

CPU12

Reference Manual



Digital DNA™
from Motorola

CPU12

Reference Manual

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 which 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, Inc. is an Equal Opportunity/Affirmative Action Employer.

Reference Manual

List of Sections

| | |
|---|------------|
| Section 1. Introduction | 21 |
| Section 2. Overview | 27 |
| Section 3. Addressing Modes | 37 |
| Section 4. Instruction Queue | 55 |
| Section 5. Instruction Set Overview | 63 |
| Section 6. Instruction Glossary | 97 |
| Section 7. Exception Processing..... | 323 |
| Section 8. Development and Debug Support..... | 335 |
| Section 9. Fuzzy Logic Support..... | 355 |
| Section 10. Memory Expansion..... | 395 |
| Appendix A. Instruction Reference | 407 |
| Appendix B. M68HC11 to M68HC12 Upgrade Path..... | 441 |
| Appendix C. High-Level Language Support..... | 465 |
| Index..... | 473 |

List of Sections

Table of Contents

Section 1. Introduction

| | | |
|-------|--|----|
| 1.1 | Contents | 21 |
| 1.2 | Introduction | 21 |
| 1.3 | Features | 21 |
| 1.4 | Readership | 22 |
| 1.5 | Symbols and Notation. | 23 |
| 1.5.1 | Abbreviations for System Resources | 23 |
| 1.5.2 | Memory and Addressing | 24 |
| 1.5.3 | Operators | 25 |
| 1.5.4 | Definitions. | 26 |

Section 2. Overview

| | | |
|---------|-----------------------------------|----|
| 2.1 | Contents | 27 |
| 2.2 | Introduction | 27 |
| 2.3 | Programming Model | 28 |
| 2.3.1 | Accumulators | 29 |
| 2.3.2 | Index Registers | 29 |
| 2.3.3 | Stack Pointer | 29 |
| 2.3.4 | Program Counter | 30 |
| 2.3.5 | Condition Code Register | 30 |
| 2.3.5.1 | S Control Bit | 31 |
| 2.3.5.2 | X Mask Bit | 32 |
| 2.3.5.3 | H Status Bit | 32 |
| 2.3.5.4 | I Mask Bit | 33 |
| 2.3.5.5 | N Status Bit | 33 |
| 2.3.5.6 | Z Status Bit | 33 |
| 2.3.5.7 | V Status Bit | 34 |
| 2.3.5.8 | C Status Bit | 34 |

Table of Contents

| | | |
|-----|-------------------------------|----|
| 2.4 | Data Types | 34 |
| 2.5 | Memory Organization | 35 |
| 2.6 | Instruction Queue | 35 |

Section 3. Addressing Modes

| | | |
|--------|---|----|
| 3.1 | Contents | 37 |
| 3.2 | Introduction | 37 |
| 3.3 | Mode Summary | 38 |
| 3.4 | Effective Address | 39 |
| 3.5 | Inherent Addressing Mode | 39 |
| 3.6 | Immediate Addressing Mode | 39 |
| 3.7 | Direct Addressing Mode | 40 |
| 3.8 | Extended Addressing Mode | 41 |
| 3.9 | Relative Addressing Mode | 41 |
| 3.10 | Indexed Addressing Modes | 43 |
| 3.10.1 | 5-Bit Constant Offset Indexed Addressing | 45 |
| 3.10.2 | 9-Bit Constant Offset Indexed Addressing | 46 |
| 3.10.3 | 16-Bit Constant Offset Indexed Addressing | 46 |
| 3.10.4 | 16-Bit Constant Indirect Indexed Addressing | 47 |
| 3.10.5 | Auto Pre/Post Decrement/Increment Indexed Addressing | 47 |
| 3.10.6 | Accumulator Offset Indexed Addressing | 49 |
| 3.10.7 | Accumulator D Indirect Indexed Addressing | 49 |
| 3.11 | Instructions Using Multiple Modes | 50 |
| 3.11.1 | Move Instructions | 50 |
| 3.11.2 | Bit Manipulation Instructions | 52 |
| 3.12 | Addressing More than 64 Kbytes | 52 |

