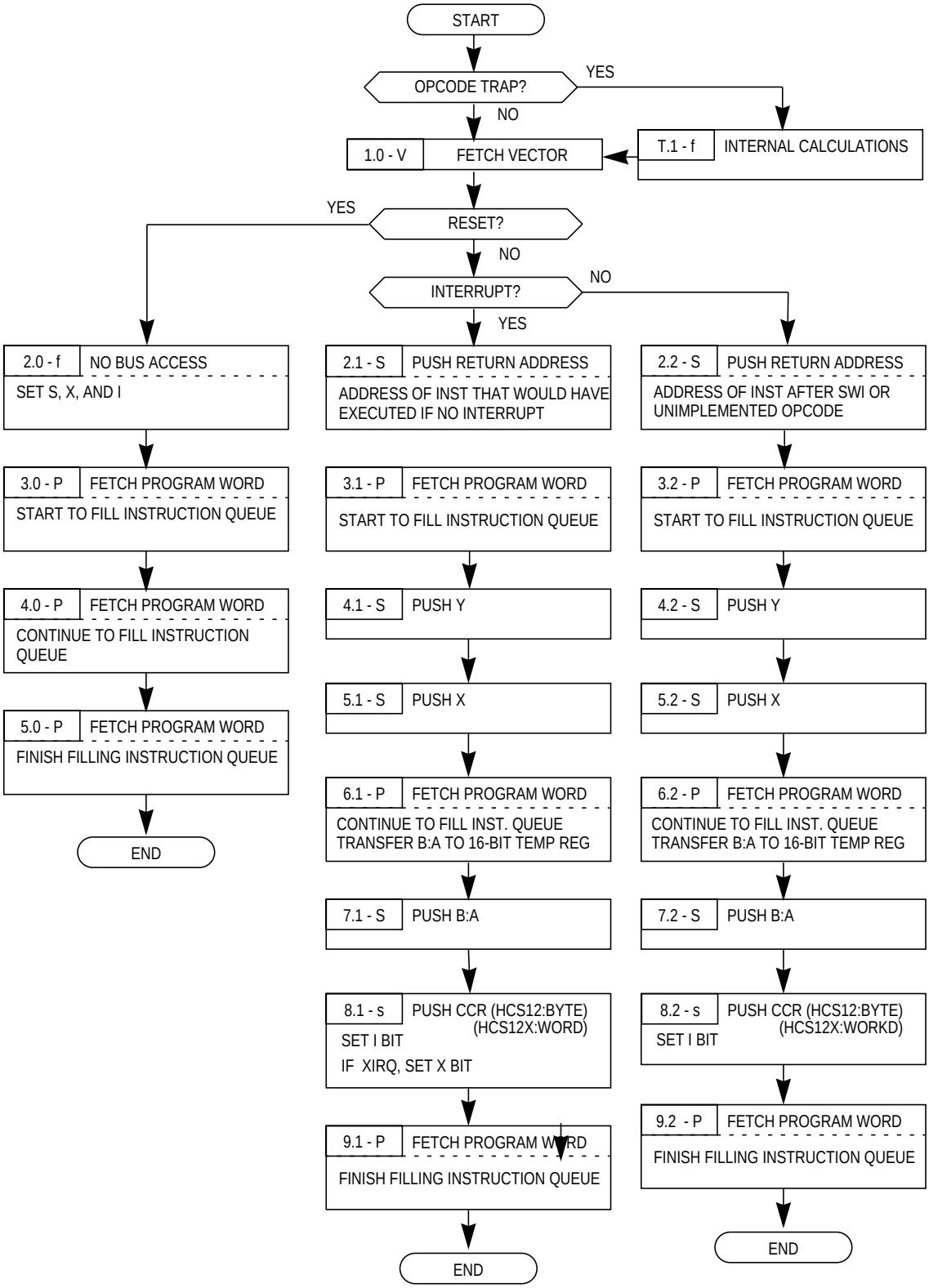


Figure 7-1. Exception Processing Flow Diagram

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.
- 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 (HCS12) respecitvely a 16-bit CCRW (HCS12X) 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 CPU12 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:

- Instruction queue operation and reconstruction
- Instruction tagging

8.2 External Reconstruction of the Queue

The CPU12 uses an instruction queue to buffer program information and increase instruction throughput.. The HCS12 queue consists of three 16-bit stages. Program information is always fetched in aligned 16-bit words. At least three bytes of program information are available to the CPU12 when instruction execution begins.

Because of the queue, program information is fetched a few cycles before it is used by the CPU12. Internally, the microcontroller unit (MCU) only needs to buffer the fetched data. But, in order to monitor cycle-by-cycle CPU12 activity externally, it is necessary to capture data and address to discern what is happening in the instruction queue.

External pins, (IPIPE1 and IPIPE0 for HCS12) (IQSTAT[3:0] for HCS12X), provide 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 CPU12 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 **8.5 Instruction Tagging (HCS12)**.

8.3 Instruction Queue Status Signals

The (IPIPE1 and IPIPE0 for HCS12) (IQSTAT[3:0] for HCS12X) signals carry information about data movement and instruction execution during normal CPU12 operation.

To reconstruct the queue, the information carried by the status signals must be captured externally. The definition of the this signals is different from HCS12 and HCS12X, refer to **8.3.1 HCS12 Timing Detail** and **8.3.2 HCS12X Timing Detail**

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.

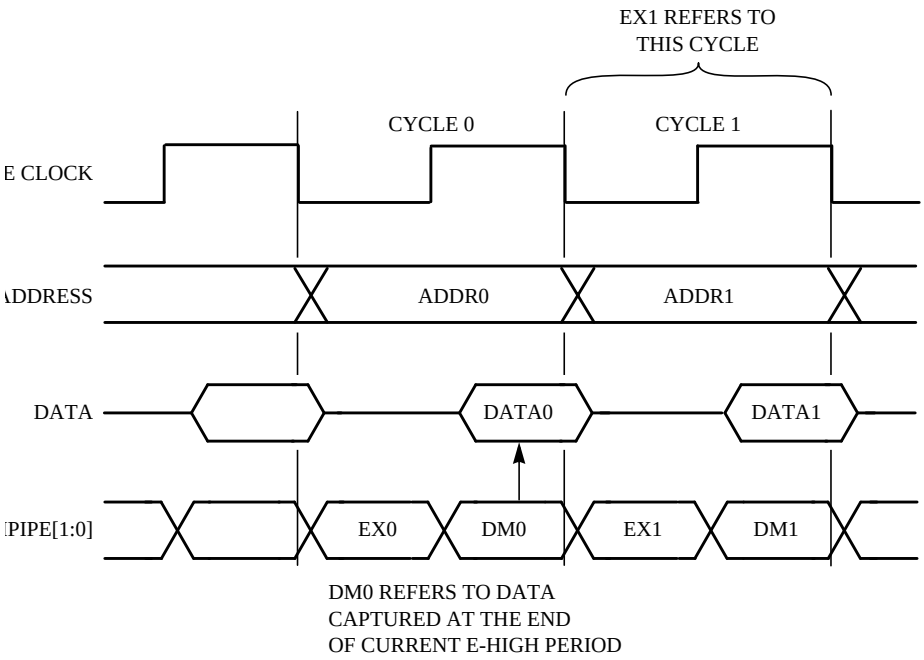


Figure 8-1. Queue Status Signal Timing (HCS12)

Table 8-1. IPIPE1 and IPIPE0 Decoding HCS12

	Mnemonic	Meaning
Data Movement	**Capture at E Fall in HCS12**	
0:0	—	No movement
0:1	—	Unused?
1:0	ALD	Advance queue and load from bus
1:1		
Execution Start	**Capture at E Rise in HCS12**	
0:0	—	No start
0:1	INT	Start interrupt sequence
1:0	SEV	Start even instruction
1:1	SOD	Start odd instruction

8.3.2 HCS12X Timing Detail

In the HCS12X, data-movement information and execution-start information are demultiplexed and available on the signal IQSTAT[3:0] when the next E clock is low, as shown in **Figure 8-2**. Data-movement information refers to data on the previous bus cycle that starts with that E-low time. Execution-start information refers to the previous bus cycle that starts with the E-high time. **Table 8-2** summarizes the information on the IQSTAT[3:0].

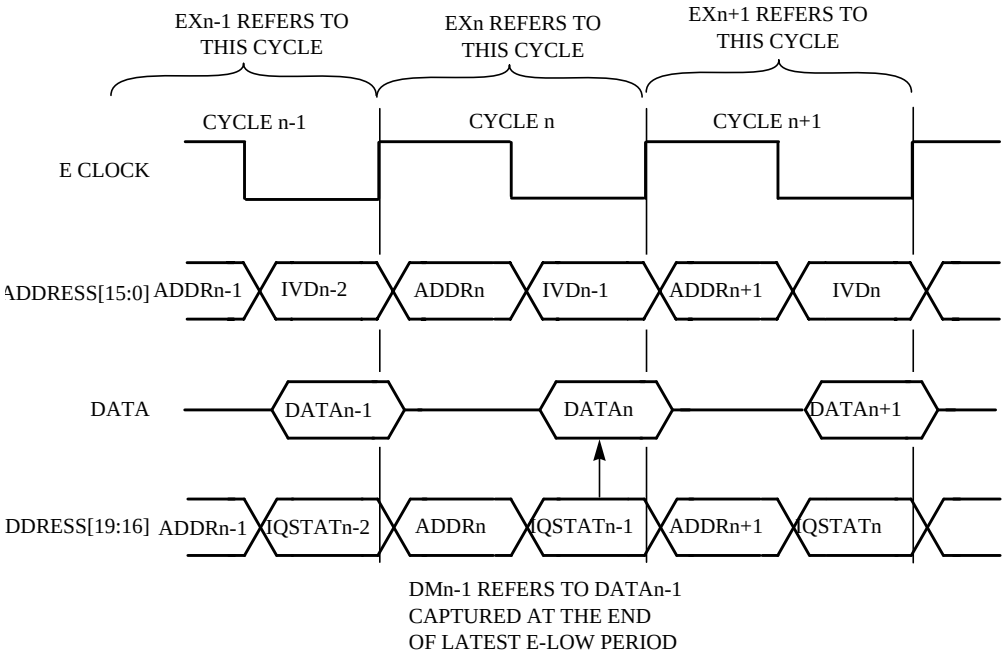


Figure 8-2. Queue Status Signal Timing (HCS12X)

Table 8-2. IQSTAT[3:0] on HCS12X

	Mnemonic	Meaning
Execution Start	**Capture at E Fall in HCS12X**	
0001	INT	Start interrupt sequence
0010	SOD	Start instruction at odd address
0100	SEV	Start instruction at even address
Data Movement	**Capture at E Fall in HCS12X**	
1000	ALD	Advance the instruction queue and load first stage
Data Movement & Execution Start	**Capture at E Fall in HCS12X**	
0000	Null	No movement, No start
1100	ALD&SEV	Advance the instruction queue and load first stage then Start instruction at even address
1010	ALD&SOD	Advance the instruction queue and load first stage then Start instruction at odd address
1001	ALD&INT	Advance the instruction queue and load first stage then Start interrupt sequence
Others	Not Implemented	---

8.3.3 Null

The (00 for HCS12) (0000 for HCS12X) data movement state indicates that there was no data movement in the instruction queue; the (00 for HCS12) (0000 for HCS12X)) execution start state indicates continuation of an instruction or interrupt sequence (no new instruction or interrupt start).

8.3.4 ALD — Advance and Load from Data Bus

The instruction queue is advanced by one word and stage one is refilled with a word of program information from the data bus. The CPU12 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.

8.3.5 INT — Interrupt Sequence Start

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.

8.3.6 SEV — Start Instruction on Even Address

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.

8.3.7 SOD — Start Instruction on Odd Address

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 CPU12 clocks long enough to do two 8-bit accesses rather than a single 16-bit access, so the CPU12 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 HCS12X)

This section describes how to use IQSTAT[3:0] signals and queue reconstruction registers to reconstruct the queue.

Note : More details will be included by the next release

8.5 Instruction Tagging (HCS12)

The instruction queue and cycle-by-cycle CPU12 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 CPU12 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 $\overline{\text{TAGLO}}$ signal shares a pin with the $\overline{\text{LSTRB}}$ signal, and the $\overline{\text{TAGHI}}$ signal shares the BKGD pin. Tagging information is latched on the falling edge of ECLK, as shown in **Figure 8-3**.

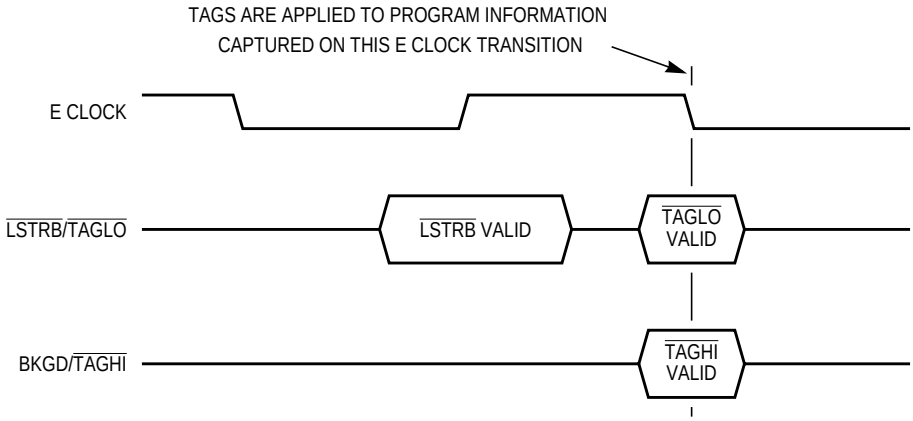


Figure 8-3. Tag Input Timing (HCS12)

Table 8-3 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 8-3. Tag Pin Function (HCS12)

$\overline{\text{TAGHI}}$	$\overline{\text{TAGLO}}$	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 CPU12 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 CPU12 enters active background debug mode rather than executing the instruction.

8.6 Instruction Tagging (HCS12X)

The instruction queue and cycle-by-cycle CPU12 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 CPU12 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.

The $\overline{\text{TAGLO}}$ signal shares a pin with the $\overline{\text{RE}}$ and MODA signals, and the $\overline{\text{TAGHI}}$ signal shares the MODB pin. Tagging information is latched on the rising edge of ECLK, as shown in **Figure 8-4**.

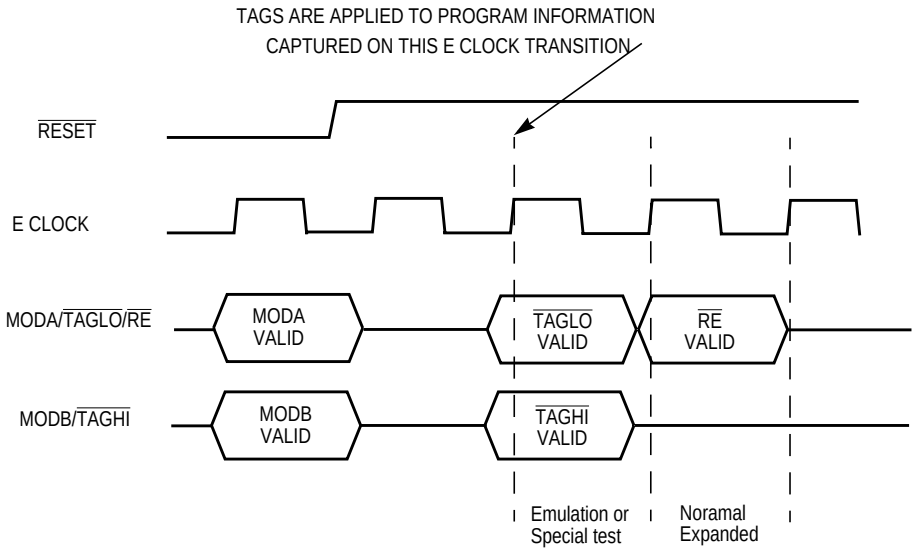


Figure 8-4. Tag Input Timing (HCS12X)

Table 8-4 shows the functions of the two independent tagging pins. The presence of logic level 0 on either pin at the rise of ECLK tags (marks) the associated byte of program information as it is read into the instruction queue. Tagging is allowed only in emulation modes. Tagging is disabled when BDM becomes active.

Table 8-4. Tag Pin Function (HCS12X)

$\overline{\text{TAGHI}}$	$\overline{\text{TAGLO}}$	Tag
1	1	No tag
1	0	Low byte

Table 8-4. Tag Pin Function (HCS12X)

$\overline{\text{TAGHI}}$	$\overline{\text{TAGLO}}$	Tag
0	1	High byte
0	0	Both bytes

IOn the HCS12X internal breakpoints can also be generated by the S12XDBG module. Breakpoints generated by the TAGLO or TAGHI have a higher priority than the internally generated breakpoints.

The tag follows program information as it advances through the queue. When a tagged instruction reaches the head of the queue, a taghit occurs generating a hardware beakpoint to BDM or SWI.

Instruction Queue

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. By incorporating fuzzy logic support into a high-volume, general-purpose microcontroller product family, Motorola has made fuzzy logic available for a huge base of applications.

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.

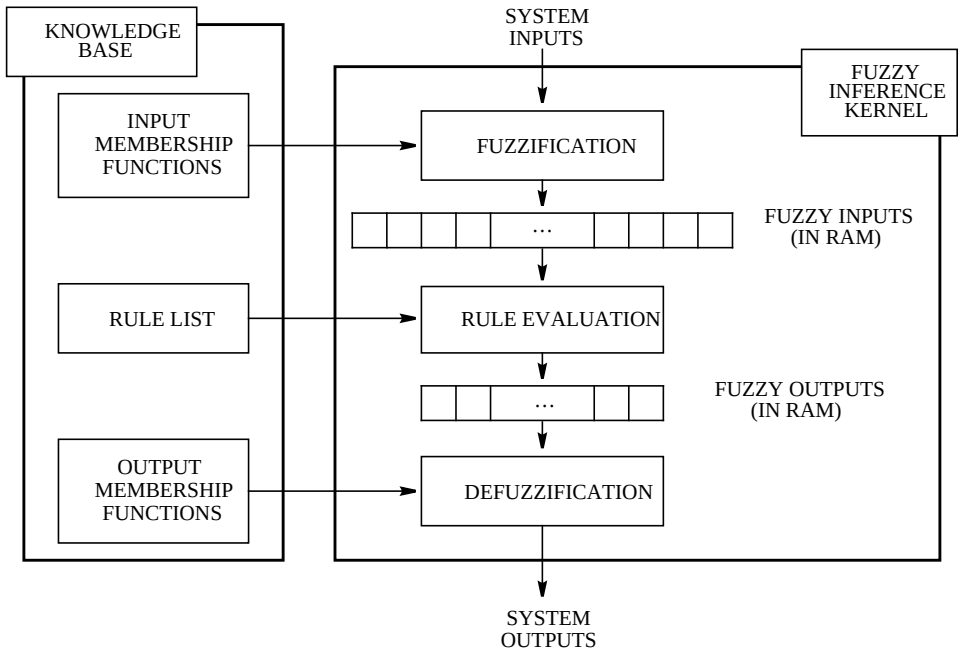


Figure 9-1. Block Diagram of a Fuzzy Logic System

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).

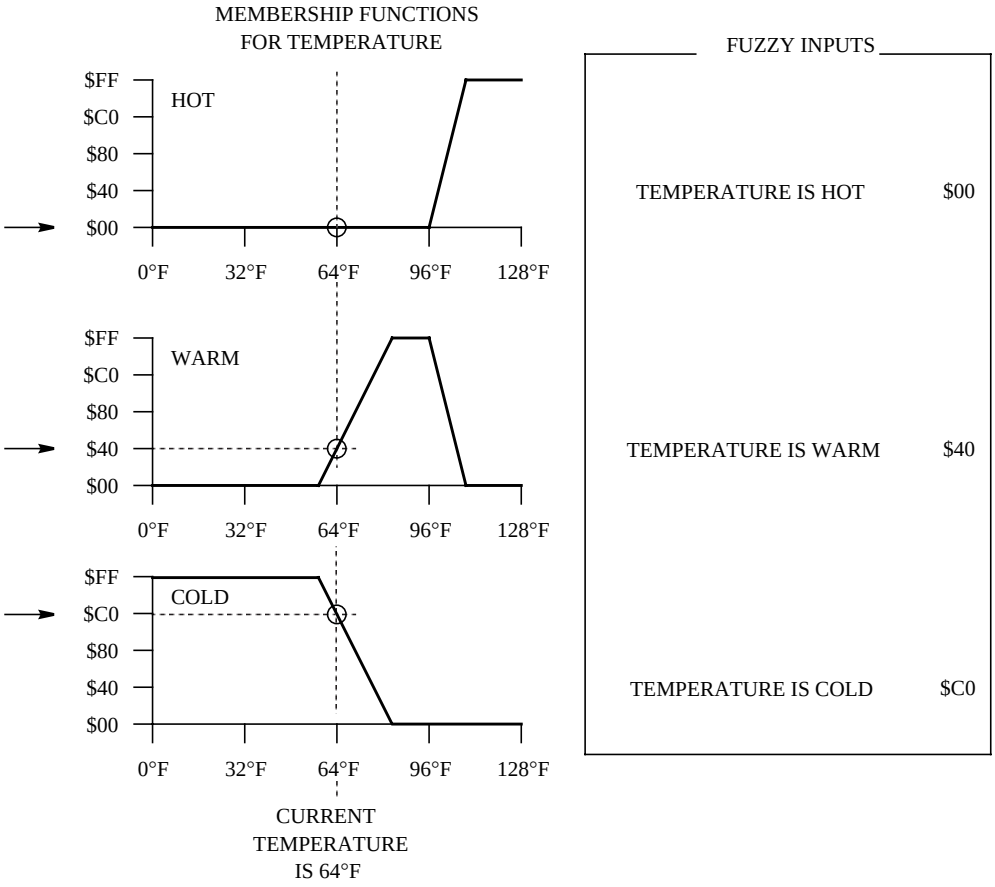


Figure 9-2. Fuzzification Using Membership Functions

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 Motorola 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.

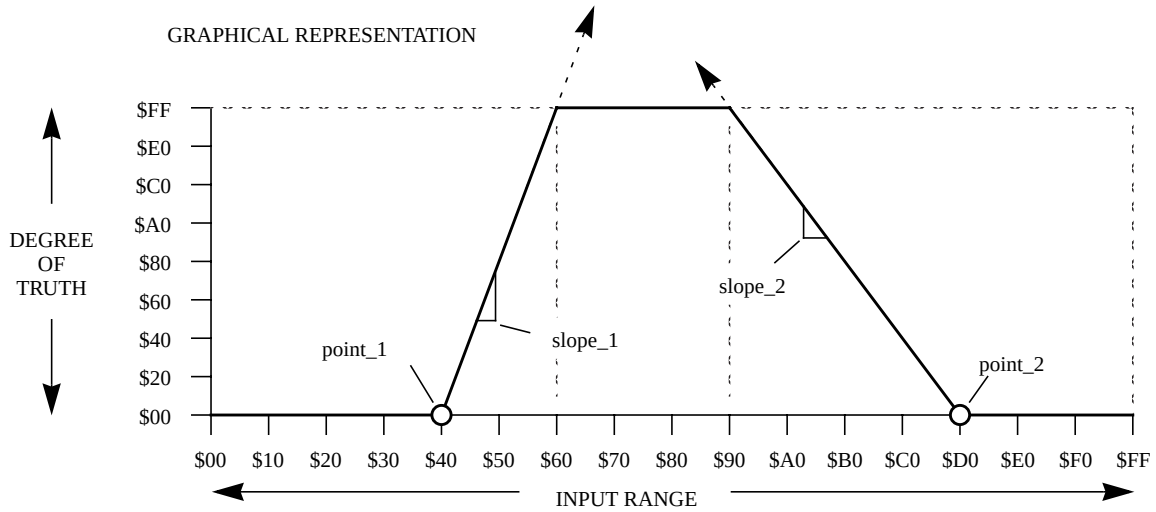
This example inference program shows how easy it is to incorporate fuzzy logic into general applications using the CPU12. Code space and execution time are no longer serious factors in the decision to use fuzzy logic. The next section begins a much more detailed look at the fuzzy logic instructions of 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 CPU12 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 CPU12 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 CPU12 processes. It is common for several internal functions to take place during a single CPU12 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.

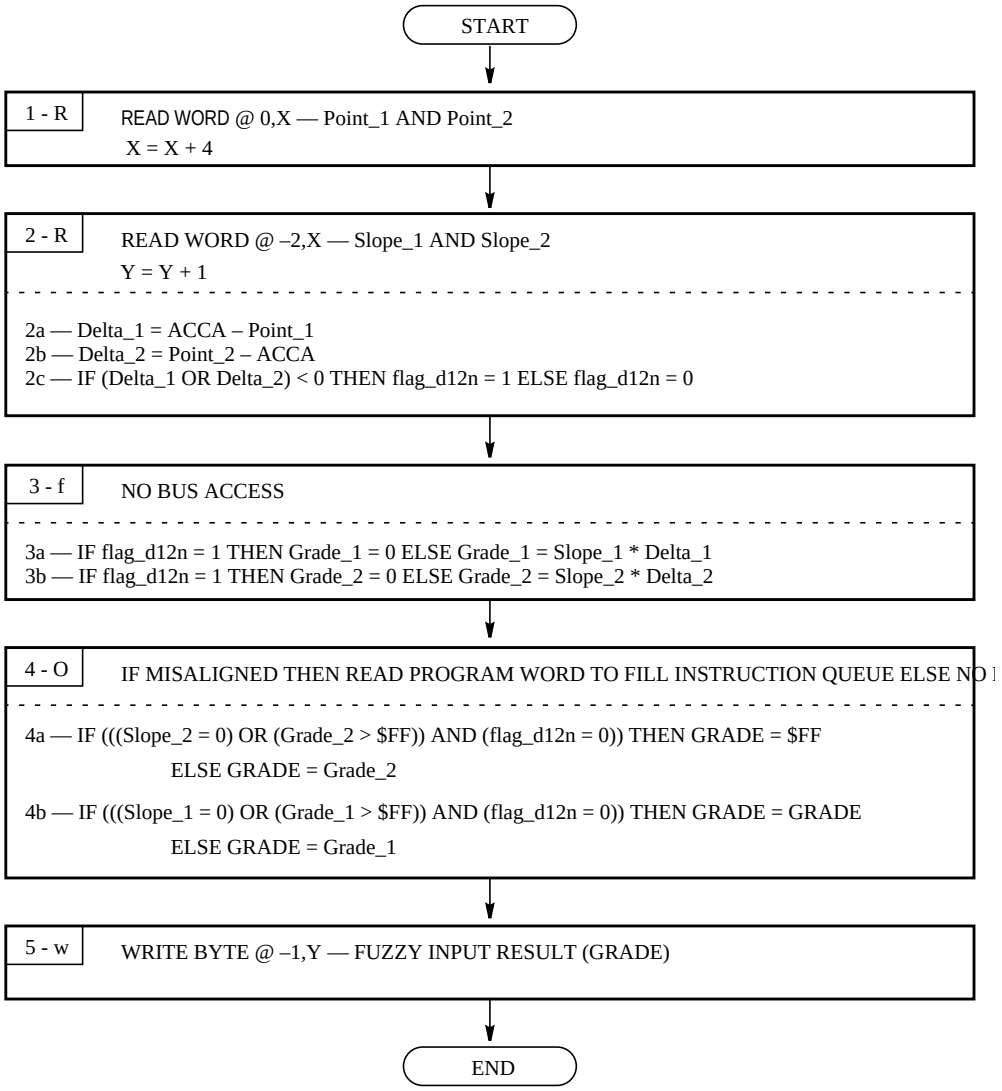


Figure 9-5. MEM Instruction Flow Diagram

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.

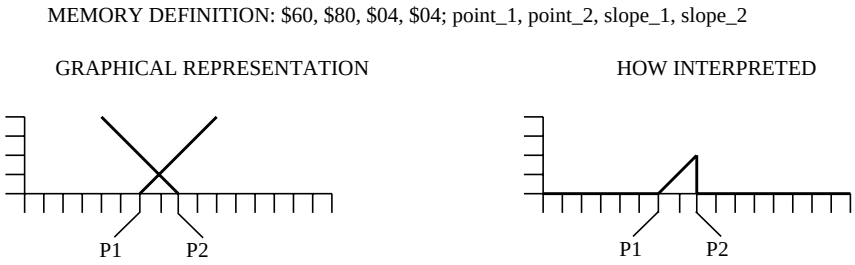


Figure 9-6. Abnormal Membership Function Case 1

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.

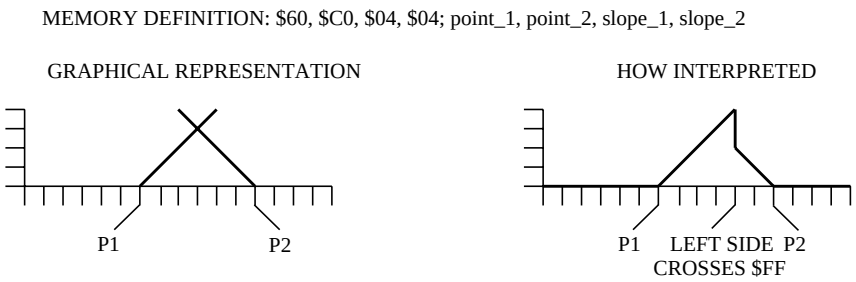


Figure 9-7. Abnormal Membership Function Case 2

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.

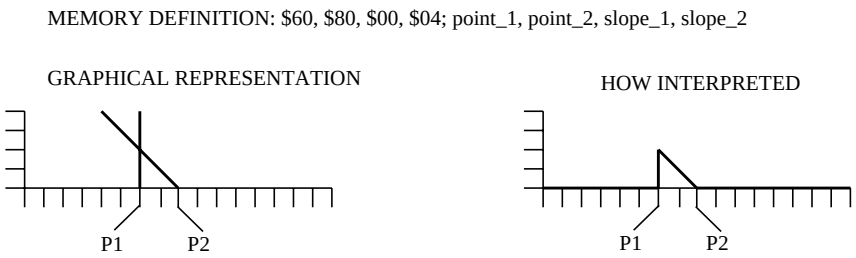


Figure 9-8. Abnormal Membership Function Case 3

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. CPU12 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 CPU12 registers, the interrupt service code can even include independent REV and REVW instructions.

9.5.1.1 Set Up Prior to Executing REV

Some CPU12 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 CPU12 registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU12 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 CPU12 to try to execute a new REV instruction upon return from the interrupt. Since the CPU12 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 CPU12 status on the stack before honoring the requested interrupt.

Figure 9-9 is a REV instruction flow diagram. Each rectangular box represents one CPU12 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 CPU12 until the second cycle after the read takes place. This is due to access and propagation delays.

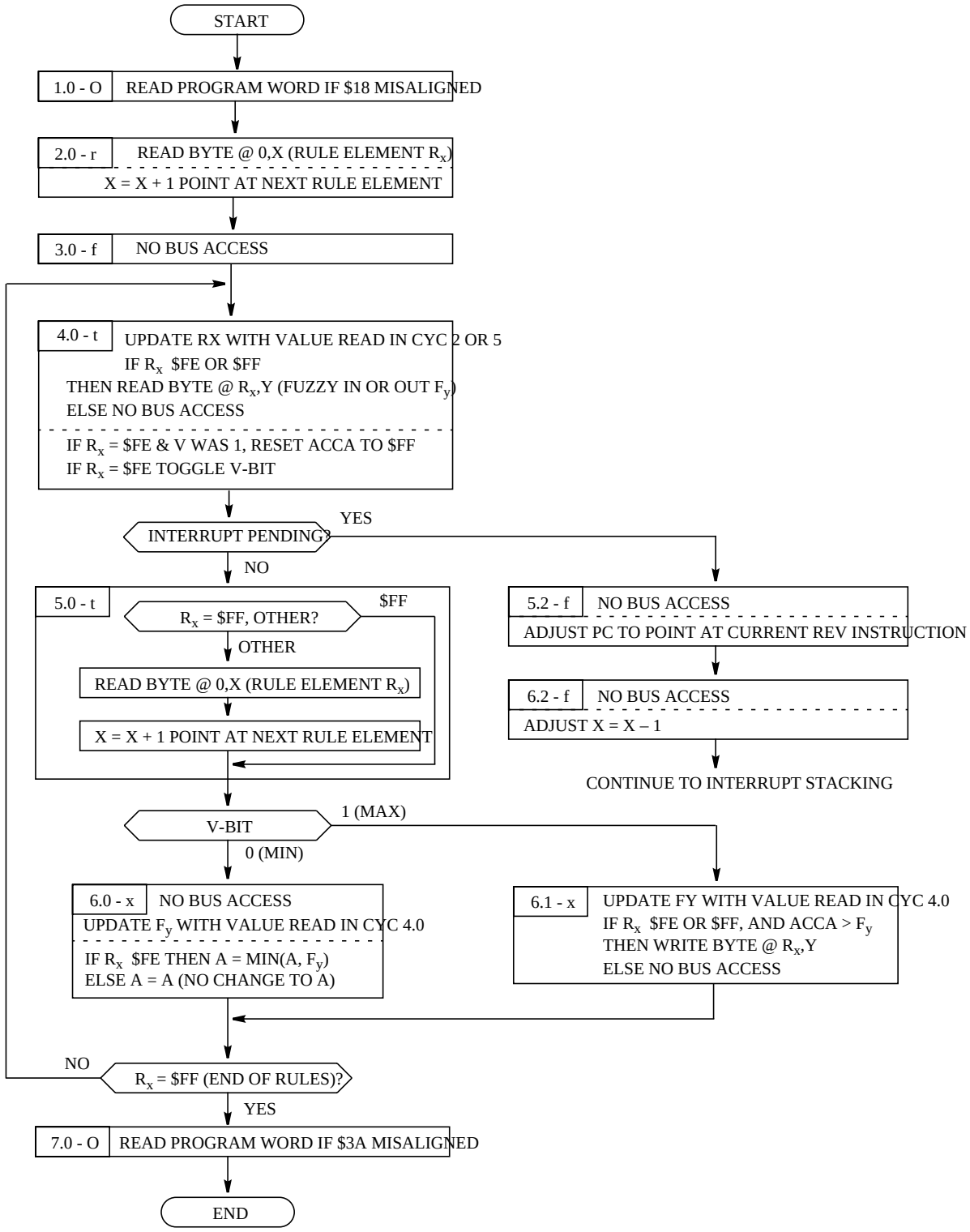


Figure 9-9. REV Instruction Flow Diagram

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 CPU12 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 CPU12 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 CPU12. Some internal CPU12 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 CPU12.

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 CPU12 progresses to cycle 6.0. If V is one, consequents are being processed and the CPU12 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 CPU12 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 CPU12 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 CPU12 registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU12 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 CPU12 to try to execute a new REVW instruction upon return from the interrupt. Since the CPU12 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 CPU12 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 CPU12 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.

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.

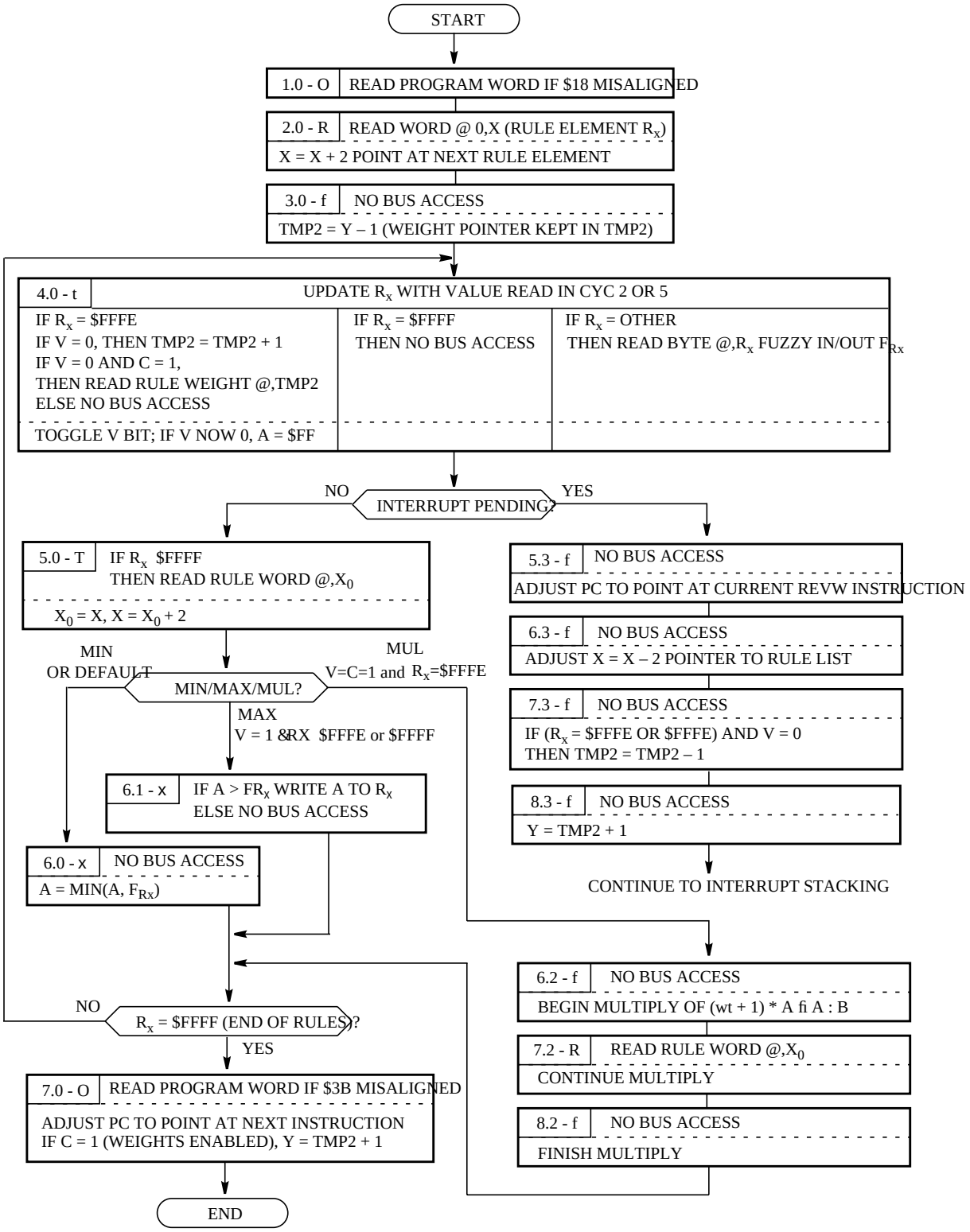


Figure 9-10. REVW Instruction Flow Diagram

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 CPU12 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 CPU12 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 CPU12 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 CPU12 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.

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 CPU12, 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 CPU12 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.

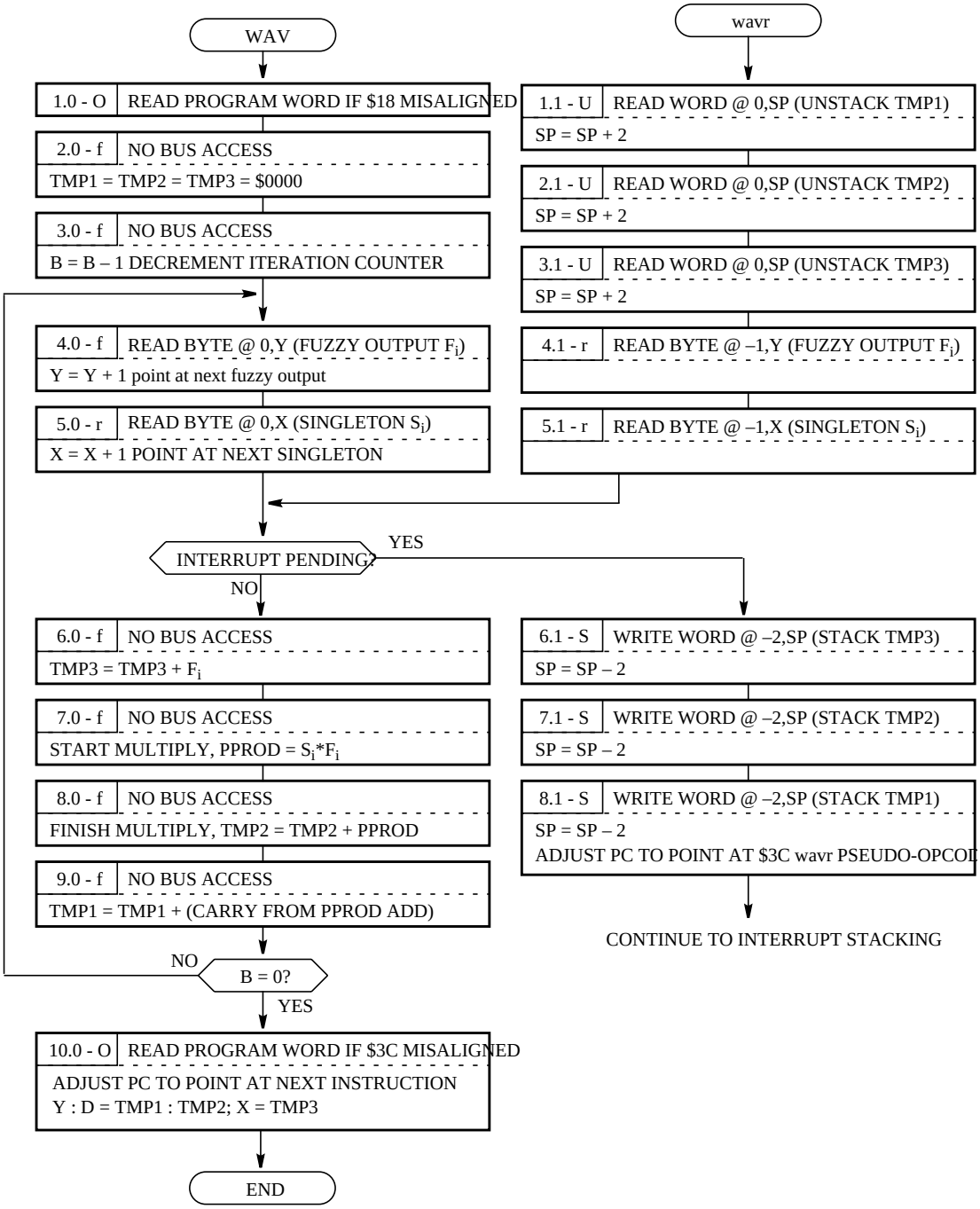


Figure 9-11. WAV and wavr Instruction Flow Diagram (for HCS12)