Section 4. Instruction Queue

| | | |
|-----|-----------------------------|----|
| 4.1 | Contents | 55 |
| 4.2 | Introduction | 55 |
| 4.3 | Queue Description | 56 |

| | | |
|---------|--------------------------------------|----|
| 4.4 | Data Movement in the Queue..... | 57 |
| 4.4.1 | No Movement..... | 57 |
| 4.4.2 | Latch Data from Bus..... | 57 |
| 4.4.3 | Advance and Load from Data Bus | 57 |
| 4.4.4 | Advance and Load from Buffer..... | 57 |
| 4.5 | Changes in Execution Flow | 58 |
| 4.5.1 | Exceptions | 58 |
| 4.5.2 | Subroutines | 59 |
| 4.5.3 | Branches | 60 |
| 4.5.3.1 | Short Branches | 60 |
| 4.5.3.2 | Long Branches..... | 61 |
| 4.5.3.3 | Bit Condition Branches | 62 |
| 4.5.3.4 | Loop Primitives | 62 |
| 4.5.4 | Jumps..... | 62 |

Section 5. Instruction Set Overview

| | | |
|------|---|----|
| 5.1 | Contents | 63 |
| 5.2 | Introduction..... | 64 |
| 5.3 | Instruction Set Description | 65 |
| 5.4 | Load and Store Instructions | 66 |
| 5.5 | Transfer and Exchange Instructions..... | 67 |
| 5.6 | Move Instructions | 68 |
| 5.7 | Addition and Subtraction Instructions..... | 69 |
| 5.8 | Binary-Coded Decimal Instructions | 70 |
| 5.9 | Decrement and Increment Instructions..... | 71 |
| 5.10 | Compare and Test Instructions..... | 72 |
| 5.11 | Boolean Logic Instructions | 73 |
| 5.12 | Clear, Complement, and Negate Instructions..... | 74 |
| 5.13 | Multiplication and Division Instructions..... | 75 |
| 5.14 | Bit Test and Manipulation Instructions | 76 |
| 5.15 | Shift and Rotate Instructions..... | 77 |

Table of Contents

| | | |
|--------|--|----|
| 5.16 | Fuzzy Logic Instructions | 78 |
| 5.16.1 | Fuzzy Logic Membership Instruction | 78 |
| 5.16.2 | Fuzzy Logic Rule Evaluation Instructions | 78 |
| 5.16.3 | Fuzzy Logic Averaging Instruction | 79 |
| 5.17 | Maximum and Minimum Instructions | 81 |
| 5.18 | Multiply and Accumulate Instruction | 82 |
| 5.19 | Table Interpolation Instructions | 82 |
| 5.20 | Branch Instructions | 83 |
| 5.20.1 | Short Branch Instructions | 84 |
| 5.20.2 | Long Branch Instructions | 85 |
| 5.20.3 | Bit Condition Branch Instructions | 86 |
| 5.21 | Loop Primitive Instructions | 87 |
| 5.22 | Jump and Subroutine Instructions | 88 |
| 5.23 | Interrupt Instructions | 89 |
| 5.24 | Index Manipulation Instructions | 91 |
| 5.25 | Stacking Instructions | 92 |
| 5.26 | Pointer and Index Calculation Instructions | 93 |
| 5.27 | Condition Code Instructions | 94 |
| 5.28 | Stop and Wait Instructions | 95 |
| 5.29 | Background Mode and Null Operations | 96 |

Section 6. Instruction Glossary

| | | |
|-----|------------------------------------|-----|
| 6.1 | Contents | 97 |
| 6.2 | Introduction | 97 |
| 6.3 | Glossary Information | 98 |
| 6.4 | Condition Code Changes | 99 |
| 6.5 | Object Code Notation | 100 |
| 6.6 | Source Forms | 101 |
| 6.7 | Cycle-by-Cycle Execution | 104 |
| 6.8 | Glossary | 109 |