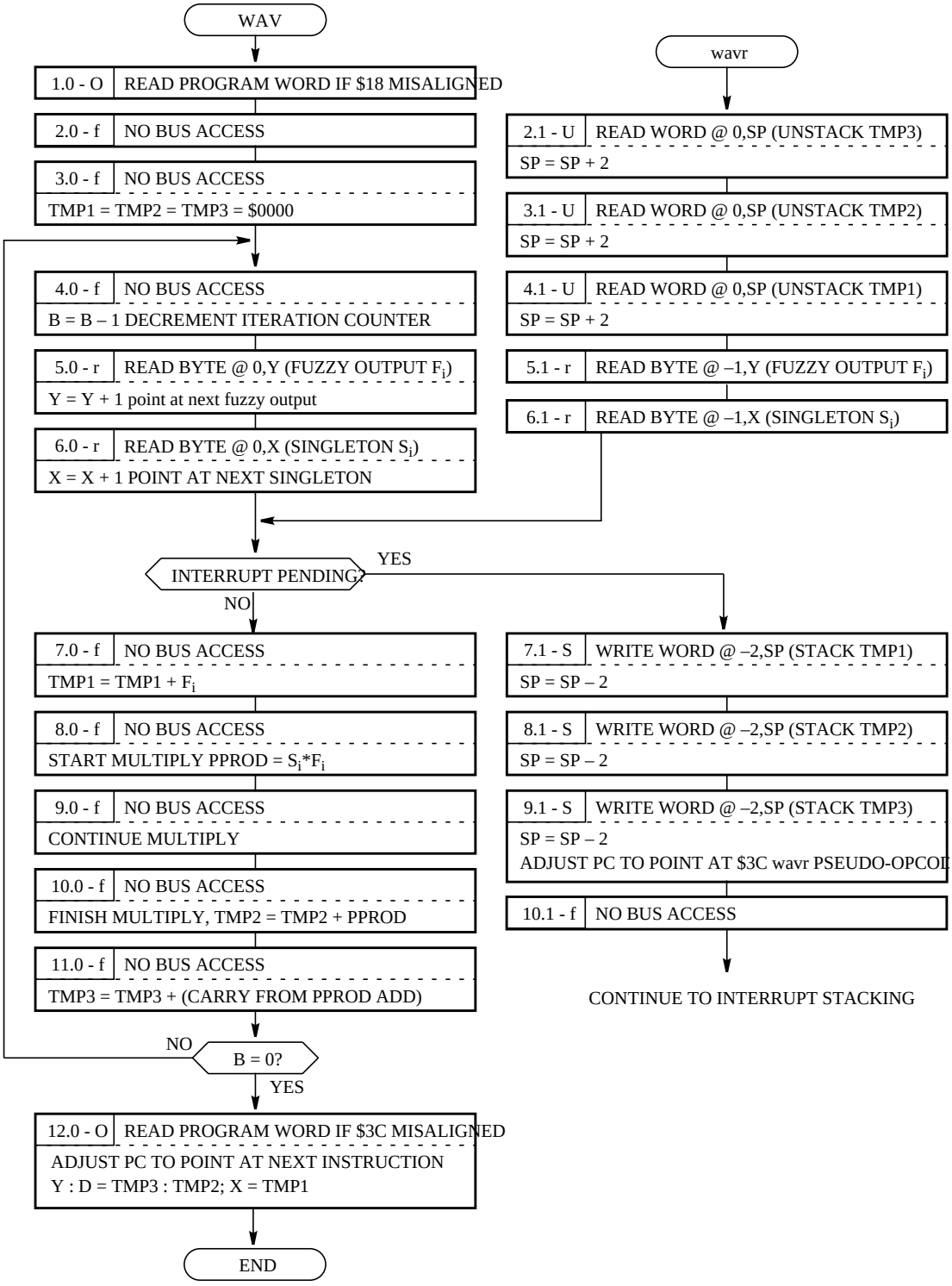


Figure 9-12. WAV and wavr Instruction Flow Diagram (for M68HC12)

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 A^{th} 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.

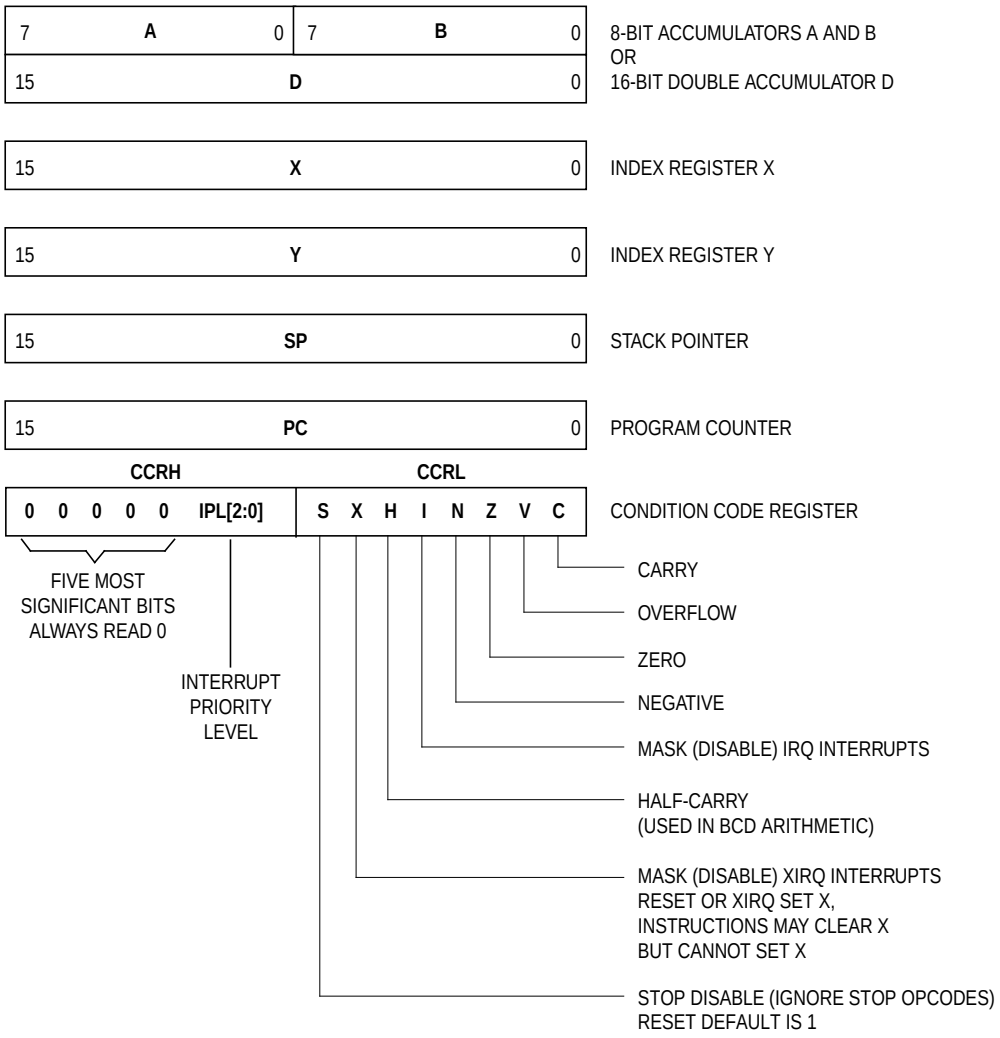
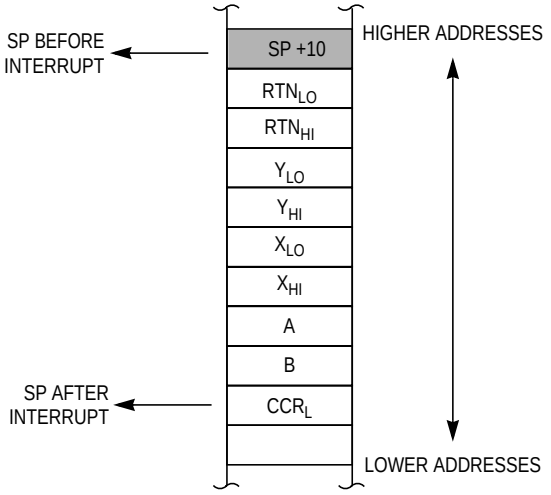


Figure 1. Programming Model

A.2 Stack and Memory Layout

HCS12



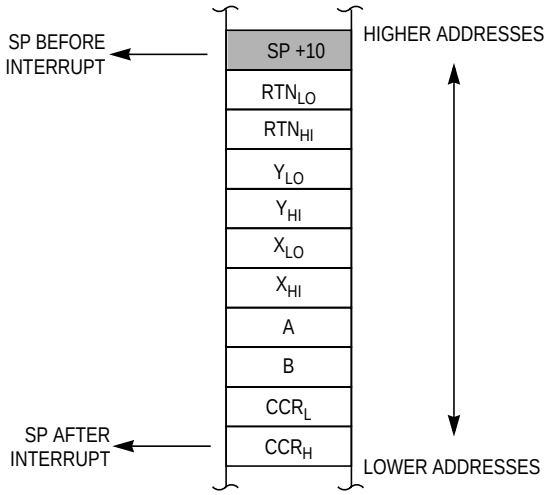
STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS ODD BEFORE INTERRUPT

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

HCS12X



STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS ODD BEFORE INTERRUPT

SP +9	RTN_{LO}		SP +10
SP +7	Y_{LO}	RTN_{HI}	SP +8
SP +5	X_{LO}	Y_{HI}	SP +6
SP +3	A	X_{HI}	SP +4
SP +1	CCR_L	B	SP +2
SP –1		CCR_H	SP

STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS EVEN BEFORE INTERRUPT

SP +10			SP +11
SP +8	RTN_{HI}	RTN_{LO}	SP +9
SP +6	Y_{HI}	Y_{LO}	SP +7
SP +4	X_{HI}	X_{LO}	SP +5
SP +2	B	A	SP +3
SP	CCR_H	CCR_L	SP +1

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
$FF00–$FFF1	Device-Specific Interrupt Sources

A.4 Notation Used in Instruction Set Summary

CPU12 Register Notation

Accumulator A — A or a	Index Register Y — Y or y
Accumulator B — B or b	Stack Pointer — SP, sp, or s
Accumulator D — D or d	Program Counter — PC, pc, or p
Index Register X — X or x	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)

⊕ — 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 505.

`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 507.

`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 503 and **Table A-4** on page 504.

Access Detail

Each code letter except (,), and comma equals one CPU12 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, CPU12 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
NA	— Not available
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.
fl	—	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 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	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	Pf	– – – –	– – – –
ABY	(B) + (Y) ⇒ Y *Translates to* LEAY B,Y	IDX	19 ED	Pf	Pf	– – – –	– – – –
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	rPf		
ADCA opr16a		EXT	B9 hh ll	rPO	rPO		
ADCA oprx0_xysp		IDX	A9 xb	rPf	rPf		
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	fIfrPf		
ADCA [oprx16,xysp]		[IDX2]	A9 xb ee ff	fIPrPf	fIPrPf		
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	rPf		
ADCB opr16a		EXT	F9 hh ll	rPO	rPO		
ADCB oprx0_xysp		IDX	E9 xb	rPf	rPf		
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	fIfrPf		
ADCB [oprx16,xysp]		[IDX2]	E9 xb ee ff	fIPrPf	fIPrPf		
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	rPf		
ADDA opr16a		EXT	BB hh ll	rPO	rPO		
ADDA oprx0_xysp		IDX	AB xb	rPf	rPf		
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	fIfrPf		
ADDA [oprx16,xysp]		[IDX2]	AB xb ee ff	fIPrPf	fIPrPf		
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	rPf		
ADDB opr16a		EXT	FB hh ll	rPO	rPO		
ADDB oprx0_xysp		IDX	EB xb	rPf	rPf		
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	fIfrPf		
ADDB [oprx16,xysp]		[IDX2]	EB xb ee ff	fIPrPf	fIPrPf		
ADDD #opr16i	(A:B) + (M:M+1) ⇒ A:B	IMM	C3 jj kk	PO	PO	– – – –	Δ Δ Δ Δ
ADDD opr8a	Add 16-Bit to D (A:B)	DIR	D3 dd	RPf	RPf		
ADDD opr16a		EXT	F3 hh ll	RPO	RPO		
ADDD oprx0_xysp		IDX	E3 xb	RPf	RPf		
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	fIfRPf		
ADDD [oprx16,xysp]		[IDX2]	E3 xb ee ff	fIPRPf	fIPRPf		
ADDX #opr16i	**(X) + (M:M+1) ⇒ X**	**IMM**	**18 8B jj kk**	**OPO**	**NA**	– – Δ –	Δ Δ Δ Δ
ADDX opr8a	**Add without Carry to X**	**DIR**	**18 9B dd**	**ORPf**	**NA**		
ADDX opr16a		**EXT**	**18 BB hh ll**	**ORPO**	**NA**		
ADDX oprx0_xysp		**IDX**	**18 AB xb**	**ORPf**	**NA**		
ADDX oprx9,xysp		**IDX1**	**18 AB xb ff**	**ORPO**	**NA**		
ADDX oprx16,xysp		**IDX2**	**18 AB xb ee ff**	**OfRPP**	**NA**		
ADDX [D,xysp]		**[D,IDX]**	**18 AB xb**	**OfIfRPf**	**NA**		
ADDX [oprx16,xysp]		**[IDX2]**	**18 AB xb ee ff**	**OfIPRPf**	**NA**		
ADDY #opr16i	**(Y) + (M:M+1) ⇒ Y**	**IMM**	**18 CB jj kk**	**OPO**	**NA**	– – Δ –	Δ Δ Δ Δ
ADDY opr8a	**Add without Carry to Y**	**DIR**	**18 DB dd**	**ORPf**	**NA**		
ADDY opr16a		**EXT**	**18 FB hh ll**	**ORPO**	**NA**		
ADDY oprx0_xysp		**IDX**	**18 EB xb**	**ORPf**	**NA**		
ADDY oprx9,xysp		**IDX1**	**18 EB xb ff**	**ORPO**	**NA**		
ADDY oprx16,xysp		**IDX2**	**18 EB xb ee ff**	**OfRPP**	**NA**		
ADDY [D,xysp]		**[D,IDX]**	**18 EB xb**	**OfIfRPf**	**NA**		
ADDY [oprx16,xysp]		**[IDX2]**	**18 EB xb ee ff**	**OfIPRPf**	**NA**		

Table A-1. Instruction Set Summary (Sheet 2 of 21)

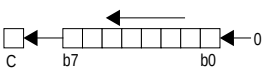
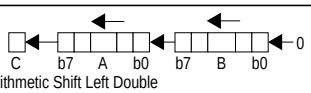
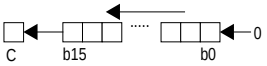
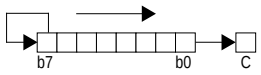
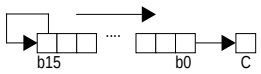
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
ADED #opr16i	**(A:B) + (M:M+1) + C ⇒ A:B**	**IMM**	**18 C3 jj kk**	**OPO**	**NA**	– – Δ –	Δ Δ Δ Δ
ADED opr8a	**Add with Carry to D (A:B)**	**DIR**	**18 D3 dd**	**ORPf**	**NA**		
ADED opr16a		**EXT**	**18 F3 hh ll**	**ORPO**	**NA**		
ADED oprx0_xysp		**IDX**	**18 E3 xb**	**ORPf**	**NA**		
ADED oprx9,xysp		**IDX1**	**18 E3 xb ff**	**ORPO**	**NA**		
ADED oprx16,xysp		**IDX2**	**18 E3 xb ee ff**	**OfRPP**	**NA**		
ADED [D,xysp]		**[D,IDX]**	**18 E3 xb**	**OfIfRPf**	**NA**		
ADED [oprx16,xysp]		**[IDX2]**	**18 E3 xb ee ff**	**OfIPRPf**	**NA**		
ADEX #opr16i	**(X) + (M:M+1) + C ⇒ X**	**IMM**	**18 89 jj kk**	**OPO**	**NA**	– – Δ –	Δ Δ Δ Δ
ADEX opr8a	**Add with Carry to X**	**DIR**	**18 99 dd**	**ORPf**	**NA**		
ADEX opr16a		**EXT**	**18 B9 hh ll**	**ORPO**	**NA**		
ADEX oprx0_xysp		**IDX**	**18 A9 xb**	**ORPf**	**NA**		
ADEX oprx9,xysp		**IDX1**	**18 A9 xb ff**	**ORPO**	**NA**		
ADEX oprx16,xysp		**IDX2**	**18 A9 xb ee ff**	**OfRPP**	**NA**		
ADEX [D,xysp]		**[D,IDX]**	**18 A9 xb**	**OfIfRPf**	**NA**		
ADEX [oprx16,xysp]		**[IDX2]**	**18 A9 xb ee ff**	**OfIPRPf**	**NA**		
ADEY #opr16i	**(Y) + (M:M+1) + C ⇒ Y**	**IMM**	**18 C9 jj kk**	**OPO**	**NA**	– – Δ –	Δ Δ Δ Δ
ADEY opr8a	**Add with Carry to Y**	**DIR**	**18 D9 dd**	**ORPf**	**NA**		
ADEY opr16a		**EXT**	**18 F9 hh ll**	**ORPO**	**NA**		
ADEY oprx0_xysp		**IDX**	**18 E9 xb**	**ORPf**	**NA**		
ADEY oprx9,xysp		**IDX1**	**18 E9 xb ff**	**ORPO**	**NA**		
ADEY oprx16,xysp		**IDX2**	**18 E9 xb ee ff**	**OfRPP**	**NA**		
ADEY [D,xysp]		**[D,IDX]**	**18 E9 xb**	**OfIfRPf**	**NA**		
ADEY [oprx16,xysp]		**[IDX2]**	**18 E9 xb ee ff**	**OfIPRPf**	**NA**		
ANDA #opr8i	(A) • (M) ⇒ A	IMM	84 ii	P	P	– – – –	Δ Δ 0 –
ANDA opr8a	Logical AND A with Memory	DIR	94 dd	rPf	rPf		
ANDA opr16a		EXT	B4 hh ll	rPO	rPO		
ANDA oprx0_xysp		IDX	A4 xb	rPf	rPf		
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	fIfrPf		
ANDA [oprx16,xysp]		[IDX2]	A4 xb ee ff	fIPrPf	fIPrPf		
ANDB #opr8i	(B) • (M) ⇒ B	IMM	C4 ii	P	P	– – – –	Δ Δ 0 –
ANDB opr8a	Logical AND B with Memory	DIR	D4 dd	rPf	rPf		
ANDB opr16a		EXT	F4 hh ll	rPO	rPO		
ANDB oprx0_xysp		IDX	E4 xb	rPf	rPf		
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	fIfrPf		
ANDB [oprx16,xysp]		[IDX2]	E4 xb ee ff	fIPrPf	fIPrPf		
ANDCC #opr8i	(CCR) • (M) ⇒ CCR Logical AND CCR with Memory	IMM	10 ii	P	P	⇓ ⇓ ⇓ ⇓	⇓ ⇓ ⇓ ⇓
ANDX #opr16i	**(X) • (M:M+1) ⇒ X**	**IMM**	**18 84 jj kk**	**OPO**	**NA**	– – – –	Δ Δ 0 –
ANDX opr8a	**Logical AND X with Memory**	**DIR**	**18 94 dd**	**ORPf**	**NA**		
ANDX opr16a		**EXT**	**18 B4 hh ll**	**ORPO**	**NA**		
ANDX oprx0_xysp		**IDX**	**18 A4 xb**	**ORPf**	**NA**		
ANDX oprx9,xysp		**IDX1**	**18 A4 xb ff**	**ORPO**	**NA**		
ANDX oprx16,xysp		**IDX2**	**18 A4 xb ee ff**	**OfRPP**	**NA**		
ANDX [D,xysp]		**[D,IDX]**	**18 A4 xb**	**OfIfRPf**	**NA**		
ANDX [oprx16,xysp]		**[IDX2]**	**18 A4 xb ee ff**	**OfIPRPf**	**NA**		
ANDY #opr16i	**(Y) • (M:M+1) ⇒ Y**	**IMM**	**18 C4 jj kk**	**OPO**	**NA**	– – – –	Δ Δ 0 –
ANDY opr8a	**Logical AND Y with Memory**	**DIR**	**18 D4 dd**	**ORPf**	**NA**		
ANDY opr16a		**EXT**	**18 F4 hh ll**	**ORPO**	**NA**		
ANDY oprx0_xysp		**IDX**	**18 E4 xb**	**ORPf**	**NA**		
ANDY oprx9,xysp		**IDX1**	**18 E4 xb ff**	**ORPO**	**NA**		
ANDY oprx16,xysp		**IDX2**	**18 E4 xb ee ff**	**OfRPP**	**NA**		
ANDY [D,xysp]		**[D,IDX]**	**18 E4 xb**	**OfIfRPf**	**NA**		
ANDY [oprx16,xysp]		**[IDX2]**	**18 E4 xb ee ff**	**OfIPRPf**	**NA**		
ASL opr16a		EXT	78 hh ll	rPwO	rPwO	– – – –	Δ Δ Δ Δ
ASL oprx0_xysp		IDX	68 xb	rPw	rPw		
ASL oprx9,xysp		IDX1	68 xb ff	rPwO	rPwO		
ASL oprx16,xysp		IDX2	68 xb ee ff	frPwP	frPwP		
ASL [D,xysp]	Arithmetic Shift Left	[D,IDX]	68 xb	fIfrPw	fIfrPw		
ASL [oprx16,xysp]		[IDX2]	68 xb ee ff	fIPrPw	fIPrPw		
ASLA	Arithmetic Shift Left Accumulator A	INH	48	O	O		
ASLB	Arithmetic Shift Left Accumulator B	INH	58	O	O		

Table A-1. Instruction Set Summary (Sheet 3 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
ASLD	 Arithmetic Shift Left Double	INH	59	O	O	– – – –	Δ Δ Δ Δ
ASLW opr16a **ASLW oprx0_xysp** **ASLW oprx9,xysp** **ASLW oprx16,xysp** **ASLW [D,xysp]** **ASLW [oprx16,xysp]** **ASLX** **ASLY**	 **Arithmetic Shift Left** **Arithmetic Shift Left Index Register X** **Arithmetic Shift Left Index Register Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 78 hh ll** **18 68 xb** **18 68 xb ff** **18 68 xb ee ff** **18 68 xb** **18 68 xb ee ff** **18 48** **18 58**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – –	Δ Δ Δ Δ
ASRW opr16a **ASRW oprx0_xysp** **ASRW oprx9,xysp** **ASRW oprx16,xysp** **ASRW [D,xysp]** **ASRW [oprx16,xysp]** **ASRX** **ASRY**	 **Arithmetic Shift Right** **Arithmetic Shift Right Index Register X** **Arithmetic Shift Right Index Register Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 77 hh ll** **18 67 xb** **18 67 xb ff** **18 67 xb ee ff** **18 67 xb** **18 67 xb ee ff** **18 47** **18 57**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	Δ Δ Δ Δ
BCC rel8	Branch if Carry Clear (if C = 0)	REL	24 rr	PPP/P[1]	PPP/P[1]	– – – –	– – – –
BCLR opr8a, msk8 BCLR opr16a, msk8 BCLR oprx0_xysp, msk8 BCLR oprx9,xysp, msk8 BCLR oprx16,xysp, msk8	(M) • ($\overline{mm}$) ⇒ M Clear Bit(s) in Memory	DIR EXT IDX IDX1 IDX2	4D dd mm 1D hh ll mm 0D xb mm 0D xb ff mm 0D xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	rPwO rPwP rPwO rPwP frPwPO	– – – –	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 CPU12 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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – –	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – –	Δ Δ 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.

Table A-1. Instruction Set Summary (Sheet 4 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
BITX #opr16i **BITX opr8a** **BITX opr16a** **BITX oprx0_xysp** **BITX oprx9,xysp** **BITX oprx16,xysp** **BITX [D,xysp]** **BITX [oprx16,xysp]**	**(x) • (M:M+1)** **Logical AND X with Memory** **Does not change Index Register or Memory**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 85 jj kk** **18 95 dd** **18 B5 hh ll** **18 A5 xb** **18 A5 xb ff** **18 A5 xb ee ff** **18 A5 xb** **18 A5 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	Δ Δ 0 –
BITY #opr16i **BITY opr8a** **BITY opr16a** **BITY oprx0_xysp** **BITY oprx9,xysp** **BITY oprx16,xysp** **BITY [D,xysp]** **BITY [oprx16,xysp]**	**(Y) • (M:M+1)** **Logical AND Ywith Memory** **Does not change Index Register or Memory**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 C5 jj kk** **18 D5 dd** **18 F5 hh ll** **18 E5 xb** **18 E5 xb ff** **18 E5 xb ee ff** **18 E5 xb** **18 E5 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	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]	– – – –	– – – –
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 rfPPP PrfPPP	– – – –	– – – –
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 rfPPP PrfPPP	– – – –	– – – –
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 Set CCR flags with respect to the result	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	rPwO rPwP rPwO rPwP frPwPO	– – – –	Δ Δ 0 –
BSR rel8	(SP) – 2 ⇒ SP; RTN_H:RTN_L ⇒ $M_{(SP)}$:$M_{(SP+1)}$ Subroutine address fi PC Branch to Subroutine	REL	07 rr	SPPP	SPPP	– – – –	– – – –
BTAS opr8, msk8 **BTAS opr16a, msk8** **BTAS oprx0_xysp, msk8** **BTAS oprx9,xysp, msk8** **BTAS oprx16,xysp, msk8**	**(M) | (Mask) ⇒ M** **Set Bit(s) in Memory** **Set CCR flags with respect to operand (M) read**	**DIR** **EXT** **IDX** **IDX1** **IDX2**	**18 35 dd mm** **18 36 hh ll mm** **18 37 xb mm** **18 37 xb ff mm** **18 37 xb ee ff mm**	**ORPWO** **ORPWP** **ORPWO** **ORPWP** **OfRPWPO**	**NA** **NA** **NA** **NA** **NA**	– – – –	Δ Δ 0 –
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]	– – – –	– – – –

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 5 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
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	gnSsPPP gnSsPPP gnSsPPP fgnSsPPP 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 ⇒ MClear Memory Location 0 ⇒ AClear Accumulator A 0 ⇒ BClear 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	PwO Pw PwO PwP PIfw PIPw O O	----	0100
CLRW opr16a **CLRW oprx0_xysp** **CLRW oprx9,xysp** **CLRW oprx16,xysp** **CLRW [D,xysp]** **CLRW [oprx16,xysp]** **CLRX** **CLRY**	**0 ⇒ M:M+1Clear Memory Location** **0 ⇒ XClear Index Register X** **0 ⇒ YClear Index Register Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 79 hh ll** **18 69 xb** **18 69 xb ff** **18 69 xb ee ff** **18 69 xb** **18 69 xb ee ff** **18 87** **18 C7**	**OPWO** **OPW** **OPWO** **OPWP** **OPIfW** **OPIPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	0100
CLV	0 ⇒ V *Translates to* ANDCC #$FD	IMM	10 FD	P	P	----	--0-

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
					rPf
					rPO
					rPf
					rPO
					frPP
					fIfrPf
					fIPrPf
						----	Δ Δ Δ Δ

CMPB #opr8i
CMPB opr8a
CMPB opr16a
CMPB oprx0_xysp
CMPB oprx9,xysp
CMPB oprx16,xysp
CMPB [D,xysp]
CMPB [oprx16,xysp]
	(B) – (M)
	Compare Accumulator B with Memory
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C1 ii
			D1 dd
			F1 hh ll
			E1 xb
			E1 xb ff
			E1 xb ee ff
			E1 xb
			E1 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					P
					rPf
					rPO
					rPf
					rPO
					frPP
					fIfrPf
					fIPrPf
						----	Δ Δ Δ Δ

COM opr16a COM oprx0_xysp COM oprx9,xysp COM oprx16,xysp COM [D,xysp] COM [oprx16,xysp] COMA COMB	($\overline{M}$) ⇒ M *equivalent to* $FF – (M) ⇒ M 1's Complement Memory Location ($\overline{A}$) ⇒ AComplement Accumulator A ($\overline{B}$) ⇒ BComplement Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	71 hh ll 61 xb 61 xb ff 61 xb ee ff 61 xb 61 xb ee ff 41 51	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	----	ΔΔ01
COMW opr16a **COMW oprx0_xysp** **COMW oprx9,xysp** **COMW oprx16,xysp** **COMW [D,xysp]** **COMW [oprx16,xysp]** **COMX** **COMY**	**($\overline{M:M+1}$) ⇒ M:M+1 equivalent to $FF – (M:M+1) ⇒ M:M+1** **($\overline{X}$) ⇒ XComplement Index Register X** **($\overline{Y}$) ⇒ YComplement Index Register Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 71 hh ll** **18 61 xb** **18 61 xb ff** **18 61 xb ee ff** **18 61 xb** **18 61 xb ee ff** **18 41** **18 51**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	ΔΔ01

Table A-1. Instruction Set Summary (Sheet 6 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
CPD #opr16i	(A:B) – (M:M+1)	IMM	8C jj kk	PO	PO	----	Δ Δ Δ Δ
CPD opr8a	Compare D to Memory (16-Bit)	DIR	9C dd	RPf	RPf		
CPD opr16a		EXT	BC hh ll	RPO	RPO		
CPD oprx0_xysp		IDX	AC xb	RPf	RPf		
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	fIfRPf		
CPD [oprx16,xysp]		[IDX2]	AC xb ee ff	fIPRPf	fIPRPf		
CPED #opr16i	**(A:B) – (M:M+1) – C**	**IMM**	**18 8C jj kk**	**OPO**	**NA**	----	Δ Δ Δ Δ
CPED opr8a	**Compare D to Memory with Borrow**	**DIR**	**18 9C dd**	**ORPf**	**NA**		
CPED opr16a		**EXT**	**18 BC hh ll**	**ORPO**	**NA**		
CPED oprx0_xysp		**IDX**	**18 AC xb**	**ORPf**	**NA**		
CPED oprx9,xysp		**IDX1**	**18 AC xb ff**	**ORPO**	**NA**		
CPED oprx16,xysp		**IDX2**	**18 AC xb ee ff**	**OfRPP**	**NA**		
CPED [D,xysp]		**[D,IDX]**	**18 AC xb**	**OfIfRPf**	**NA**		
CPED [oprx16,xysp]		**[IDX2]**	**18 AC xb ee ff**	**OfIPRPf**	**NA**		
CPES #opr16i	**(SP) – (M:M+1) – C**	**IMM**	**18 8F jj kk**	**OPO**	**NA**	----	Δ Δ Δ Δ
CPES opr8a	**Compare SP to Memory with Borrow**	**DIR**	**18 9F dd**	**ORPf**	**NA**		
CPES opr16a		**EXT**	**18 BF hh ll**	**ORPO**	**NA**		
CPES oprx0_xysp		**IDX**	**18 AF xb**	**ORPf**	**NA**		
CPES oprx9,xysp		**IDX1**	**18 AF xb ff**	**ORPO**	**NA**		
CPES oprx16,xysp		**IDX2**	**18 AF xb ee ff**	**OfRPP**	**NA**		
CPES [D,xysp]		**[D,IDX]**	**18 AF xb**	**OfIfRPf**	**NA**		
CPES [oprx16,xysp]		**[IDX2]**	**18 AF xb ee ff**	**OfIPRPf**	**NA**		
CPEX #opr16i	**(X) – (M:M+1) – C**	**IMM**	**18 8E jj kk**	**OPO**	**NA**	----	Δ Δ Δ Δ
CPEX opr8a	**Compare X to Memory with Borrow**	**DIR**	**18 9E dd**	**ORPf**	**NA**		
CPEX opr16a		**EXT**	**18 BE hh ll**	**ORPO**	**NA**		
CPEX oprx0_xysp		**IDX**	**18 AE xb**	**ORPf**	**NA**		
CPEX oprx9,xysp		**IDX1**	**18 AE xb ff**	**ORPO**	**NA**		
CPEX oprx16,xysp		**IDX2**	**18 AE xb ee ff**	**OfRPP**	**NA**		
CPEX [D,xysp]		**[D,IDX]**	**18 AE xb**	**OfIfRPf**	**NA**		
CPEX [oprx16,xysp]		**[IDX2]**	**18 AE xb ee ff**	**OfIPRPf**	**NA**		
CPEY #opr16i	**(Y) – (M:M+1) – C**	**IMM**	**18 8D jj kk**	**OPO**	**NA**	----	Δ Δ Δ Δ
CPEY opr8a	**Compare Y to Memory with Borrow**	**DIR**	**18 9D dd**	**ORPf**	**NA**		
CPEY opr16a		**EXT**	**18 BD hh ll**	**ORPO**	**NA**		
CPEY oprx0_xysp		**IDX**	**18 AD xb**	**ORPf**	**NA**		
CPEY oprx9,xysp		**IDX1**	**18 AD xb ff**	**ORPO**	**NA**		
CPEY oprx16,xysp		**IDX2**	**18 AD xb ee ff**	**OfRPP**	**NA**		
CPEY [D,xysp]		**[D,IDX]**	**18 AD xb**	**OfIfRPf**	**NA**		
CPEY [oprx16,xysp]		**[IDX2]**	**18 AD xb ee ff**	**OfIPRPf**	**NA**		
CPS #opr16i	(SP) – (M:M+1)	IMM	8F jj kk	PO	PO	----	Δ Δ Δ Δ
CPS opr8a	Compare SP to Memory (16-Bit)	DIR	9F dd	RPf	RPf		
CPS opr16a		EXT	BF hh ll	RPO	RPO		
CPS oprx0_xysp		IDX	AF xb	RPf	RPf		
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	fIfRPf		
CPS [oprx16,xysp]		[IDX2]	AF xb ee ff	fIPRPf	fIPRPf		
CPX #opr16i	(X) – (M:M+1)	IMM	8E jj kk	PO	PO	----	Δ Δ Δ Δ
CPX opr8a	Compare X to Memory (16-Bit)	DIR	9E dd	RPf	RPf		
CPX opr16a		EXT	BE hh ll	RPO	RPO		
CPX oprx0_xysp		IDX	AE xb	RPf	RPf		
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	fIfRPf		
CPX [oprx16,xysp]		[IDX2]	AE xb ee ff	fIPRPf	fIPRPf		
CPY #opr16i	(Y) – (M:M+1)	IMM	8D jj kk	PO	PO	----	Δ Δ Δ Δ
CPY opr8a	Compare Y to Memory (16-Bit)	DIR	9D dd	RPf	RPf		
CPY opr16a		EXT	BD hh ll	RPO	RPO		
CPY oprx0_xysp		IDX	AD xb	RPf	RPf		
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	fIfRPf		
CPY [oprx16,xysp]		[IDX2]	AD xb ee ff	fIPRPf	fIPRPf		
DAA	Adjust Sum to BCD Decimal Adjust Accumulator A	INH	18 07	OfO	OfO	----	Δ Δ ? Δ

Table A-1. Instruction Set Summary (Sheet 7 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
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 (branch) PPO (no branch)	– – – –	– – – –
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 (branch) PPO (no branch)	– – – –	– – – –
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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – –	Δ Δ Δ –
DECW opr16a **DECW oprx0_xysp** **DECW oprx9,xysp** **DECW oprx16,xysp** **DECW [D,xysp]** **DECW [oprx16,xysp]** **DECX** **DECY**	**(M:M+1) – $01 ⇒ M:M+1** **Decrement Memory Location** **(X) – $01 ⇒ X Decrement X** **(Y) – $01 ⇒ Y Decrement Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 73 hh ll** **18 63 xb** **18 63 xb ff** **18 63 xb ee ff** **18 63 xb** **18 63 xb ee ff18** **18 43** **18 53**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	Δ Δ Δ –
DES	(SP) – $0001 ⇒ SP *Translates to* LEAS –1,SP	IDX	1B 9F	Pf	Pf	– – – –	– – – –
DEX	(X) – $0001 ⇒ X Decrement Index Register X	INH	09	O	O	– – – –	– Δ – –
DEY	(Y) – $0001 ⇒ Y Decrement Index Register Y	INH	03	O	O	– – – –	– Δ – –
EDIV	(Y:D) ÷ (X) ⇒ Y Remainder fi D 32 by 16 Bit ⇒ 16 Bit Divide (unsigned)	INH	11	fffffffffO	fffffffffO	– – – –	Δ Δ Δ Δ
EDIVS	(Y:D) ÷ (X) ⇒ Y Remainder fi D 32 by 16 Bit ⇒ 16 Bit Divide (signed)	INH	18 14	OfffffffffO	OfffffffffO	– – – –	Δ Δ Δ Δ
EMACS opr16a[1]	($M_{(X)}$:$M_{(X+1)}$) × ($M_{(Y)}$:$M_{(Y+1)}$) + (M~M+3) ⇒ M~M+3 16 by 16 Bit ⇒ 32 Bit Multiply and Accumulate (signed)	Special	18 12 hh ll	ORRORRWPP	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	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – –	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	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – –	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

Note:1. *opr16a* is an extended address specifiation. Both X and Y point to source operands.

Table A-1. Instruction Set Summary (Sheet 8 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
EMUL	(D) × (Y) ⇒ Y:D 16 by 16 Bit Multiply (unsigned)	INH	13	O	ff0	----	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 OffO	----	Δ Δ – D
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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	Δ Δ 0 –
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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	Δ Δ 0 –
EORX #opr16i **EORX opr8a** **EORX opr16a** **EORX oprx0_xysp** **EORX oprx9,xysp** **EORX oprx16,xysp** **EORX [D,xysp]** **EORX [oprx16,xysp]**	**(X) ⊕ (M:M+1) ⇒ X** **Exclusive-OR X with Memory**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 88 jj kk** **18 98 dd** **18 B8 hh ll** **18 A8 xb** **18 A8 xb ff** **18 A8 xb ee ff** **18 A8 xb** **18 A8 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ 0 –
EORY #opr16i **EORY opr8a** **EORY opr16a** **EORY oprx0_xysp** **EORY oprx9,xysp** **EORY oprx16,xysp** **EORY [D,xysp]** **EORY [oprx16,xysp]**	**(Y) ⊕ (M:M+1) ⇒ Y** **Exclusive-OR Y with Memory**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 C8 jj kk** **18 D8 dd** **18 F8 hh ll** **18 E8 xb** **18 E8 xb ff** **18 E8 xb ee ff** **18 E8 xb** **18 E8 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ 0 –
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	----	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 fi D 16 by 16 Bit Fractional Divide	INH	18 11	OfffffffffO	OfffffffffO	----	– Δ Δ Δ
GLDAA opr8a **GLDAA opr16a** **GLDAA oprx0_xysp** **GLDAA oprx9,xysp** **GLDAA oprx16,xysp** **GLDAA [D,xysp]** **GLDAA [oprx16,xysp]**	**G(M) ⇒ A** **Load Accumulator A from Global Memory**	**DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 96 dd** **18 B6 hh ll** **18 A6 xb** **18 A6 xb ff** **18 A6 xb ee ff** **18 A6 xb** **18 A6 xb ee ff**	**OrPf** **OrPO** **OrPf** **OrPO** **OfrPP** **OfIfrPf** **OfIPrPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ 0 –