Section 7. Exception Processing

| | | |
|-------|--|-----|
| 7.1 | Contents | 323 |
| 7.2 | Introduction | 323 |
| 7.3 | Types of Exceptions | 324 |
| 7.4 | Exception Priority | 325 |
| 7.5 | Resets | 326 |
| 7.5.1 | Power-On Reset | 327 |
| 7.5.2 | External Reset | 327 |
| 7.5.3 | COP Reset | 327 |
| 7.5.4 | Clock Monitor Reset | 327 |
| 7.6 | Interrupts | 328 |
| 7.6.1 | Non-Maskable Interrupt Request (XIRQ) | 328 |
| 7.6.2 | Maskable Interrupts | 329 |
| 7.6.3 | Interrupt Recognition | 329 |
| 7.6.4 | External Interrupts | 330 |
| 7.6.5 | Return-from-Interrupt Instruction (RTI) | 330 |
| 7.7 | Unimplemented Opcode Trap | 331 |
| 7.8 | Software Interrupt Instruction (SWI) | 331 |
| 7.9 | Exception Processing Flow | 331 |
| 7.9.1 | Vector Fetch | 333 |
| 7.9.2 | Reset Exception Processing | 333 |
| 7.9.3 | Interrupt and Unimplemented Opcode Trap Exception Processing | 333 |

Section 8. Development and Debug Support

| | | |
|-------|---|-----|
| 8.1 | Contents | 335 |
| 8.2 | Introduction | 336 |
| 8.3 | External Reconstruction of the Queue | 336 |
| 8.4 | Instruction Queue Status Signals | 337 |
| 8.4.1 | Zero Encoding (0:0) | 338 |
| 8.4.2 | LAT — Latch Data from Bus Encoding (0:1) | 339 |
| 8.4.3 | ALD — Advance and Load from Data Bus Encoding (1:0) | 339 |
| 8.4.4 | ALL — Advance and Load from Latch Encoding (1:1) | 339 |

Table of Contents

| | | |
|---------|---|-----|
| 8.4.5 | INT — Interrupt Sequence Encoding (0:1) | 339 |
| 8.4.6 | SEV — Start Instruction on Even Address Encoding (1:0) | 340 |
| 8.4.7 | SOD — Start Instruction on Odd Address Encoding (1:1) | 340 |
| 8.5 | Implementing Queue Reconstruction | 340 |
| 8.5.1 | Queue Status Registers | 341 |
| 8.5.1.1 | in_add, in_dat Registers | 341 |
| 8.5.1.2 | fetch_add, fetch_dat Registers | 341 |
| 8.5.1.3 | st1_add, st1_dat Registers | 342 |
| 8.5.1.4 | st2_add, st2_dat Registers | 342 |
| 8.5.2 | Reconstruction Algorithm | 342 |
| 8.5.2.1 | LAT Decoding | 343 |
| 8.5.2.2 | ALD Decoding | 343 |
| 8.5.2.3 | ALL Decoding | 343 |
| 8.6 | Background Debug Mode | 344 |
| 8.6.1 | Enabling BDM | 344 |
| 8.6.2 | BDM Serial Interface | 345 |
| 8.6.3 | BDM Commands | 347 |
| 8.6.4 | BDM Registers | 350 |
| 8.7 | Instruction Tagging | 352 |
| 8.8 | Breakpoints | 353 |
| 8.8.1 | Breakpoint Type | 353 |
| 8.8.2 | Breakpoint Operation | 354 |

Section 9. Fuzzy Logic Support

| | | |
|-------|---|-----|
| 9.1 | Contents | 355 |
| 9.2 | Introduction | 356 |
| 9.3 | Fuzzy Logic Basics | 357 |
| 9.3.1 | Fuzzification (MEM) | 359 |
| 9.3.2 | Rule Evaluation (REV and REVW) | 361 |
| 9.3.3 | Defuzzification (WAV) | 363 |
| 9.4 | Example Inference Kernel | 364 |
| 9.5 | MEM Instruction Details | 366 |
| 9.5.1 | Membership Function Definitions | 366 |