Table A-1. Instruction Set Summary (Sheet 9 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
GLDAB oprx8a	G(M) ⇒ B	DIR	18 D6 dd	OrPf	NA	– – – –	Δ Δ 0 –
GLDAB oprx16a	Load Accumulator B from Global Memory	EXT	18 F6 hh ll	OrPO	NA		
GLDAB oprx0_xysp		IDX	18 E6 xb	OrPf	NA		
GLDAB oprx9,xysp		IDX1	18 E6 xb ff	OrPO	NA		
GLDAB oprx16,xysp		IDX2	18 E6 xb ee ff	OfrPP	NA		
GLDAB [D,xysp]		[D,IDX]	18 E6 xb	OfIfrPf	NA		
GLDAB [oprx16,xysp]		[IDX2]	18 E6 xb ee ff	OfIPrPf	NA		
GLDD oprx8a	G(M:M+1) ⇒ A:B	DIR	18 DC dd	ORPf	NA	– – – –	Δ Δ 0 –
GLDD oprx16a	Load Double Accumulator D (A:B) from Global Memo-	EXT	18 FC hh ll	ORPO	NA		
GLDD oprx0_xysp	ry	IDX	18 EC xb	ORPf	NA		
GLDD oprx9,xysp		IDX1	18 EC xb ff	ORPO	NA		
GLDD oprx16,xysp		IDX2	18 EC xb ee ff	OfRPP	NA		
GLDD [D,xysp]		[D,IDX]	18 EC xb	OfIfRPf	NA		
GLDD [oprx16,xysp]		[IDX2]	18 EC xb ee ff	OfIPRPf	NA		
GLDS oprx8a	G(M:M+1) ⇒ SP	DIR	18 DF dd	ORPf	NA	– – – –	Δ Δ 0 –
GLDS oprx16a	Load Stack Pointer from Global Memory	EXT	18 FF hh ll	ORPO	NA		
GLDS oprx0_xysp		IDX	18 EF xb	ORPf	NA		
GLDS oprx9,xysp		IDX1	18 EF xb ff	ORPO	NA		
GLDS oprx16,xysp		IDX2	18 EF xb ee ff	OfRPP	NA		
GLDS [D,xysp]		[D,IDX]	18 EF xb	OfIfRPf	NA		
GLDS [oprx16,xysp]		[IDX2]	18 EF xb ee ff	OfIPRPf	NA		
GLDX oprx8a	G(M:M+1) ⇒ X	DIR	18 DE dd	ORPf	NA	– – – –	Δ Δ 0 –
GLDX oprx16a	Load Index Register X from Global Memory	EXT	18 FE hh ll	ORPO	NA		
GLDX oprx0_xysp		IDX	18 EE xb	ORPf	NA		
GLDX oprx9,xysp		IDX1	18 EE xb ff	ORPO	NA		
GLDX oprx16,xysp		IDX2	18 EE xb ee ff	OfRPP	NA		
GLDX [D,xysp]		[D,IDX]	18 EE xb	OfIfRPf	NA		
GLDX [oprx16,xysp]		[IDX2]	18 EE xb ee ff	OfIPRPf	NA		
GLDY oprx8a	G(M:M+1) ⇒ Y	DIR	18 DD dd	ORPf	NA	– – – –	Δ Δ 0 –
GLDY oprx16a	Load Index Register Y from Global Memory	EXT	18 FD hh ll	ORPO	NA		
GLDY oprx0_xysp		IDX	18 ED xb	ORPf	NA		
GLDY oprx9,xysp		IDX1	18 ED xb ff	ORPO	NA		
GLDY oprx16,xysp		IDX2	18 ED xb ee ff	OfRPP	NA		
GLDY [D,xysp]		[D,IDX]	18 ED xb	OfIfRPf	NA		
GLDY [oprx16,xysp]		[IDX2]	18 ED xb ee ff	OfIPRPf	NA		
GSTAA oprx8a	(A) ⇒ G(M)	DIR	18 5A dd	OPw	NA	– – – –	Δ Δ 0 –
GSTAA oprx16a	Store Accumulator A to Global Memory	EXT	18 7A hh ll	OPwO	NA		
GSTAA oprx0_xysp		IDX	18 6A xb	OPw	NA		
GSTAA oprx9,xysp		IDX1	18 6A xb ff	OPwO	NA		
GSTAA oprx16,xysp		IDX2	18 6A xb ee ff	OPwP	NA		
GSTAA [D,xysp]		[D,IDX]	18 6A xb	OPIfw	NA		
GSTAA [oprx16,xysp]		[IDX2]	18 6A xb ee ff	OPIPw	NA		
GSTAB oprx8a	(B) ⇒ G(M)	DIR	18 5B dd	OPw	NA	– – – –	Δ Δ 0 –
GSTAB oprx16a	Store Accumulator B to Global Memory	EXT	18 7B hh ll	OPwO	NA		
GSTAB oprx0_xysp		IDX	18 6B xb	OPw	NA		
GSTAB oprx9,xysp		IDX1	18 6B xb ff	OPwO	NA		
GSTAB oprx16,xysp		IDX2	18 6B xb ee ff	OPwP	NA		
GSTAB [D,xysp]		[D,IDX]	18 6B xb	OPIfw	NA		
GSTAB [oprx16,xysp]		[IDX2]	18 6B xb ee ff	OPIPw	NA		
GSTD oprx8a	(A) ⇒ G(M), (B) ⇒ G(M+1)	DIR	18 5C dd	OPW	NA	– – – –	Δ Δ 0 –
GSTD oprx16a	Store Double Accumulator to Global Memory	EXT	18 7C hh ll	OPWO	NA		
GSTD oprx0_xysp		IDX	18 6C xb	OPW	NA		
GSTD oprx9,xysp		IDX1	18 6C xb ff	OPWO	NA		
GSTD oprx16,xysp		IDX2	18 6C xb ee ff	OPWP	NA		
GSTD [D,xysp]		[D,IDX]	18 6C xb	OPIfW	NA		
GSTD [oprx16,xysp]		[IDX2]	18 6C xb ee ff	OPIPW	NA		
GSTS oprx8a	(SP) ⇒ G(M:M+1)	DIR	18 5F dd	OPW	NA	– – – –	Δ Δ 0 –
GSTS oprx16a	Store Stack Pointer to Global Memory	EXT	18 7F hh ll	OPWO	NA		
GSTS oprx0_xysp		IDX	18 6F xb	OPW	NA		
GSTS oprx9,xysp		IDX1	18 6F xb ff	OPWO	NA		
GSTS oprx16,xysp		IDX2	18 6F xb ee ff	OPWP	NA		
GSTS [D,xysp]		[D,IDX]	18 6F xb	OPIfW	NA		
GSTS [oprx16,xysp]		[IDX2]	18 6F xb ee ff	OPIPW	NA		

Table A-1. Instruction Set Summary (Sheet 10 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
GSTX opr8a **GSTX opr16a** **GSTX oprx0_xysp** **GSTX oprx9,xysp** **GSTX oprx16,xysp** **GSTX [D,xysp]** **GSTX [oprx16,xysp]**	**(X) ⇒ G(M:M+1)** **Store Index Register X to Global Memory**	**DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 5E dd** **18 7E hh ll** **18 6E xb** **18 6E xb ff** **18 6E xb ee ff** **18 6E xb** **18 6E xb ee ff**	**OPW** **OPWO** **OPW** **OPWO** **OPWP** **OPIfW** **OPIPW**	**NA** **NA** **NA** **NA** **NA** **NA** **NA**	-----	ΔΔ0–
GSTY opr8a **GSTY opr16a** **GSTY oprx0_xysp** **GSTY oprx9,xysp** **GSTY oprx16,xysp** **GSTY [D,xysp]** **GSTY [oprx16,xysp]**	**(Y) ⇒ G(M:M+1)** **Store Index Register Yto Global Memory**	**DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 5D dd** **18 7D hh ll** **18 6D xb** **18 6D xb ff** **18 6D xb ee ff** **18 6D xb** **18 6D xb ee ff**	**OPW** **OPWO** **OPW** **OPWO** **OPWP** **OPIfW** **OPIPW**	**NA** **NA** **NA** **NA** **NA** **NA** **NA**	-----	ΔΔ0–
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 (branch) PPO (no branch)	-----	----
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 (branch) PPO (no branch)	-----	----
IDIV	(D) ÷ (X) ⇒ X; Remainder ⇒ D 16 by 16 Bit Integer Divide (unsigned)	INH	18 10	OfffffffffO	OfffffffffO	-----	–Δ0Δ
IDIVS	(D) ÷ (X) ⇒ X; Remainder ⇒ D 16 by 16 Bit Integer Divide (signed)	INH	18 15	OfffffffffO	OfffffffffO	-----	ΔΔΔΔ
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 ⇒ AIncrement Acc. A (B) + $01 ⇒ BIncrement 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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	-----	ΔΔΔ–
INCW opr16a **INCW oprx0_xysp** **INCW oprx9,xysp** **INCW oprx16,xysp** **INCW [D,xysp]** **INCW [oprx16,xysp]** **INCX** **INCY**	**(M:M+1) + $01 ⇒ M:M+1** **Increment Memory** **(X) + $01 ⇒ XIncrement Index Register X** **(Y) + $01 ⇒ YIncrement Index Register Y**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 72 hh ll** **18 62 xb** **18 62 xb ff** **18 62 xb ee ff** **18 62 xb** **18 62 xb ee ff** **18 42** **18 52**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	-----	ΔΔΔ–
INS	(SP) + $0001 ⇒ SP *Translates to* LEAS 1,SP	IDX	1B 81	Pf	Pf	-----	----
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	SPPP SPPP 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]	-----	----

Table A-1. Instruction Set Summary (Sheet 11 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
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]	----	----
Notes: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.							
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 LBCS*	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	(M) ⇒ A	IMM	86 ii	P	P	----	ΔΔ0–
LDAA opr8a	Load Accumulator A	DIR	96 dd	rPf	rPf		
LDAA opr16a		EXT	B6 hh ll	rPO	rPO		
LDAA oprx0_xysp		IDX	A6 xb	rPf	rPf		
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	fIfrPf		
LDAA [oprx16,xysp]		[IDX2]	A6 xb ee ff	fIPrPf	fIPrPf		
LDAB #opr8i	(M) ⇒ B	IMM	C6 ii	P	P	----	ΔΔ0–
LDAB opr8a	Load Accumulator B	DIR	D6 dd	rPf	rPf		
LDAB opr16a		EXT	F6 hh ll	rPO	rPO		
LDAB oprx0_xysp		IDX	E6 xb	rPf	rPf		
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	fIfrPf		
LDAB [oprx16,xysp]		[IDX2]	E6 xb ee ff	fIPrPf	fIPrPf		
LDD #opr16i	(M:M+1) ⇒ A:B	IMM	CC jj kk	PO	PO	----	ΔΔ0–
LDD opr8a	Load Double Accumulator D (A:B)	DIR	DC dd	RPf	RPf		
LDD opr16a		EXT	FC hh ll	RPO	RPO		
LDD oprx0_xysp		IDX	EC xb	RPf	RPf		
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	fIfRPf		
LDD [oprx16,xysp]		[IDX2]	EC xb ee ff	fIPRPf	fIPRPf		
LDS #opr16i	(M:M+1) ⇒ SP	IMM	CF jj kk	PO	PO	----	ΔΔ0–
LDS opr8a	Load Stack Pointer	DIR	DF dd	RPf	RPf		
LDS opr16a		EXT	FF hh ll	RPO	RPO		
LDS oprx0_xysp		IDX	EF xb	RPf	RPf		
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	fIfRPf		
LDS [oprx16,xysp]		[IDX2]	EF xb ee ff	fIPRPf	fIPRPf		

Table A-1. Instruction Set Summary (Sheet 12 of 21)

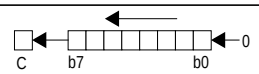
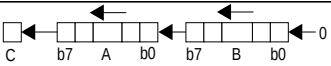
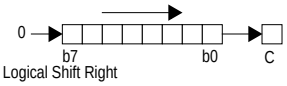
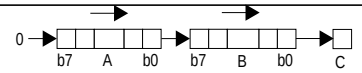
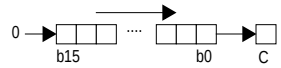
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
LDX #opr16i	(M:M+1) ⇒ X	IMM	CE jj kk	PO	PO	----	ΔΔ0-
LDX opr8a	Load Index Register X	DIR	DE dd	RPf	RPf		
LDX opr16a		EXT	FE hh ll	RPO	RPO		
LDX oprx0_xysp		IDX	EE xb	RPf	RPf		
LDX oprx9,xysp		IDX1	EE xb ff	RPO	RPO		
LDX oprx16,xysp		IDX2	EE xb ee ff	fRPP	fRPP		
LDX [D,xysp]		[D,IDX]	EE xb	fIfRPf	fIfRPf		
LDX [oprx16,xysp]		[IDX2]	EE xb ee ff	fIPRPf	fIPRPf		
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.							
LDY #opr16i	(M:M+1) ⇒ Y	IMM	CD jj kk	PO	PO	----	ΔΔ0-
LDY opr8a	Load Index Register Y	DIR	DD dd	RPf	RPf		
LDY opr16a		EXT	FD hh ll	RPO	RPO		
LDY oprx0_xysp		IDX	ED xb	RPf	RPf		
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	fIfRPf		
LDY [oprx16,xysp]		[IDX2]	ED xb ee ff	fIPRPf	fIPRPf		
LEAS oprx0_xysp	Effective Address ⇒ SP	IDX	1B xb	Pf	Pf	----	----
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	Pf	----	----
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	Pf	----	----
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	rPwO	----	Δ Δ Δ Δ
LSL oprx0_xysp		IDX	68 xb	rPw	rPw		
LSL oprx9,xysp		IDX1	68 xb ff	rPwO	rPwO		
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	 Logical Shift Left D Accumulator same function as ASLD	INH	59	O	O	----	Δ Δ Δ Δ
LSR opr16a		EXT	74 hh ll	rPwO	rPwO	----	0 Δ Δ Δ
LSR oprx0_xysp		IDX	64 xb	rPw	rPw		
LSR oprx9,xysp		IDX1	64 xb ff	rPwO	rPwO		
LSR oprx16,xysp	Logical Shift Right	IDX2	64 xb ee ff	frPwP	frPwP		
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	 Logical Shift Right D Accumulator	INH	49	O	O	----	0 Δ Δ Δ
LSRW opr16a		**EXT**	**18 74 hh ll**	**ORPWO**	**NA**	----	0 Δ Δ Δ
LSRW oprx0_xysp		**IDX**	**18 64 xb**	**ORPW**	**NA**		
LSRW oprx9,xysp		**IDX1**	**18 64 xb ff**	**ORPWO**	**NA**		
LSRW oprx16,xysp		**IDX2**	**18 64 xb ee ff**	**OfRPWP**	**NA**		
LSRW [D,xysp]		**[D,IDX]**	**18 64 xb**	**OfIfRPW**	**NA**		
LSRW [oprx16,xysp]		**[IDX2]**	**18 64 xb ee ff**	**OfIPRPW**	**NA**		
LSRX	**Logical Shift Index Register X to Right**	**INH**	**18 44**	**OO**	**NA**		
LSRY	**Logical Shift Index Register Y to Right**	**INH**	**18 54**	**OO**	**NA**		
MAXA oprx0_xysp	MAX((A), (M)) ⇒ A	IDX	18 18 xb	OrPf	OrPf	----	Δ Δ Δ Δ
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	OfIfrPf		
MAXA [oprx16,xysp]	internal compare ((A) – (M)).	[IDX2]	18 18 xb ee ff	OfIPrPf	OfIPrPf		

Table A-1. Instruction Set Summary (Sheet 13 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
MAXM oprx0_xysp MAXM oprx9,xysp MAXM oprx16,xysp MAXM [D,xysp] MAXM [oprx16,xysp]	MAX((A), (M)) ⇒ M MAX 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 1C xb 18 1C xb ff 18 1C xb ee ff 18 1C xb 18 1C xb ee ff	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	– – – –	Δ Δ Δ Δ
MEM	m (grade) fi $M_{(Y)}$; (X) + 4 ⇒ X; (Y) + 1 ⇒ Y; A unchanged if (A) < P1 or (A) > P2 then m = 0, else m = 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	– – ? –	? ? ? ?
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	OrPf OrPO OfrPP OfIfrPf OfIPrPf	– – – –	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 #opr8i, opr16a[1] **MOVB #opr8i, oprx0_xysp[1]** **MOVB #opr8i, oprx9_xysp[1]** **MOVB #opr8i, oprx16_xysp[1]** **MOVB #opr8i, [D_xysp][1]** **MOVB #opr8i, [oprx16_xysp][1]**	**# ⇒ M** **Immediate to Memory Byte-Move (8-Bit)**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0B ii hh ll** **18 08 xb[2] ii** **18 08 xb[2] ff ii** **18 08 xb[2] ee ff ii** **18 08 xb[2] ii** **18 08 xb[2] ee ff ii**	PwP PwO PwP PPwO PIOw PIOwP	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –
MOVB opr16a, opr16a[1] **MOVB opr16a, oprx0_xysp[1]** **MOVB opr16a, oprx9_xysp[1]** **MOVB opr16a, oprx16_xysp[1]** **MOVB opr16a, [D_xysp][1]** **MOVB opr16a, [oprx16_xysp][1]**	**(M_1) ⇒ M_2** **Memory to Memory Byte-Move (8-Bit)** **EXT Source fi Addr. Mode Destination**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0C hh ll hh ll** **18 09 xb[2] hh ll** **18 09 xb[2] ff hh ll** **18 09 xb[2] ee ff hh ll** **18 09 xb[2] hh ll** **18 09 xb[2] ee ff hh ll**	PrPwO PrPw PrPwO PPrPw PrIPw PPrIPw	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –
MOVB oprx0_xysp, opr16a[1] **MOVB oprx0_xysp, oprx0_xysp[1]** **MOVB oprx0_xysp, oprx9_xysp[1]** **MOVB oprx0_xysp, oprx16_xysp[1]** **MOVB oprx0_xysp, [D_xysp][1]** **MOVB oprx0_xysp, [oprx16_xysp][1]**	**(M_1) ⇒ M_2** **Memory to Memory Byte-Move (8-Bit)** **IDX Source fi Addr. Mode Destination**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0D xb hh ll** **18 0A xb xb** **18 0A xbxb ff** **18 0A xb xb ee ff** **18 0A xb xb** **18 0A xb xb ee ff**	rPPw rPOw rPPw rPOPw rPIOw rPPIOw	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –
MOVB oprx9_xysp, opr16a[1] **MOVB oprx9_xysp, oprx0_xysp[1]** **MOVB oprx9_xysp, oprx9_xysp[1]** **MOVB oprx9_xysp, oprx16_xysp[1]** **MOVB oprx9_xysp, [D_xysp][1]** **MOVB oprx9_xysp, [oprx16_xysp][1]**	**(M_1) ⇒ M_2** **Memory to Memory Byte-Move (8-Bit),** **IDX1 Source fi Addr. Mode Destination**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0D xb ff hh ll** **18 0A xb ff xb** **18 0A xb ff xb ff** **18 0A xb ff xb ee ff** **18 0A xb ff xb** **18 0A xb ff xb ee ff**	PrOPw PrOOw PrOPw PrOOPw PrOIOw PrOPIOw	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –
MOVB oprx16_xysp, opr16a[1] **MOVB oprx16_xysp, oprx0_xysp[1]** **MOVB oprx16_xysp, oprx9_xysp[1]** **MOVB oprx16_xysp, oprx16_xysp[1]** **MOVB oprx16_xysp, [D_xysp][1]** **MOVB oprx16_xysp, [oprx16_xysp][1]**	**(M_1) ⇒ M_2** **Memory to Memory Byte-Move (8-Bit),** **IDX2 Source fi Addr. Mode Destination**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0D xb ee ff hh ll** **18 0A xb ee ff xb** **18 0A xb ee ff xb ff** **18 0A xb ee ff xb ee ff** **18 0A xb ee ff xb** **18 0A xb ee ff xb ee ff**	PrPPw PrPOw PrPPw PrPOPw PrPIOw PrPPIO	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –
MOVB [D_xysp], opr16a[1] **MOVB [D_xysp], oprx0_xysp[1]** **MOVB [D_xysp], oprx9_xysp[1]** **MOVB [D_xysp], oprx16_xysp[1]** **MOVB [D_xysp], [D_xysp][1]** **MOVB [D_xysp], [oprx16_xysp][1]**	**(M_1) ⇒ M_2** **Memory to Memory Byte-Move (8-Bit),** **[D,IDX] Source fi Addr. Mode Destination**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 0D xb hh ll** **18 0A xb xb** **18 0A xb xb ff** **18 0A xb xb ee ff** **18 0A xb xb** **18 0A xb xb ee ff**	IPrfPw IPrfOw IPrfPw IPrfOPw IPrfIOw IPrfPIOw	**NA** **NA** **NA** **NA** **NA** **NA**	– – – –	– – – –

Table A-1. Instruction Set Summary (Sheet 14 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C

MOVB [oprx16_xysp], opr16a[1]
MOVB [oprx16_xysp], oprx0_xysp[1]
MOVB [oprx16_xysp], oprx9_xysp[1]
MOVB [oprx16_xysp], oprx16_xysp[1]
MOVB [oprx16_xysp], [D_xysp][1]
MOVB [oprx16_xysp], [oprx16_xysp][1]
	$(M_1) \Rightarrow M_2$
	Memory to Memory Byte-Move (8-Bit),
	[IDX2] Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 0D xb ee ff hh ll
			18 0A xb ee ff xb
			18 0A xb ee ff xb ff
			18 0A xb ee ff xb ee ff
			18 0A xb ee ff xb
			18 0A xb ee ff xb ee ff
				PIPrfPw
				PIPrfOw
				PIPrfPw
				PIPrfOPw
				PIPrfIOw
				PIPrfPIOw
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

Notes:1. The first operand in the source code statement specifies the source for the move. 2. The IDX destination code is listed before the source for backwards compatibility.							

MOVW #opr16i, opr16a[1]
MOVW #opr16i, oprx0_xysp[1]
MOVW #opr16i, oprx9_xysp[1]
MOVW #opr16i, oprx16_xysp[1]
MOVW #opr16i, [D_xysp][1]
MOVW #opr16i, [oprx16_xysp][1]
	$\# \Rightarrow M{:}M{+}1_2$
	Immediate to Memory Word-Move (16-Bit)
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 03 jj kk hh ll
			18 00 xb[2] jj kk
			18 00 xb[2] ff jj kk
			18 00 xb[2] ee ff jj kk
			18 00 xb[2] jj kk
			18 00 xb[2] ee ff jj kk
				PWPO
				PWP
				PPWO
				PPWP
				PIPW
				PIPWP
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW opr16a, opr16a[1]
MOVW opr16a, oprx0_xysp[1]
MOVW opr16a, oprx9_xysp[1]
MOVW opr16a, oprx16_xysp[1]
MOVW opr16a, [D_xysp][1]
MOVW opr16a, [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	EXT Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 04 hh ll hh ll
			18 01 xb[2] hh ll
			18 01 xb[2] ff hh ll
			18 01 xb[2] ee ff hh ll
			18 01 xb[2] hh ll
			18 01 xb[2] ee ff hh ll
				PRPWO
				PRPW
				PRPWO
				PPRPW
				PRIPW
				PPRIPW
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW oprx0_xysp, opr16a[1]
MOVW oprx0_xysp, oprx0_xysp[1]
MOVW oprx0_xysp, oprx9_xysp[1]
MOVW oprx0_xysp, oprx16_xysp[1]
MOVW oprx0_xysp, [D_xysp][1]
MOVW oprx0_xysp, [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	IDX Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 05 xb hh ll
			18 02 xb xb
			18 02 xb xb ff
			18 02 xb xb ee ff
			18 02 xb xb
			18 02 xb xb ee ff
				RPPW
				RPOW
				RPPW
				RPOPW
				RPIOW
				RPPIOW
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW oprx9_xysp, opr16a[1]
MOVW oprx9_xysp, oprx0_xysp[1]
MOVW oprx9_xysp, oprx9_xysp[1]
MOVW oprx9_xysp, oprx16_xysp[1]
MOVW oprx9_xysp, [D_xysp][1]
MOVW oprx9_xysp, [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	IDX1 Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 05 xb ff hh ll
			18 02 xb ff xb
			18 02 xb ff xb ff
			18 02 xb ff xb ee ff
			18 02 xb ff xb
			18 02 xb ff xb ee ff
				PROPW
				PROOW
				PROPW
				PROOPW
				PROIOW
				PROPIOW
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW oprx16_xysp, opr16a[1]
MOVW oprx16_xysp, oprx0_xysp[1]
MOVW oprx16_xysp, oprx9_xysp[1]
MOVW oprx16_xysp, oprx16_xysp[1]
MOVW oprx16_xysp, [D_xysp][1]
MOVW oprx16_xysp, [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	IDX2 Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 05 xb ee ff hh ll
			18 02 xb ee ff xb
			18 02 xb ee ff xb ff
			18 02 xb ee ff xb ee ff
			18 02 xb ee ff xb
			18 02 xb ee ff xb ee ff
				PRPPW
				PRPOW
				PRPPW
				PRPOPW
				PRPIOW
				PRPPIO
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW [D_xysp], opr16a[1]
MOVW [D_xysp], oprx0_xysp[1]
MOVW [D_xysp], oprx9_xysp[1]
MOVW [D_xysp], oprx16_xysp[1]
MOVW [D_xysp], [D_xysp][1]
MOVW [D_xysp], [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	[D,IDX] Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 05 xb hh ll
			18 02 xb xb
			18 02 xb xb ff
			18 02 xb xb ee ff
			18 02 xb xb
			18 02 xb xb ee ff
				IPRfPW
				IPRfOW
				IPRfPW
				IPRfOPW
				IPRfIOW
				IPRfPIOW
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MOVW [oprx16_xysp], opr16a[1]
MOVW [oprx16_xysp], oprx0_xysp[1]
MOVW [oprx16_xysp], oprx9_xysp[1]
MOVW [oprx16_xysp], oprx16_xysp[1]
MOVW [oprx16_xysp], [D_xysp][1]
MOVW [oprx16_xysp], [oprx16_xysp][1]
	$(M{:}M{+}1_1) \Rightarrow M{:}M{+}1_2$
	Memory to Memory Word-Move (16-Bit),
	[IDX2] Source fi Addr. Mode Destination
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			18 05 xb ee ff hh ll
			18 02 xb ee ff xb
			18 02 xb ee ff xb ff
			18 02 xb ee ff xb ee ff
			18 02 xb ee ff xb
			18 02 xb ee ff xb ee ff
				PIPRfPW
				PIPRfOW
				PIPRfPW
				PIPRfOPW
				PIPRfIOW
				PIPRfPIOW
					NA
					NA
					NA
					NA
					NA
					NA
						----	----

MUL	$(A) \times (B) \Rightarrow A{:}B$ 8 by 8 Unsigned Multiply	INH	12	0	0	----	---Δ
NEG opr16a NEG oprx0_xysp NEG oprx9,xysp NEG oprx16,xysp NEG [D,xysp] NEG [oprx16,xysp]	$0 - (M) \Rightarrow M$ *equivalent to* $(\overline{M}) + 1 \Rightarrow M$ Two's Complement Negate	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	70 hh ll 60 xb 60 xb ff 60 xb ee ff 60 xb 60 xb ee ff	rPwO rPw rPwO frPwP fIfrPw fIPrPw	rPwO rPw rPwO frPwP fIfrPw fIPrPw	----	$\Delta\ \Delta\ \Delta\ \Delta$
NEGA	$0 - (A) \Rightarrow A$ *equivalent to* $(\overline{A}) + 1 \Rightarrow A$ Negate Accumulator A	INH	40	0	0		
NEGB	$0 - (B) \Rightarrow B$ *equivalent to* $(\overline{B}) + 1 \Rightarrow B$ Negate Accumulator B	INH	50	0	0		

Table A-1. Instruction Set Summary (Sheet 15 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
NEGW opr16a	**0–(M:M+1)⇒M:M+1 equivalent to ($\overline{M:M+1}$) +1⇒M:M+1**	**EXT**	**18 70 hh ll**	**ORPWO**	**NA**	– – – –	Δ Δ Δ Δ
NEGW oprx0_xysp	**Two's Complement Negate**	**IDX**	**18 60 xb**	**ORPW**	**NA**		
NEGW oprx9,xysp		**IDX1**	**18 60 xb ff**	**ORPWO**	**NA**		
NEGW oprx16,xysp		**IDX2**	**18 60 xb ee ff**	**OfRPWP**	**NA**		
NEGW [D,xysp]		**[D,IDX]**	**18 60 xb**	**OfIfRPW**	**NA**		
NEGW [oprx16,xysp]		**[IDX2]**	**18 60 xb ee ff**	**OfIPRPW**	**NA**		
NEGX	**0 – (X) ⇒ X equivalent to ($\overline{X}$) + 1 ⇒ X** **Negate Index Register X**	**INH**	**18 40**	**OO**	**NA**		
NEGY	**0 – (Y) ⇒ Yequivalent to ($\overline{Y}$) + 1 ⇒ Y** **Negate Index Register Y**	**INH**	**18 50**	**OO**	**NA**		
NOP	No Operation	INH	A7	O	O	– – – –	– – – –
ORAA #opr8i	(A) | (M) ⇒ A	IMM	8A ii	P	P	– – – –	Δ Δ 0 –
ORAA opr8a	Logical OR A with Memory	DIR	9A dd	rPf	rPf		
ORAA opr16a		EXT	BA hh ll	rPO	rPO		
ORAA oprx0_xysp		IDX	AA xb	rPf	rPf		
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	fIfrPf		
ORAA [oprx16,xysp]		[IDX2]	AA xb ee ff	fIPrPf	fIPrPf		
ORAB #opr8i	(B) | (M) ⇒ B	IMM	CA ii	P	P	– – – –	Δ Δ 0 –
ORAB opr8a	Logical OR B with Memory	DIR	DA dd	rPf	rPf		
ORAB opr16a		EXT	FA hh ll	rPO	rPO		
ORAB oprx0_xysp		IDX	EA xb	rPf	rPf		
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	fIfrPf		
ORAB [oprx16,xysp]		[IDX2]	EA xb ee ff	fIPrPf	fIPrPf		
ORCC #opr8i	(CCR) | M ⇒ CCR Logical OR CCR with Memory	IMM	14 ii	P	P	⇑ – ⇑ ⇑	⇑ ⇑ ⇑ ⇑
ORX #opr16i	**(X) | (M:M+1) ⇒ X**	**IMM**	**18 8A jj kk**	**OPO**	**NA**	– – – –	Δ Δ 0 –
ORX opr8a	**Logical OR X with Memory**	**DIR**	**18 9A dd**	**ORPf**	**NA**		
ORX opr16a		**EXT**	**18 BA hh ll**	**ORPO**	**NA**		
ORX oprx0_xysp		**IDX**	**18 AA xb**	**ORPf**	**NA**		
ORX oprx9,xysp		**IDX1**	**18 AA xb ff**	**ORPO**	**NA**		
ORX oprx16,xysp		**IDX2**	**18 AA xb ee ff**	**OfRPP**	**NA**		
ORX [D,xysp]		**[D,IDX]**	**18 AA xb**	**OfIfRPf**	**NA**		
ORX [oprx16,xysp]		**[IDX2]**	**18 AA xb ee ff**	**OfIPRPf**	**NA**		
ORY #opr16i	**(Y) | (M:M+1) ⇒ Y**	**IMM**	**18 CA jj kk**	**OP**	**NA**	– – – –	Δ Δ 0 –
ORY opr8a	**Logical OR Y with Memory**	**DIR**	**18 DA dd**	**ORPf**	**NA**		
ORY opr16a		**EXT**	**18 FA hh ll**	**ORPO**	**NA**		
ORY oprx0_xysp		**IDX**	**18 EA xb**	**ORPf**	**NA**		
ORY oprx9,xysp		**IDX1**	**18 EA xb ff**	**ORPO**	**NA**		
ORY oprx16,xysp		**IDX2**	**18 EA xb ee ff**	**OfRPP**	**NA**		
ORY [D,xysp]		**[D,IDX]**	**18 EA xb**	**OfIfRPf**	**NA**		
ORY [oprx16,xysp]		**[IDX2]**	**18 EA xb ee ff**	**OfIPRPf**	**NA**		
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	– – – –	– – – –
PSHCW	**(SP) – 2 ⇒ SP; (CCR_H:CCR_L) ⇒ $M_{(SP)}$:$M_{(SP+1)}$** **Push CCR onto Stack**	**INH**	**18 39**	**OOS**	**NA**	– – – –	– – – –
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	– – – –	– – – –
PULA	($M_{(SP)}$) ⇒ A; (SP) + 1 ⇒ SP Pull Accumulator A from Stack	INH	32	ufO	ufO	– – – –	– – – –
PULB	($M_{(SP)}$) ⇒ B; (SP) + 1 ⇒ SP Pull Accumulator B from Stack	INH	33	ufO	ufO	– – – –	– – – –

Notes1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

Table A-1. Instruction Set Summary (Sheet 16 of 21)

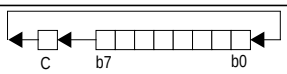
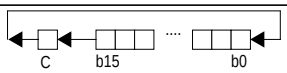
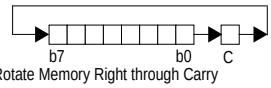
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
PULC	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ Pull CCR from Stack	INH	38	ufO	ufO	Δ fl Δ Δ	Δ Δ Δ Δ
PULCW	**$(M_{(SP)}:M_{(SP+1)}) \Rightarrow CCR_H:CCR_L; (SP) + 2 \Rightarrow SP$** **Pull CCR from Stack**	**INH**	**18 38**	**OUfO**	**NA**	Δ fl 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) ffff + 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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	----	Δ Δ Δ Δ
ROLW oprx0_xysp **ROLW oprx9,xysp** **ROLW oprx16,xysp** **ROLW [D,xysp]** **ROLW [oprx16,xysp]** **ROLX** **ROLY**	 **Rotate Memory Left through Carry** **Rotate XLeft through Carry** **Rotate YLeft through Carry**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 75 hh ll** **18 65 xb** **18 65 xb ff** **18 65 xb ee ff** **18 65 xb** **18 65 xb ee ff** **18 45** **18 55**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **fOIPRPW** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	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	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	----	Δ Δ Δ Δ

Table A-1. Instruction Set Summary (Sheet 17 of 21)

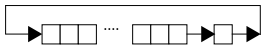
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
RORW oprx16a **RORW oprx0_xysp** **RORW oprx9,xysp** **RORW oprx16,xysp** **RORW [D,xysp]** **RORW [oprx16,xysp]**	 **Rotate Memory Right through Carry**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 76 hh ll** **18 66 xb** **18 66 xb ff** **18 66 xb ee ff** **18 66 xb** **18 66 xb ee ff**	**ORPWO** **ORPW** **ORPWO** **OfRPWP** **OfIfRPW** **OfIPRPW**	**NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
RORX **RORY**	**Rotate X Right through Carry** **Rotate Y Right through Carry**	**INH** **INH**	**18 46** **18 56**	**OO** **OO**	**NA** **NA**		
RTC	($M_{(SP)}$) ⇒ PPAGE; (SP) + 1 ⇒ SP; ($M_{(SP)}$:$M_{(SP+1)}$) ⇒ PC_H:PC_L; (SP) + 2 ⇒ SP Return from Call	INH	0A	uUnfPPP	uUnfPPP	----	----
RTI	($M_{(SP)}$) ⇒ CCR; (SP) + 1 ⇒ SP ($M_{(SP)}$:$M_{(SP+1)}$) ⇒ B:A; (SP) + 2 ⇒ SP ($M_{(SP)}$:$M_{(SP+1)}$) ⇒ X_H:X_L; (SP) + 4 ⇒ SP ($M_{(SP)}$:$M_{(SP+1)}$) ⇒ PC_H:PC_L; (SP) – 2 ⇒ SP ($M_{(SP)}$:$M_{(SP+1)}$) ⇒ Y_H:Y_L; (SP) + 4 ⇒ SP Return from Interrupt	INH	0B	UUUUUPPP (with interrupt pending) UUUUUVfPPP	UUUUUPPP UUUUUVfPPP	Δ fl Δ Δ	Δ Δ Δ Δ
RTS	($M_{(SP)}$:$M_{(SP+1)}$) ⇒ PC_H:PC_L; (SP) + 2 ⇒ SP Return from Subroutine	INH	3D	UfPPP	UfPPP	----	----
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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	Δ Δ Δ Δ
SBED #opr16i **SBED opr8a** **SBED opr16a** **SBED oprx0_xysp** **SBED oprx9,xysp** **SBED oprx16,xysp** **SBED [D,xysp]** **SBED [oprx16,xysp]**	**(D) – (M:M+1) – C ⇒ D** **Subtract with Borrow from D**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 83 jj kk** **18 93 dd** **18 B3 hh ll** **18 A3 xb** **18 A3 xb ff** **18 A3 xb ee ff** **18 A3 xb** **18 A3 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
SBEX #opr16i **SBEX opr8a** **SBEX opr16a** **SBEX oprx0_xysp** **SBEX oprx9,xysp** **SBEX oprx16,xysp** **SBEX [D,xysp]** **SBEX [oprx16,xysp]**	**(X) – (M:M+1) – C ⇒ X** **Subtract with Borrow from X**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 82 jj kk** **18 92 dd** **18 B2 hh ll** **18 A2 xb** **18 A2 xb ff** **18 A2 xb ee ff** **18 A2 xb** **18 A2 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
SBEY #opr16i **SBEY opr8a** **SBEY opr16a** **SBEY oprx0_xysp** **SBEY oprx9,xysp** **SBEY oprx16,xysp** **SBEY [D,xysp]** **SBEY [oprx16,xysp]**	**(Y) – (M:M+1) – C ⇒ Y** **Subtract with Borrow from Y**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 C2 jj kk** **18 D2 dd** **18 F2 hh ll** **18 E2 xb** **18 E2 xb ff** **18 E2 xb ee ff** **18 E2 xb** **18 E2 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
SEC	1 ⇒ C *Translates to* ORCC #$01	IMM	14 01	P	P	----	---1

Table A-1. Instruction Set Summary (Sheet 18 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
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 PwO Pw PwO PwP PIfw PIPw	----	ΔΔ0-
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 PwO Pw PwO PwP PIfw PIPw	----	ΔΔ0-
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 PWO PW PWO PWP PIfW PIPW	----	ΔΔ0-
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 fi SP; (CCR) ⇒ $M_{(SP)}$; 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	(entering STOP) OOSSSSSf (exiting STOP) fVfPPP (continue) ff (if STOP disabled) OO	----	----
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 PWO PW PWO PWP PIfW PIPW	----	ΔΔ0-
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 PWO PW PWO PWP PIfW PIPW	----	ΔΔ0-
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 PWO PW PWO PWP PIfW PIPW	----	ΔΔ0-

Table A-1. Instruction Set Summary (Sheet 19 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C

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
					rPf
					rPO
					rPf
					rPO
					frPP
					fIfrPf
					fIPrPf
						----	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
					rPf
					rPO
					rPf
					rPO
					frPP
					fIfrPf
					fIPrPf
						----	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
					PO
					RPf
					RPO
					RPf
					RPO
					fRPP
					fIfRPf
					fIPRPf
						----	Δ Δ Δ Δ

SUBX #opr16i **SUBX opr8a** **SUBX opr16a** **SUBX oprx0_xysp** **SUBX oprx9,xysp** **SUBX oprx16,xysp** **SUBX [D,xysp]** **SUBX [oprx16,xysp]**	**(X) – (M:M+1) ⇒ X** **Subtract Memory from X**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 80 jj kk** **18 90 dd** **18 B0 hh ll** **18 A0 xb** **18 A0 xb ff** **18 A0 xb ee ff** **18 A0 xb** **18 A0 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
SUBY #opr16i **SUBY opr8a** **SUBY opr16a** **SUBY oprx0_xysp** **SUBY oprx9,xysp** **SUBY oprx16,xysp** **SUBY [D,xysp]** **SUBY [oprx16,xysp]**	**(Y) – (M:M+1) ⇒ Y** **Subtract Memory from Y**	**IMM** **DIR** **EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]**	**18 C0 jj kk** **18 D0 dd** **18 F0 hh ll** **18 E0 xb** **18 E0 xb ff** **18 E0 xb ee ff** **18 E0 xb** **18 E0 xb ee ff**	**OPO** **ORPf** **ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	----	Δ Δ Δ Δ
SWI	(SP) – $0002 ⇒ SP; RTN_H : RTN_L ⇒ ($M_{(SP)}$: $M_{(SP+1)}$) (SP) – $0002 ⇒ SP; Y_H : Y_L ⇒ ($M_{(SP)}$: $M_{(SP+1)}$) (SP) – $0002 ⇒ SP; X_H : X_L ⇒ ($M_{(SP)}$: $M_{(SP+1)}$) (SP) – $0002 ⇒ SP; B : A⇒ ($M_{(SP)}$: $M_{(SP+1)}$) (SP) – $0002 ⇒ SP; CCR_H : CCR_L ⇒ ($M_{(SP)}$: $M_{(SP+1)}$) 1 ⇒ I; (SWI Vector) ⇒ PC Software Interrupt	INH	3F	VSPSSPSSP* (for Reset) VfPPP	VSPSSPSSP* VfPPP	---1 11-1	---- ----
*The CPU12 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 fl 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 (branch) PPO (no branch)	----	----

Table A-1. Instruction Set Summary (Sheet 20 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X	Access Detail HCS12	S X H I	N Z V C
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	ORfffP	– – – –	Δ Δ – Δ
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 (branch) PPO (no branch)	– – – –	– – – –
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 fl Δ Δ	– – – – or 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	OVSPSSPSSP	– – – 1	– – – –
TST opr16a TST oprx0_xysp TST oprx9,xysp TST oprx16,xysp TST [D,xysp] TST [oprx16,xysp] TSTA TSTB	(M) – 0 Test Memory for Zero or Minus (A) – 0Test A for Zero or Minus (B) – 0Test B for Zero or Minus	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	F7 hh ll E7 xb E7 xb ff E7 xb ee ff E7 xb E7 xb ee ff 97 D7	rPO rPf rPO frPP fIfrPf fIPrPf O O	rPO rPf rPO frPP fIfrPf fIPrPf O O	– – – –	Δ Δ 0 0
TSTW opr16a **TSTW oprx0_xysp** **TSTW oprx9,xysp** **TSTW oprx16,xysp** **TSTW [D,xysp]** **TSTW [oprx16,xysp]** **TSTX** **TSTY**	**(M:M+1) – 0** **Test Memory for Zero or Minus** **(X) – 0Test X for Zero or Minus** **(Y) – 0Test Yfor Zero or Minus**	**EXT** **IDX** **IDX1** **IDX2** **[D,IDX]** **[IDX2]** **INH** **INH**	**18 F7 hh ll** **18 E7 xb** **18 E7 xb ff** **18 E7 xb ee ff** **18 E7 xb** **18 E7 xb ee ff** **18 97** **18 D7**	**ORPO** **ORPf** **ORPO** **OfRPP** **OfIfRPf** **OfIPRPf** **OO** **OO**	**NA** **NA** **NA** **NA** **NA** **NA** **NA** **NA**	– – – –	Δ Δ 0 0
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	OSSSSsf (after interrupt) fVfPPP	– – – – or – – – 1 or – 1 – 1	– – – – or – – – – or – – – –

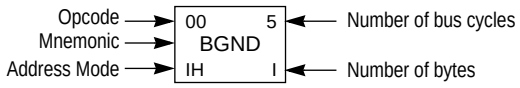
Table A-1. Instruction Set Summary (Sheet 21 of 21)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail HCS12X / HCS12	S X H I	N Z V C
WAV	$\sum_{i=1}^{B} S_i F_i$ *fi* Y:D and $\sum_{i=1}^{B} F_i$ *fi* 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 Of(frr,ffff)O (add if interrupt) SSS + UUUrr, SSS + 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 UUUrr,ffff (frr,ffff)O (exit + re-entry replaces comma above if interrupted) SSS + UUUrr, SSS + 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. Opcode Map (Sheet 1 of 3) — HCS12 and HCS12X Page 1 Opcodes

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 §1 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



* The opcode $04 (on sheet 1 of 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

Page 2 When the CPU12 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.

† Refer to instruction summary for more information.

§ EMUL requires 3 cycles for HCS12.

Table A-2. Opcode Map (Sheet 2 of 3) — HCS12 Page 2 Opcodes

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 5 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	1 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 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

† Refer to instruction summary for more information.