| | | |
|---------|--|-----|
| 9.5.2 | Abnormal Membership Function Definitions | 368 |
| 9.5.2.1 | Abnormal Membership Function Case 1 | 370 |
| 9.5.2.2 | Abnormal Membership Function Case 2 | 371 |
| 9.5.2.3 | Abnormal Membership Function Case 3 | 371 |
| 9.6 | REV and REVW Instruction Details | 372 |
| 9.6.1 | Unweighted Rule Evaluation (REV) | 372 |
| 9.6.1.1 | Set Up Prior to Executing REV | 372 |
| 9.6.1.2 | Interrupt Details | 374 |
| 9.6.1.3 | Cycle-by-Cycle Details for REV | 374 |
| 9.6.2 | Weighted Rule Evaluation (REVW) | 378 |
| 9.6.2.1 | Set Up Prior to Executing REVW | 378 |
| 9.6.2.2 | Interrupt Details | 380 |
| 9.6.2.3 | Cycle-by-Cycle Details for REVW | 380 |
| 9.7 | WAV Instruction Details | 383 |
| 9.7.1 | Set Up Prior to Executing WAV | 384 |
| 9.7.2 | WAV Interrupt Details | 384 |
| 9.7.3 | Cycle-by-Cycle Details for WAV and wavr | 385 |
| 9.8 | Custom Fuzzy Logic Programming | 388 |
| 9.8.1 | Fuzzification Variations | 388 |
| 9.8.2 | Rule Evaluation Variations | 391 |
| 9.8.3 | Defuzzification Variations | 392 |

Section 10. Memory Expansion

| | | |
|----------|---|-----|
| 10.1 | Contents | 395 |
| 10.2 | Introduction | 396 |
| 10.3 | Expansion System Description | 396 |
| 10.4 | CALL and Return from Call Instructions | 398 |
| 10.5 | Address Lines for Expansion Memory | 401 |
| 10.6 | Overlay Window Controls | 401 |
| 10.7 | Using Chip-Select Circuits | 402 |
| 10.7.1 | Program Memory Expansion Chip-Select Controls | 403 |
| 10.7.1.1 | CSP1E Control Bit | 403 |
| 10.7.1.2 | CSP0E Control Bit | 403 |
| 10.7.1.3 | CSP1FL Control Bit | 403 |
| 10.7.1.4 | CSPA21 Control Bit | 403 |

Table of Contents

| | | |
|----------|--|-----|
| 10.7.1.5 | STRP0A:STRP0B Control Field | 403 |
| 10.7.1.6 | STRP1A:STRP1B Control Field | 404 |
| 10.7.2 | Data Expansion Chip Select Controls | 404 |
| 10.7.2.1 | CSDE Control Bit | 404 |
| 10.7.2.2 | CSDHF Control Bit | 404 |
| 10.7.2.3 | STRDA:STRDB Control Field | 405 |
| 10.7.3 | Extra Expansion Chip Select Controls | 405 |
| 10.7.3.1 | CSEE Control Bit | 405 |
| 10.7.3.2 | CSEEP Control Bit | 405 |
| 10.7.3.3 | STREA:STREB Control Field | 405 |
| 10.8 | System Notes | 405 |

Appendix A. Instruction Reference

| | | |
|---------|--|-----|
| A.1 | Contents | 407 |
| A.2 | Introduction | 407 |
| A.3 | Instruction Set Summary | 408 |
| A.3.1 | Notation Used in Instruction Set Summary | 408 |
| A.3.1.1 | Explanation of Italic Expressions in Source Form Column | 408 |
| A.3.1.2 | Address Modes | 409 |
| A.3.1.3 | Machine Coding | 409 |
| A.3.1.4 | Access Detail | 410 |
| A.3.1.5 | Condition Codes Columns | 410 |
| A.4 | Indexed Addressing Postbyte Encoding | 411 |
| A.5 | Transfer and Exchange Postbyte Encoding | 411 |
| A.6 | Loop Primitive Postbyte Encoding | 411 |
| A.7 | Opcode Map | 412 |
| A.8 | Memory Expansion | 432 |
| A.9 | Hexadecimal to ASCII Conversion | 438 |
| A.10 | Hexadecimal to Decimal Conversion | 439 |
| A.11 | Decimal to Hexadecimal Conversion | 439 |

Appendix B. M68HC11 to M68HC12 Upgrade Path

| | | |
|--------|--|-----|
| B.1 | Contents | 441 |
| B.2 | Introduction | 442 |
| B.3 | CPU12 Design Goals | 442 |
| B.4 | Source Code Compatibility | 442 |
| B.5 | Programmer's Model and Stacking | 445 |
| B.6 | True 16-Bit Architecture | 445 |
| B.6.1 | Bus Structures | 446 |
| B.6.2 | Instruction Queue | 446 |
| B.6.3 | Stack Function | 448 |
| B.7 | Improved Indexing | 449 |
| B.7.1 | Constant Offset Indexing | 450 |
| B.7.2 | Auto-Increment Indexing | 451 |
| B.7.3 | Accumulator Offset Indexing | 452 |
| B.7.4 | Indirect Indexing | 453 |
| B.8 | Improved Performance | 453 |
| B.8.1 | Reduced Cycle Counts | 453 |
| B.8.2 | Fast Math | 454 |
| B.8.3 | Code Size Reduction | 455 |
| B.9 | Additional Functions | 456 |
| B.9.1 | Memory-to-Memory Moves | 458 |
| B.9.2 | Universal Transfer and Exchange | 459 |
| B.9.3 | Loop Construct | 459 |
| B.9.4 | Long Branches | 459 |
| B.9.5 | Minimum and Maximum Instructions | 460 |
| B.9.6 | Fuzzy Logic Support | 461 |
| B.9.7 | Table Lookup and Interpolation | 461 |
| B.9.8 | Extended Bit Manipulation | 462 |
| B.9.9 | Push and Pull D and CCR | 462 |
| B.9.10 | Compare SP | 462 |
| B.9.11 | Support for Memory Expansion | 462 |

Table of Contents

Appendix C. High-Level Language Support

| | | |
|-------|--|-----|
| C.1 | Contents | 465 |
| C.2 | Introduction..... | 465 |
| C.3 | Data Types | 466 |
| C.4 | Parameters and Variables | 466 |
| C.4.1 | Register Pushes and Pulls | 467 |
| C.4.2 | Allocating and Deallocating Stack Space..... | 467 |
| C.4.3 | Frame Pointer..... | 468 |
| C.5 | Increment and Decrement Operators..... | 469 |
| C.6 | Higher Math Functions | 469 |
| C.7 | Conditional If Constructs..... | 470 |
| C.8 | Case and Switch Statements | 470 |
| C.9 | Pointers..... | 470 |
| C.10 | Function Calls..... | 471 |
| C.11 | Instruction Set Orthogonality | 472 |

Index

| | |
|-------------|-----|
| Index | 473 |
|-------------|-----|

List of Figures

| Figure | Title | Page |
|---------------|--|-------------|
| 2-1 | Programming Model..... | 28 |
| 6-1 | Example Glossary Page..... | 98 |
| 7-1 | Exception Processing Flow Diagram | 332 |
| 8-1 | Queue Status Signal Timing..... | 338 |
| 8-2 | BDM Host to Target Serial Bit Timing | 346 |
| 8-3 | BDM Target to Host Serial Bit Timing (Logic 1) | 346 |
| 8-4 | BDM Target to Host Serial Bit Timing (Logic 0) | 347 |
| 8-5 | BDM Status Register (STATUS) | 350 |
| 8-6 | Tag Input Timing | 352 |
| 9-1 | Block Diagram of a Fuzzy Logic System | 358 |
| 9-2 | Fuzzification Using Membership Functions | 360 |
| 9-3 | Fuzzy Inference Engine | 364 |
| 9-4 | Defining a Normal Membership Function..... | 367 |
| 9-5 | MEM Instruction Flow Diagram | 369 |
| 9-6 | Abnormal Membership Function Case 1 | 370 |
| 9-7 | Abnormal Membership Function Case 2 | 371 |
| 9-8 | Abnormal Membership Function Case 3 | 371 |
| 9-9 | REV Instruction Flow Diagram..... | 375 |
| 9-10 | REVW Instruction Flow Diagram | 382 |
| 9-11 | WAV and wavr Instruction Flow Diagram..... | 386 |
| 9-12 | Endpoint Table Handling | 390 |
| 10-1 | Memory Expansion Paging Summary | 434 |