Page 2 When the CPU12 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-2. Opcode Map (Sheet 3 of 3) — HCS12X Page 2 Opcodes

00 4-6 MOVW IM-ID 5	10 12 IDIV IH 2	20 4 LBRA RL 4	30 10 TRAP IH 2	40 2 NEGX IH 2	50 2 NEGY IH 2	60 4-7 NEGW ID 3-5	70 5 NEGW EX 4	80 3 SUBX IM 4	90 4 SUBX DI 3	A0 4-7 SUBX ID 3-5	B0 4 SUBX EX 4	C0 3 SUBY IM 4	D0 4 SUBY DI 3	E0 4-7 SUBY ID 3-5	F0 4 SUBY EX 4
01 5-7 MOVW EX-ID 5	11 12 FDIV IH 2	21 3 LBRN RL 4	31 10 TRAP IH 2	41 2 COMX IH 2	51 2 COMY IH 2	61 4-7 COMW ID 3-5	71 5 COMW EX 4	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 5-10 MOVW ID-ID 4	12 9 EMACS SP 4	22 4/3 LBHI RL 4	32 10 TRAP IH 2	42 2 INCX IH 2	52 2 INCY IH 2	62 4-7 INCW ID 3-5	72 5 INCW EX 4	82 3 SBEX IM 4	92 4 SBEX DI 3	A2 4-7 SBEX ID 3-5	B2 4 SBEX EX 4	C2 3 SBEY IM 4	D2 4 SBEY DI 3	E2 4-7 SBEY ID 3-5	F2 4 SBEY EX 4
03 5 MOVW IM-EX 6	13 3 EMULS IH 2	23 4/3 LBLS RL 4	33 10 TRAP IH 2	43 2 DECX IH 2	53 2 DECY IH 2	63 4-7 DECW ID 3-5	73 5 DECW EX 4	83 3 SBED IM 4	93 4 SBED DI 3	A3 4-7 SBED ID 3-5	B3 4 SBED EX 4	C3 3 ADED IM 4	D3 4 ADED DI 3	E3 4-7 ADED ID 3-5	F3 4 ADED EX 4
04 6 MOVW EX-EX 6	14 12 EDIVS IH 2	24 4/3 LBCC RL 4	34 10 TRAP IH 2	44 2 LSRX IH 2	54 2 LSRY IH 2	64 4-7 LSRW ID 3-5	74 5 LSRW EX 4	84 3 ANDX IM 4	94 4 ANDX DI 3	A4 4-7 ANDX ID 3-5	B4 3 ANDX EX 4	C4 3 ANDY IM 4	D4 4 ANDY DI 3	E4 4-7 ANDY ID 3-5	F4 3 ANDY EX 4
05 5-8 MOVW ID-EX 5	15 12 IDIVS IH 2	25 4/3 LBCS RL 4	35 5 BTAS DI 4	45 2 ROLX IH 2	55 2 ROLY IH 2	65 4-7 ROLW ID 3-5	75 5 ROLW EX 4	85 3 BITX IM 4	95 4 BITX DI 3	A5 4-7 BITX ID 3-5	B5 3 BITX EX 4	C5 3 BITY IM 4	D5 4 BITY DI 3	E5 4-7 BITY ID 3-5	F5 3 BITY EX 4
06 2 ABA IH 2	16 2 SBA IH 2	26 4/3 LBNE RL 4	36 6 BTAS EX 5	46 2 RORX IH 2	56 2 RORY IH 2	66 4-7 RORW ID 3-5	76 5 RORW EX 4	86 10 TRAP IH 2	96 4 GLDAA DI 3	A6 4-7 GLDAA ID 3-5	B6 4 GLDAA EX 4	C6 10 TRAP IH 2	D6 4 GLDAB DI 3	E6 4-7 GLDAB ID 3-5	F6 4 GLDAB EX 4
07 3 DAA IH 2	17 2 CBA IH 2	27 4/3 LBEQ RL 4	37 5-7 BTAS ID 4-6	47 2 ASRX IH 2	57 2 ASRY IH 2	67 4-7 ASRW ID 3-5	77 5 ASRW EX 4	87 2 CLRX IH 2	97 2 TSTX IH 2	A7 10 TRAP IH 2	B7 10 TRAP IH 2	C7 2 CLRY IH 2	D7 2 TSTY IH 2	E7 4-7 TSTW ID 3-5	F7 4 TSTW EX 4
08 4-6 MOVB IM-ID 4	18 4-7 MAXA ID 3-5	28 4/3 LBVC RL 4	38 4 PULCW IH 2	48 2 ASLX IH 2	58 2 ASLY IH 2	68 4-7 ASLW ID 3-5	78 5 ASLW EX 4	88 3 EORX IM 4	98 4 EORX DI 3	A8 4-7 EORX ID 3-5	B8 3 EORX EX 4	C8 3 EORY IM 4	D8 4 EORY DI 3	E8 4-7 EORY ID 3-5	F8 3 EORY EX 4
09 5-7 MOVB EX-ID 5	19 4-7 MINA ID 3-5	29 4/3 LBVS RL 4	39 3 PSHCW IH 2	49 10 TRAP IH 2	59 10 TRAP IH 2	69 4-7 CLRW ID 3-5	79 5 CLRW EX 4	89 3 ADEX IM 4	99 4 ADEX DI 3	A9 4-7 ADEX ID 3-5	B9 4 ADEX EX 4	C9 3 ADEY IM 4	D9 4 ADEY DI 3	E9 4-7 ADEY ID 3-5	F9 4 ADEY EX 4
0A 5-10 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 3 GSTAA DI 3	6A 3-5 GSTAA ID 3-5	7A 4 GSTAA EX 4	8A 3 ORX IM 4	9A 4 ORX DI 3	AA 4-7 ORX ID 3-5	BA 3 ORX EX 4	CA 3 ORY IM 4	DA 4 ORY DI 3	EA 4-7 ORY ID 3-5	FA 3 ORY EX 4
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 3 GSTAB DI 3	6B 3-5 GSTAB ID 3-5	7B 4 GSTAB EX 4	8B 3 ADDX IM 4	9B 4 ADDX DI 3	AB 4-7 ADDX ID 3-5	BB 4 ADDX EX 4	CB 3 ADDY IM 4	DB 4 ADDY DI 3	EB 4-7 ADDY ID 3-5	FB 4 ADDY EX 4
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 3 GSTD DI 3	6C 3-5 GSTD ID 3-5	7C 4 GSTD EX 4	8C 3 CPED IM 4	9C 4 CPED DI 3	AC 4-7 CPED ID 3-5	BC 4 CPED EX 4	CC 10 TRAP IH 2	DC 4 GLDD DI 3	EC 4-7 GLDD ID 3-5	FC 4 GLDD EX 4
0D 5-8 MOVB ID-EX 5	1 D4-7 MINM ID 3-5	2D 4/3 LBLT RL 4	3D 6 TBL ID 3	4D 10 TRAP IH 2	5D 3 GSTY DI 3	6D 3-5 GSTY ID 3-5	7D 4 GSTY EX 4	8D 3 CPEY IM 4	9D 4 CPEY DI 3	AD 4-7 CPEY ID 3-5	BD 4 CPEY EX 4	CD 10 TRAP IH 2	DD 4 GLDY DI 3	ED 4-7 GLDY ID 3-5	FD 4 GLDY EX 4
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 3 GSTX DI 3	6E 3-5 GSTX ID 3-5	7E 4 GSTX EX 4	8E 3 CPEX IM 4	9E 4 CPEX DI 3	AE 4-7 CPEX ID 3-5	BE 4 CPEX EX 4	CE 10 TRAP IH 2	DE 4 GLDX DI 3	EE 4-7 GLDX ID 3-5	FE 4 GLDX EX 4
0F 2 TBA IH 2	1F 4-7 EMINM ID 3-5	2F 4/3 LBLE RL 4	3F 8 ETBL ID 3	4F 10 TRAP IH 2	5F 3 GSTS DI 3	6F 3-5 GSTS ID 3-5	7F 4 GSTS EX 4	8F 3 CPES IM 4	9F 4 CPES DI 3	AF 4-7 CPES ID 3-5	BF 4 CPES EX 4	CF 10 TRAP IH 2	DF 4 GLDS DI 3	EF 4-7 GLDS ID 3-5	FF 4 GLDS EX 4

* The opcode $04 (on sheet 1 of 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

† Refer to instruction summary for more information.

Page 2 When the CPU12 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

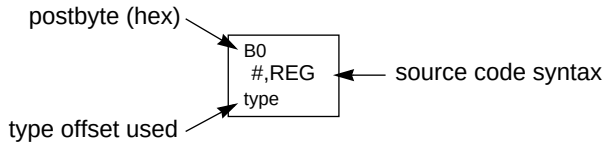


Table A-4. Indexed Addressing Mode Summary

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									
	MS⇒	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
⇓ LS		**A**	**B**	**CCR**	**TMPx**	**D**	**X**	**Y**	**SP**
0	**A**	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR_L ⇒ A TFR CCR,A TFR CCRL,A	$TMP3_L$ ⇒ A TFR TMP3,A TFR TMP3L,A	B ⇒ A TFR D,A	X_L ⇒ A TFR X, A TFR XL,A	Y_L ⇒ A TFR Y,A TFR YL,A	SP_L ⇒ A TFR SP,A TFR SPL,A
1	**B**	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR_L ⇒ B TFR CCR,B TFR CCRL,B	$TMP3_L$ ⇒ B TFR TMP3,B TFR TMP3L,B	B ⇒ B TFR D,B	X_L ⇒ B TFR X, B TFR XL,B	Y_L ⇒ B TFR Y,B TFR YL,B	SP_L ⇒ B TFR SP,B TFR SPL,B
2	**CCR**	A ⇒ CCR TFR A,CCR TFR A,CCRL	B ⇒ CCR TFR B,CCR TFR B,CCRL	CCR_L ⇒ CCR_L TFR CCR,CCR TFR CCRL,CCRL	$TMP3_L$ ⇒ CCR TFR TMP3,CCR TFR TMP3L,CCRL	B ⇒ CCR TFR D,CCR TFR D,CCRL	X_L ⇒ CCR TFR X,CCR TFR XL,CCRL	Y_L ⇒ CCR TFR Y,CCR TFR YL,CCRL	SP_L ⇒ CCR TFR SP,CCR TFR SPL,CCRL
3	**TMP2**	sex:A ⇒ TMP2 SEX A,TMP2	sex:B ⇒ TMP2 SEX B,TMP2	sex:CCR_L ⇒ TMP2 SEX CCR,TMP2 SEX CCRL,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP2 TFR D,TMP2	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
4	**D**	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR_L ⇒ D SEX CCR_L,D SEX CCRL,D	TMP3 ⇒ D TFR TMP3,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
5	**X**	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR_L ⇒ X SEX CCR,X SEX CCRL,X	TMP3 ⇒ X TFR TMP3,X	D ⇒ X TFR D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
6	**Y**	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR_L ⇒ Y SEX CCR,Y SEX CCRL,Y	TMP3 ⇒ Y TFR TMP3,Y	D ⇒ Y TFR D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
7	**SP**	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR_L ⇒ SP SEX CCR,SP SEX CCRL,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP
8	**A**	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR_H ⇒ A TFR CCRH,A	$TMP3_H$ ⇒ A TFR TMP3H,A	B ⇒ A TFR D,A	X_H ⇒ A TFR XH, A	Y_H ⇒ A TFR YH,A	SP_H ⇒ A TFR SPH,A
9	**B**	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR_L ⇒ B TFR CCRL,B	$TMP3_L$ ⇒ B TFR TMP3L,B	B ⇒ B TFR D,B	X_L ⇒ B TFR XL, B	Y_L ⇒ B TFR YL,B	SP_L ⇒ B TFR SPL,B
A	**CCR**	A ⇒ CCR_H TFR A,CCRH	B ⇒ CCR_L TFR B,CCRL	CCRW ⇒ CCRW TFR CCRW,CCRW	TMP3 ⇒ $CCR_{H:L}$ TFR TMP3,CCRW	D ⇒ $CCR_{H:L}$ TFR D,CCRW	X ⇒ $CCR_{H:L}$ TFR X,CCRW	Y ⇒ $CCR_{H:L}$ TFR Y,CCRW	SP ⇒ $CCR_{H:L}$ TFR SP,CCRW
B	**TMPx**	A ⇒ $TMP2_H$ TFR A,TMP2H	B ⇒ $TMP2_L$ TFR B,TMP2L	$CCR_{H:L}$ ⇒ TMP2 TFR CCRW,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP1 TFR D,TMP1	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
C	**D**	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	$CCR_{H:L}$ ⇒ D TFR CCRW,D	TMP1 ⇒ D TFR TMP1,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
D	**X**	A ⇒ X_H TFR A,XH	B ⇒ X_L TFR B,XL	$CCR_{H:L}$ ⇒ X TFR CCRW,X	TMP3 ⇒ X TFR TMP3,X	sex:D ⇒ X SEX D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
E	**Y**	A ⇒ Y_H TFR A,YH	B ⇒ Y_L TFR B,YL	$CCR_{H:L}$ ⇒ Y TFR CCRW,Y	TMP3 ⇒ Y TFR TMP3,Y	sex:D ⇒ Y SEX D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
F	**SP**	A ⇒ SP_H TFR A,SPH	B ⇒ SP_L TFR B,SPL	$CCR_{H:L}$ ⇒ SP TFR CCRW,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the S12X.

Table A-5. Transfer and Exchange Postbyte Encoding (Continued)

EXCHANGES									
	MS⇒	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
⇓ LS		**A**	**B**	**CCR**	**TMPx**	**D**	**X**	**Y**	**SP**
0	**A**	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR_L ⇔ A EXG CCR,A *EXG CCRL,A*	$TMP3_L$ ⇒ A $00:A ⇒ TMP3 EXG A, TMP3	B ⇔ A EXG D,A	X_L ⇒ A $00:A ⇒ X EXG X,A	Y_L ⇒ A $00:A ⇒ Y EXG Y,A	SP_L ⇒ A $00:A ⇒ SP EXG SP,A
1	**B**	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR_L ⇔ B EXG CCR,B *EXG CCRL,B*	$TMP3_L$ ⇒ B $FF:B ⇒ TMP3 EXG B,TMP3	B ⇒ B $FF ⇒ A EXG D,B	X_L ⇒ B $FF:B ⇒ X EXG X,B	Y_L ⇒ B $FF:B ⇒ Y EXG Y,B	SP_L ⇒ B $FF:B ⇒ SP EXG SP,B
2	**CCR**	A ⇔ CCR_L EXG A, CCR *EXG A,CCRL*	B ⇔ CCR_L EXG B,CCR *EXG B,CCRL*	CCR_L ⇔ CCR_L EXG CCR,CCR *EXG CCRL,CCRL*	$TMP3_L$ ⇒ CCR_L $FF:$CCR_L$ ⇒ TMP3 EXG, TMP3,CCR *EXG TMP3,CCRL*	B ⇒ CCR_L $FF:$CCR_L$ ⇒ D EXG D,CCR *EXG D,CCRL*	X_L ⇒ CCR_L $FF:$CCR_L$ ⇒ X EXG X,CCR *EXG X,CCRL*	Y_L ⇒ CCR_L $FF:$CCR_L$ ⇒ Y EXG Y,CCR *EXG Y,CCRL*	SP_L ⇒ CCR_L $FF:$CCR_L$ ⇒ SP EXG SP,CCR *EXG SP,CCRL*
3	**TMP2**	$00:A ⇒ $TMP2_H$ $TMP2_L$ ⇒ A EXG A,TMP2	$00:B ⇒ $TMP2_H$ $TMP2_L$ ⇒ B EXG B,TMP2	$00:$CCR_L$ ⇒ $TMP2_H$ $TMP2_L$ ⇒ CCR EXG CCR,TMP2	$TMP3_H$ ⇔ $TMP2_H$ EXG TMP3,TMP2	D ⇔ $TMP2_H$ EXG D,TMP2	X ⇔ $TMP2_H$ EXG X,TMP2	Y ⇔ $TMP2_H$ EXG Y,TMP2	SP ⇔ $TMP2_H$ EXG SP,TMP2
4	**D**	$00:A ⇒ D EXG A,D	$00:B ⇒ D EXG B,D	$00:$CCR_L$ ⇒ D B ⇒ CCR_L EXG CCR,D *EXG CCRL,D*	$TMP3_H$ ⇔ D EXG TMP3,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
5	**X**	$00:A ⇒ X X_L ⇒ A EXG A,X	$00:B ⇒ X X_L ⇒ B EXG B,X	$00:$CCR_L$ ⇒ X X_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	$TMP3_H$ ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
6	**Y**	$00:A ⇒ Y Y_L ⇒ A EXG A,Y	$00:B ⇒ Y Y_L ⇒ B EXG B,Y	$00:$CCR_L$ ⇒ Y Y_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	$TMP3_H$ ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
7	**SP**	$00:A ⇒ SP SP_L ⇒ A EXG A,SP	$00:B ⇒ SP SP_L ⇒ B EXG B,SP	$00:$CCR_L$ ⇒ SP SP_L ⇒ CCR_L EXG CCR,X *EXG CCRL,X*	$TMP3_H$ ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP
8	**A**	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR_H ⇔ A EXG CCRH,A	$TMP3_H$ ⇔ A EXG TMP3H,A	B ⇔ A EXG D,A	X_H ⇔ A EXG XH,A	Y_H ⇔ A EXG YH,A	SP_H ⇔ A EXG SPH,A
9	**B**	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR_L ⇔ B EXG CCRL,B	$TMP3_L$ ⇔ B EXG TMP3L,B	$FF ⇒ A, B ⇒ B EXG D,B	X_L ⇔ B EXG XL,B	Y_L ⇔ B EXG YL,B	SP_L ⇔ B EXG SPL,B
A	**CCR**	A ⇔ CCR_H EXG A,CCRH	B ⇔ CCR_L EXG B,CCRL	$CCR_{H:L}$ ⇔ $CCR_{H:L}$ EXG CCRW,CCRW	$TMP3_H$ ⇔ $CCR_{H:L}$ EXG TMP3,CCRW	D ⇔ $CCR_{H:L}$ EXG D,CCRW	X ⇔ $CCR_{H:L}$ EXG X,CCRW	Y ⇔ $CCR_{H:L}$ EXG Y,CCRW	SP ⇔ $CCR_{H:L}$ EXG, SP,CCRW
B	**TMPx**	A ⇔ $TMP2_H$ EXG A,TMP2H	B ⇔ $TMP2_L$ EXG B,TMP2L	$CCR_{H:L}$⇔ TMP2 EXG CCRW,TMP2	$TMP3_H$ ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP1 EXG D,TMP1	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
C	**D**	$00:A ⇒ D EXG A,D	$00:B ⇒ D EXG B,D	$CCR_{H:L}$ ⇔ D EXG CCRW,D	TMP1 ⇔ D EXG TMP1,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
D	**X**	A ⇔ X_H EXG A,XH	B ⇔ X_L EXG B,XL	$CCR_{H:L}$ ⇔ X EXG CCRW,X	$TMP3_H$ ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
E	**Y**	A ⇔ Y_H EXG A,YH	B ⇔ Y_L EXG B,YL	$CCR_{H:L}$ ⇔ Y EXG CCRW,Y	$TMP3_H$ ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
F	**SP**	A ⇔ SP_H EXG A,SPH	B ⇔ SP_L EXG B,SPL	$CCR_{H:L}$ ⇔ SP EXG CCRW,SP	$TMP3_H$ ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the S12X.

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

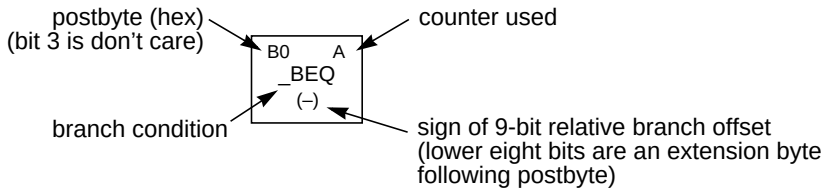


Table A-7. Branch/Complementary Branch

Branch				Complementary Branch			
Test	**Mnemonic**	**Opcode**	**Boolean**	**Test**	**Mnemonic**	**Opcode**	**Comment**
r>m	BGT	2E	$Z + (N \oplus V) = 0$	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

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

15Bit8				7Bit0			
1512		118		74		30	
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.

Instruction Reference

Appendix B. High-Level Language Support

B.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.

Motorola solicited the cooperation of third-party software developers to assure that the CPU12 instruction set would meet the needs of a more efficient generation of compilers. Several features of the CPU12 were specifically designed to improve the efficiency of compiled HLL, and thus minimize cost.

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.

B.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.

B.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 CPU12 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.

B.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 CPU12 registers can be pushed or pulled. These instructions are also useful for preserving the CCR value during a function call subroutine.

B.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).

B.3.3 Frame Pointer

In the C language, it is common to have a frame pointer in addition to the CPU12 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 CPU12 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.

B.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.

B.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 CPU12 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.

B.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.

B.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.

B.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.

B.9 Function Calls

Bank switching is a fairly common way of adapting a CPU12 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 CPU12 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 CPU12 registers during a function call. The CPU12 can push and pull any of the CPU12 registers A, B, CCR, D, X, Y, or SP.

B.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.