List of Figures

List of Tables

| Table | Title | Page |
|--------------|--|-------------|
| 3-1 | M68HC12 Addressing Mode Summary | 38 |
| 3-2 | Summary of Indexed Operations | 44 |
| 3-3 | PC Offsets for MOVE Instructions | 51 |
| 5-1 | Load and Store Instructions | 66 |
| 5-2 | Transfer and Exchange Instructions..... | 67 |
| 5-3 | Move Instructions | 68 |
| 5-4 | Addition and Subtraction Instructions..... | 69 |
| 5-5 | BCD Instructions..... | 70 |
| 5-6 | Decrement and Increment Instructions..... | 71 |
| 5-7 | Compare and Test Instructions | 72 |
| 5-8 | Boolean Logic Instructions | 73 |
| 5-9 | Clear, Complement, and Negate Instructions..... | 74 |
| 5-10 | Multiplication and Division Instructions..... | 75 |
| 5-11 | Bit Test and Manipulation Instructions | 76 |
| 5-12 | Shift and Rotate Instructions | 77 |
| 5-13 | Fuzzy Logic Instructions..... | 79 |
| 5-14 | Minimum and Maximum Instructions | 81 |
| 5-15 | Multiply and Accumulate Instructions | 82 |
| 5-16 | Table Interpolation Instructions | 83 |
| 5-17 | Short Branch Instructions | 84 |
| 5-18 | Long Branch Instructions | 85 |
| 5-19 | Bit Condition Branch Instructions | 86 |
| 5-20 | Loop Primitive Instructions | 87 |
| 5-21 | Jump and Subroutine Instructions | 89 |
| 5-22 | Interrupt Instructions..... | 90 |
| 5-23 | Index Manipulation Instructions | 91 |
| 5-24 | Stacking Instructions | 92 |
| 5-25 | Pointer and Index Calculation Instructions | 93 |

List of Tables

| Table | Title | Page |
|-------|---|------|
| 5-26 | Condition Code Instructions | 94 |
| 5-27 | Stop and Wait Instructions | 95 |
| 5-28 | Background Mode and Null Operation Instructions | 96 |
| 7-1 | CPU12 Exception Vector Map | 324 |
| 7-2 | Stacking Order on Entry to Interrupts | 329 |
| 8-1 | IPIPE1 and IPIPE0 Decoding | 338 |
| 8-2 | BDM Commands Implemented in Hardware | 348 |
| 8-3 | BDM Firmware Commands | 349 |
| 8-4 | BDM Register Mapping | 350 |
| 8-5 | Tag Pin Function | 352 |
| 10-1 | Mapping Precedence | 398 |
| A-1 | Instruction Set Summary | 413 |
| A-2 | Indexed Addressing Mode Postbyte Encoding (xb) | 428 |
| A-3 | Summary of Indexed Operations | 429 |
| A-4 | Transfer and Exchange Postbyte Encoding | 430 |
| A-5 | Loop Primitive Postbyte Encoding (lb) | 431 |
| A-6 | Branch/Complementary Branch | 431 |
| A-7 | CPU12 Opcode Map | 436 |
| A-8 | Hexadecimal to ASCII Conversion | 438 |
| A-9 | Hexadecimal to/from Decimal Conversion | 439 |
| B-1 | Translated M68HC11 Mnemonics | 443 |
| B-2 | Instructions with Smaller Object Code | 444 |
| B-3 | Comparison of Math Instruction Speeds | 454 |
| B-4 | New M68HC12 Instructions | 456 |