High-Level Language Support

Index

A

ABA instruction 108
Abbreviations for system resources 20
ABX instruction 109
ABY instruction 110
Access details 102–107, 478
Accumulator offset indexed addressing mode 47
Accumulator offset indexed indirect addressing mode 46
Accumulators 27
  A 25, 41
  B 25, 41
  D 25, 41
ADCA instruction 111
ADCB instruction 112
ADCD instruction 118
ADCX instruction 119
ADCY instruction 120
ADDA instruction 113
ADDB instruction 114
ADDD instruction 115
Addition instructions 66
ADDR mnemonic 23
Addressing modes 35
  Direct 38
  Extended 39
  Immediate 37
  Indexed 27, 40
  Inherent 36
  Relative 39
ADDX instruction 116
ADDY instruction 117
ANDA instruction 121
ANDB instruction 122
ANDCC instruction 123
ANDX instruction 124

ANDY instruction 125
Arithmetic shift 133, 136, 137, 138
ASL instruction 126
ASLA instruction 127
ASLB instruction 128
ASLD instruction 129
ASLW instruction 130, 282
ASLX instruction 131, 283
ASLY instruction 132, 284
ASR instruction 133
ASRA instruction 134
ASRB instruction 135
ASRW instruction 136
ASRX instruction 137
ASRY instruction 138
Asserted 23
Auto increment 45

B

Background debug mode 93
  Instruction 93, 144
Base index register 43–47
BCC instruction 139
BCD instructions 67, 195
BCLR instruction 140
BCS instruction 141
BEQ instruction 142
BGE instruction 143
BGND instruction 93, 144
BGT instruction 145
BHI instruction 146
BHS instruction 147
Binary-coded decimal instructions 67, 195
Bit manipulation instructions 73, 140, 163, 165, 511
  Mask operand 49, 140, 160, 162, 163, 165
  Multiple addressing modes 49
Bit test instructions 73, 83, 148, 149, 150, 151, 160, 162
BITA instruction 148
BITB instruction 149
BITBY instruction 151
Bit-condition branches 83, 160, 162
BITX instruction 150

BLE instruction 152
BLO instruction 153
BLS instruction 154
BLT instruction 155
BMI instruction 156
BNE instruction 157
Boolean logic instructions 70
AND 121, 122, 123, 124, 125
Complement 181, 182, 183, 184, 185, 186
Exclusive OR 216, 217, 218, 219, 229
Inclusive OR 319, 320, 321, 322, 323
Negate 312, 313, 314, 315, 316, 317
BPL instruction 158
BRA instruction 159
Branch instructions 39, 56–58, 80, 515
Bit-condition 58, 83, 160, 162
Long 57, 58, 81
Loop primitive 58, 84, 507
Offset values 81, 82, 83, 84
Offsets 40
Short 57, 58, 80
Signed 80–82
Simple 80–82
Subroutine 85, 164
Summary of complementary branches 139, 251
Taken/not-taken cases 57, 107
Unary 80–82
Unsigned 80–82
Branch offset 39–40
BRCLR instruction 160
BRN instruction 161
BRSET instruction 162
BSET instruction 163
BSR instruction 56, 164
BTAS instruction 165
Bus cycles 102
BVC instruction 166
BVS instruction 167
Byte moves 65, 297, 298, 299, 300, 301, 302, 303
Byte order in memory 34
Byte-sized instructions 58

C

C 126
C status bit 33, 74, 139, 141
CALL instruction . . . . . 50–??, 56, 85, 168, 516
Case statements 515
CBA instruction 169
CCR *(see Condition codes register)*
Changes in execution flow 55–59
CLC instruction 170
Clear instructions 71
Clear memory 172, 175, 176, 177
Cleared 23
CLI instruction 171
Clock monitor reset 410
CLR instruction 172
CLRA instruction 173
CLRB instruction 174
CLRW instruction 175
CLRX instruction 176
CLRY instruction 177
CLV instruction 178
CMPA instruction 179
CMPB instruction 180
COM instruction 181
COMA instruction 182
COMB instruction 183
Compare instructions 69
Complement instructions 71
Computer operating properly (COP) watchdog 410
COMW instruction 184
COMX instruction 185
COMY instruction 186
Condition codes instructions . . 91, 123, 321, 326, 327, 333, 334, 382, 389
Condition codes register 25, 28–33
C status bit 33, 74, 139, 141
H status bit 31, 195
I mask bit 32, 123, 171, 365, 401, 408, 411
Manipulation 91, 123, 321, 365
N status bit 32
S control bit 371
V status bit 33
X mask bit . . . 30, 221, 333, 334, 356, 371, 382, 387, 401, 408, 410, 411

Z status bit 32, 142, 157
Conditional 16-bit read cycle 106, 478
Conditional 8-bit read cycle 106, 478
Conditional 8-bit write cycle 106, 478
Conserving power 92, 371, 401
Constant indirect indexed addressing mode 44
Constant offset indexed addressing mode 43, 44
COP reset 410
CPCD instruction 188
CPCS instruction 189
CPCX instruction 190
CPCY instruction 191
CPD instruction 187
CPS instruction 192
CPX instruction 193
CPY instruction 194
Cycle code letters 102, 478
Cycle-by-cycle operation 102, 478

D

DAA instruction 195
DATA mnemonic 23
Data types 33, 511
DBEQ instruction 196, 507
DBNE instruction 197, 507
DEC instruction 198
DECA instruction 199
DECB instruction 200
Decrement instructions 68
DECW instruction 201
DECX instruction 202
DECY instruction 203
Defuzzification 440, 461–464
DES instruction 204
DEX instruction 205
DEY instruction 206
Direct addressing mode 38
Division instructions 72, 514
16-bit fractional 223
16-bit integer 238, 239
32-bit extended 207, 208
Double accumulator 25, 27

E

EDIV instruction 207
EDIVS instruction 208
Effective address 35, 41, 90, 275, 276, 277, 512–514
EMACS instruction 78, 209
EMAXD instruction 210
EMAXM instruction 211, 434
EMIND instruction 212, 433
EMINM instruction 213
EMUL instruction 214
EMULS instruction 215
Enabling maskable interrupts 32, 171
EORA instruction 216
EORB instruction 217, 229
EORX instruction 218
EORY instruction 219
ETBL instruction 79, 220, 434
Even bytes 34
Exceptions 55, 405
  Interrupts 410
  Maskable interrupts 411, 412
  Non-maskable interrupts 410
  Priority 407
  Processing flow 414
  Resets 405, 409–410
  Software interrupts 86, 380, 414
  Unimplemented opcode trap 405, 408, 413
  Vectors 405, 414
Exchange instructions 63, 221, 222
  Postbyte encoding 505
Execution cycles 102
Execution time 102
EXG instruction 221, 222
Expanded memory 50, 56, 516
  Bank switching 50
  Instructions 50, 85, 168, 355
  Page registers 50
  Subroutines 85, 516
Extended addressing mode 39
Extended division 72
Extension byte 41
External interrupts 412

External queue reconstruction 419
HCS12 queue reconstruction 425
HCS12 reconstruction algorithm 427
HCS12 timing detail 421, 423
External reset 409

F

f-cycle (free cycle) 102, 478
FDIV instruction 72, 223
Fractional division 72, 223
Frame pointer 513, 514
Free cycle 102, 478
Fuzzy logic 433–472
Antecedents 438, 470
Consequents 439, 470
Custom programming 467
Defuzzification 76, 440, 461–466
Fuzzification 75, 436, 467
Inference kernel 435, 441
Inputs 470
Instructions 75, 76, 294, 338–342, 402, 433, 444–466
Interrupts 458, 462–464
Knowledge base 435, 439, 470
Membership functions 75, 294, 434, 435, 436, 444–449, 467–469
Outputs 76, 470
Rule evaluation 75, 338–342, 438, 450–461, 470
Rules 436, 438, 470
Sets 435
Tabular membership functions 79, 467
Weighted average 76, 402, 433, 440, 461–466

G

g-cycle (read PPAGE) 103, 478
General purpose accumulators 25
GLDAA instruction 224
GLDAB instruction 225
GLDD instruction 226
GLDS instruction 227
GLDX instruction 228
Global interrupt mask 32, 408
GSTAA instruction 230

GSTAB instruction 231
GSTD instruction 232
GSTS instruction 233
GSTX instruction 234
GSTY instruction 235

H

H status bit 31, 195
Highest priority interrupt 408
High-level language 511–517
    Addressing modes 511, 513, 515
    Condition codes register 515
    Expanded memory 516
    Instructions 511
    Loop primitives 514
    Stack 512, 513

I

I mask bit 32, 123, 171, 365, 408
IBEQ instruction 236, 507
IBNE instruction 237, 507
I-cycle (16-bit read indirect) 103, 478
i-cycle (8-bit read indirect) 103, 478
IDIV instruction 238
IDIVS instruction 239, 514
Immediate addressing mode 37
INC instruction 240
INCA instruction 241
INCB instruction 242
Increment instructions 68
INCW instruction 243
INCX instruction 244
INCY instruction 245
Index calculation instructions 90
Index manipulation instructions 88
Index registers 25, 88, 90, 513
    PC (as an index register) 28, 42, 43, 102
    SP (as an index register) 27, 42, 43, 102
    X 27, 42, 43, 102
    Y 27, 42, 43, 102
Indexed addressing modes 27, 40–50, 503

16-bit constant indirect 44
16-bit constant offset 44
5-bit constant offset 43
9-bit constant offset 43
Accumulator direct 47
Accumulator offset 46
Auto increment/decrement indexing 45
Base index register 43–47
Extension byte 41
Limitations for BIT and MOV instructions 140, 160, 162, 163, 165, 297, 298, 300, 301, 302, 303, 305, 306, 307, 308, 309, 310
Postbyte 43
Postbyte encoding 41, 503
Inference kernel, fuzzy logic 441
Inherent addressing mode 36
INS instruction 246
Instruction pipe, see *Instruction queue*
Instruction queue 34, 53, 419
Data movement 54
Debugging 419
Reconstruction . . . . . 419–??
Status registers 427
Status signals . . . . . 54, 420–??
Instruction set 61, 95, 479
Integer division 72, 238–239
Interrupt instructions 86
Interrupts 410–416
Enabling and disabling 30, 32, 171, 365, 411
External 412
I mask bit 32, 171, 365, 412
Instructions 86, 87, 171, 356, 365, 380, 390
Low-power stop 92, 371
Maskable 32, 411
Non-maskable 30, 405–408, 410, 411
Recognition 411
Return 30, 32, 87, 356, 413
Service routines 412
Software 86, 380, 414
Stacking order 412
Vectors 405, 413, 414
Wait instruction 92, 401
X mask bit 30, 371, 401, 412
INX instruction 247

INY instruction 248

J

JMP instruction 59, 249
JSR instruction 56, 250
Jump instructions 59, 85

K

Knowledge base 435

L

Label 99
LBCC instruction 251
LBCS instruction 252
LBEQ instruction 253
LBGE instruction 254
LBGT instruction 255
LBHI instruction 256
LBHS instruction 257
LBLE instruction 258
LBLO instruction 259
LBLS instruction 260
LBLT instruction 261
LBMI instruction 262
LBNE instruction 263
LBPL instruction 264
LBRA instruction 265
LBRN instruction 266
LBVC instruction 267
LBVS instruction 268
LDAA instruction 269
LDAB instruction 270
LDD instruction 271
LDS instruction 272
LDX instruction 273
LDY instruction 274
LEAS instruction 275, 513, 516
Least significant byte 23
Least significant word 23
LEAX instruction 276, 516

LEAY instruction 277, 516
Legal label 99
Literal expression 99
Load instructions 62
Logic level one 23
Logic level zero 23
Loop primitive instructions 58, 84, 507, 514
    Offset values 84
    Postbyte encoding 507
Low-power stop 92, 371
LSL instruction 74, 278
LSLA instruction 279
LSLB instruction 280
LSLD instruction 281
LSR instruction 285
LSRA instruction 286
LSRB instruction 287
LSRD instruction 288
LSRW instruction 289
LSRX instruction 290
LSRY instruction 291

M

M68HC11 compatibility 35
Maskable interrupts 32, 411
MAXA instruction 292
Maximum instructions 78
    16-bit 210, 211
    8-bit 292, 293
MAXM instruction 293, 433
MEM instruction 75, 294, 433, 444–449
Membership functions 435, 444–449
Memory and addressing symbols 21
MINA instruction 295, 433
Minimum instructions 78
    16-bit 212, 213
    8-bit 295, 296
MINM instruction 296
Misaligned instructions 58
Mnemonic 95
Most significant byte 23
Most significant word 23

MOVB instruction 297, 298, 299, 300, 301, 302, 303
Move instructions65, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310
Destination 48
Multiple addressing modes 48
PC relative addressing 48
Reference index register 48
Source 48
MOVW instruction 304, 305, 306, 307, 308, 309, 310
MUL instruction 311
Multiple addressing modes
Bit manipulation instructions 49
Move instructions 48
Multiplication instructions 72
16-bit 214, 215
8-bit 311
Multiply and accumulate instructions 78, 209, 402, 471

N

N status bit 32
n-cycle (write PPAGE) 103, 478
NEG instruction 312
NEGA instruction 313
Negate instructions 71
Negated 23
Negative integers 33
NEGB instruction 314
NEGW instruction 315
NEGX instruction 316
NEGY instruction 317
Non-maskable interrupts 30, 408, 410
NOP instruction 93, 318
Notation
Branch taken/not taken 107, 478
Changes in CCR bits 97
Cycle-by-cycle operation 102
Memory and addressing 21
Object code 98
Operators 22, 476
Source forms 99
System resources 20
Null operation instruction 93, 318

Numeric range of branch offsets 40, 81–84

O

Object code notation 98
O-cycle (optional program word fetch) 58, 104, 478
Odd bytes 34
Offset
  Branch 39–40
  Index 40–45
Opcode map . . . . . 500, ??–501, ??–502
Operators 22, 476
Optional cycles 58, 104, 478
ORAA instruction 319
ORAB instruction 320
ORCC instruction 321
Orthogonality 517
ORX instruction 322
ORY instruction 323

P

Page 2 prebyte 57, 104, 500, 501, 502
P-cycle (program word fetch) 104, 478
Pipeline 34
Pointer calculation instructions 90, 275, 276, 277
Pointers 516
Postbyte encoding
  Exchange instructions 221, 222, 505
  Indexed addressing instructions 42
  Indexed addressing modes 43, 503
  Loop primitive instructions 507
  Transfer instructions 367, 387, 505
Post-decrement indexed addressing mode 45
Post-increment indexed addressing mode 45
Power conservation 92, 371, 401
Power-on reset 409
Prebyte 57, 104, 500, 501, 502
Pre-decrement indexed addressing mode 45
Pre-increment indexed addressing mode 45
Priority, exception 407
Program counter 25, 28, 41, 144
Program word access cycle 104, 478

Programming model 19, 25
Pseudo-non-maskable interrupt 408
PSHA instruction 324
PSHB instruction 325
PSHC instruction 326
PSHCW instruction 327
PSHD instruction 328, 512
PSHX instruction 329
PSHY instruction 330
PULA instruction 331
PULB instruction 332
PULC instruction 333, 512
PULCW instruction 334
PULD instruction 335, 512
Pull instructions 517
PULX instruction 336
PULY instruction 337
Push instructions 517

Q

Queue reconstruction 419
HCS12 queue reconstruction 425
HCS12 reconstruction algorithm 427
HCS12 timing detail 421, 423

R

R-cycle (16-bit data read) 105, 478
r-cycle (8-bit data read) 104, 478
Read 16-bit data cycle 105, 478
Read 8-bit data cycle 104, 478
Read indirect pointer cycle 103, 478
Read indirect PPAGE value cycle 103, 478
Read PPAGE cycle 103, 478
Register designators 99
Relative addressing mode 39
Relative offset 39
Resets 405, 409
Clock monitor 410
COP 410
External 409
Power-on 409

Return from call 355
Return from interrupt 356
Return from subroutine 357
REV instruction 75, 338–339, 433, 438, 450–455, 470
REVW instruction 75, 340–342, 433, 438, 456–461, 470
ROL instruction 343
ROLA instruction 344
ROLB instruction 345
ROLW instruction 346
ROLX instruction 347
ROLY instruction 348
ROR instruction 349
RORA instruction 350
RORB instruction 351
RORW instruction 352
RORX instruction 353
RORY instruction 354
Rotate instructions 74
RTC instruction 50, 56, 85, 355, 516
RTI instruction 32, 87, 356, 413
RTS instruction 56, 357

S

S control bit 371
SBA instruction 358
SBCA instruction 359
SBCB instruction 360
SBCD instruction 361
SBCX instruction 362
SBCY instruction 363
S-cycle (16-bit stack write) 105, 478
s-cycle (8-bit stack write) 105, 478
SEC instruction 364
SEI instruction 365
Service routine 406
Set 23
Setting memory bits 163, 165
SEV instruction 366
SEX instruction 63, 367
Shift instructions 74
    Arithmetic 133, 136, 137, 138
Sign extension instruction 63, 367, 512

Signed branches 80–82
Signed integers 33
Signed multiplication 72
Simple branches 80–82
Software interrupts 380
Source code compatibility 19
Source form notation 99
STAA instruction 368
STAB instruction 369
Stack 27
Stack 16-bit data cycle 105, 478
Stack 8-bit data cycle 105, 478
Stack operation instructions 89
Stack pointer 25, 27, 41, 512
    Initialization 27
    Manipulation 89
    Stacking order 412
Stack pointer instructions 89, 512
Standard CPU12 address space 34
STD instruction 370
STOP continue 371
STOP disable 30, 371
STOP instruction 92, 371
Store instructions 62
STS instruction 372
STX instruction 373
STY instruction 374
SUBA instruction 375
SUBB instruction 376
SUBD instruction 377
Subroutine instructions 85
Subroutines 56, 516
    Expanded memory 56, 85, 168, 355, 516
    Instructions 85, 164, 168, 250, 516
    Return 355, 357
Subtraction instructions 66
SUBX instruction 378
SUBY instruction 379
SWI instruction 86, 380, 414
Switch statements 515
Symbols and notation 20, 475

T

TAB instruction . . . 381
Table interpolation instructions . . . 79, 220, 385
Tabular membership functions . . . 467–469
TAP instruction . . . 382
TBA instruction . . . 383
TBEQ instruction . . . 384, 507
TBL instruction . . . 79, 385, 434, 467–468
TBNE instruction . . . 386, 507
T-cycle (16-bit conditional read) . . . 106, 478
t-cycle (8-bit conditional read) . . . 106, 478
Termination of interrupt service routines . . . 87, 356, 413
Termination of subroutines . . . 355, 357
Test instructions . . . 69
TFR instruction . . . 387, 388
TPA instruction . . . 389
Transfer instructions . . . 63
  Postbyte encoding . . . 505
TRAP instruction . . . 87, 390, 413, 501
TST instruction . . . 391
TSTA instruction . . . 392
TSTB instruction . . . 393
TSTW instruction . . . 394
TSTX instruction . . . 395
TSTY instruction . . . 396
TSX instruction . . . 397
TSY instruction . . . 398
Twos-complement form . . . 33
TXS instruction . . . 399
Types of instructions
  Addition and Subtraction . . . 66
  Background and null . . . 93
  Binary-coded decimal . . . 67
  Bit test and manipulation . . . 73
  Boolean logic . . . 70
  Branch . . . 80
  Clear, complement, and negate . . . 71
  Compare and test . . . 69
  Condition code . . . 91
  Decrement and increment . . . 68
  Fuzzy logic . . . 75
  Index manipulation . . . 88

Interrupt 86–87
Jump and subroutine 85
Load and store 62
Loop primitives 84
Maximum and minimum 78
Move 65
Multiplication and division 72
Multiply and accumulate 78
Pointer and index calculation 90
Shift and rotate 74
Sign extension 63
Stacking 89
Stop and wait 92
Table interpolation 79
Transfer and exchange 63
TYS instruction 400

U

U-cycle (16-bit stack read) 106, 478
u-cycle (8-bit stack read) 105, 478
Unary branches 80–82
Unimplemented opcode trap 87, 390, 405, 408, 501
Unsigned branches 80–82
Unsigned multiplication 72
Unstack 16-bit data cycle 106, 478
Unstack 8-bit data cycle 105, 478
Unweighted rule evaluation 338–339, 438, 450–455, 470

V

V status bit 33, 91
V-cycle (vector fetch) 106, 478
Vector fetch cycle 106, 478
Vectors, exception 405, 414

W

WAI instruction 92, 401
Wait instruction 92, 401
Watchdog 410
WAV instruction 76, 402, 433, 440, 461–464
HCS12 465

M68HC12 466
wavr pseudo-instruction 462–464
HCS12 465
M68HC12 466
W-cycle (16-bit data write) 105, 478
w-cycle (8-bit data write) 105, 478
Weighted average 402
Weighted rule evaluation 340–342, 438, 450–452, 456–461, 470
Word moves 65, 304, 305, 306, 307, 308, 309, 310
Write 16-bit data cycle 105, 478
Write 8-bit data cycle 105, 478
Write PPAGE cycle 103, 478

X

X mask bit 30, 221, 333, 334, 356, 371, 382, 387, 401
x-cycle (8-bit conditional write) 106, 478
XGDX instruction 403
XGDY instruction 404

Z

Z status bit 32, 142, 157
Zero-page addressing 38

Index

HOW TO REACH US:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-800-521-6274 or 480-768-2130

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573, Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
852-26668334

HOME PAGE:
http://motorola.com/semiconductors

Information in this document is provided solely to enable system and software implementers to use Motorola 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.

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola 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 Motorola 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. Motorola does not convey any license under its patent rights nor the rights of others. Motorola 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 Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola 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 Motorola was negligent regarding the design or manufacture of the part.

MOTOROLA and the Stylized M Logo are registered in the US Patent and Trademark Office. All other product or service names are the property of their respective owners. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

© Motorola Inc. 2004

S12XCPUV1
Rev. 01.00
2/2004