Section 1. Introduction

1.1 Contents

| | | |
|-------|--|----|
| 1.2 | Introduction | 21 |
| 1.3 | Features | 21 |
| 1.4 | Readership | 22 |
| 1.5 | Symbols and Notation. | 23 |
| 1.5.1 | Abbreviations for System Resources | 23 |
| 1.5.2 | Memory and Addressing | 24 |
| 1.5.3 | Operators | 25 |
| 1.5.4 | Definitions. | 26 |

1.2 Introduction

This manual describes the features and operation of the CPU12 (central processing unit, CPU) used in all M68HC12 microcontrollers.

1.3 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.

The CPU12 has full 16-bit data paths and can perform arithmetic operations up to 20 bits wide for high-speed math execution.

Unlike many other 16-bit CPUs, the CPU12 allows instructions with odd byte counts, including many single-byte instructions. This allows much more efficient use of read-only memory (ROM) space.

An instruction queue buffers program information so the CPU has immediate access to at least three bytes of machine code at the start of every instruction.

In addition to the addressing modes found in other Motorola MCUs, the CPU12 offers an extensive set of indexed addressing capabilities including:

- Using the stack pointer as an index register in all indexed operations
- Using the program counter as an index register in all but auto increment/decrement mode
- Accumulator offsets allowed using A, B, or D accumulators
- Automatic pre- or post-increment or pre- or post-decrement (by -8 to +8)
- 5-bit, 9-bit, or 16-bit signed constant offsets
- 16-bit offset indexed-indirect and accumulator D offset indexed-indirect addressing

1.4 Readership

This manual is written for professionals and students in electronic design and software development. The primary goal is to provide information necessary to implement control systems using M68HC12 devices. Basic knowledge of electronics, microprocessors, and assembly language programming is required to use the manual effectively. Because the CPU12 has a great deal of commonality with the M68HC11 CPU, prior knowledge of M68HC11 devices is helpful, but is not essential. The CPU12 also includes features that are new and unique. In these cases, supplementary material in the text explains the new technology.

1.5 Symbols and Notation

The symbols and notation shown here are used throughout the manual. More specialized usages that apply only to the instruction glossary are described at the beginning of that section.

1.5.1 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 |

1.5.2 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 location pointed to by the effective address concatenated with the next higher memory location. The most significant byte is at location M. |
| M~M+3 | — 32-bit memory location. Consists of the effective address of the instruction concatenated with 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, respectively. |
| 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 |

1.5.3 Operators

- + — Addition
- — Subtraction
- — Logical AND
- + — Logical OR (inclusive)
- \oplus — Logical exclusive OR
- \times — Multiplication
- \div — 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.

- \Rightarrow — Transfer
 - Example: (A) \Rightarrow M means the content of accumulator A is transferred to memory location M.
- \Leftrightarrow — Exchange
 - Example: D \Leftrightarrow X means exchange the contents of D with those of X.

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

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

Section 2. Overview

2.1 Contents

| | | |
|---------|-----------------------------------|----|
| 2.2 | Introduction | 27 |
| 2.3 | Programming Model | 28 |
| 2.3.1 | Accumulators | 29 |
| 2.3.2 | Index Registers | 29 |
| 2.3.3 | Stack Pointer | 29 |
| 2.3.4 | Program Counter | 30 |
| 2.3.5 | Condition Code Register | 30 |
| 2.3.5.1 | S Control Bit | 31 |
| 2.3.5.2 | X Mask Bit | 32 |
| 2.3.5.3 | H Status Bit | 32 |
| 2.3.5.4 | I Mask Bit | 33 |
| 2.3.5.5 | N Status Bit | 33 |
| 2.3.5.6 | Z Status Bit | 33 |
| 2.3.5.7 | V Status Bit | 34 |
| 2.3.5.8 | C Status Bit | 34 |
| 2.4 | Data Types | 34 |
| 2.5 | Memory Organization | 35 |
| 2.6 | Instruction Queue | 35 |

2.2 Introduction

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

2.3 Programming Model

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

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

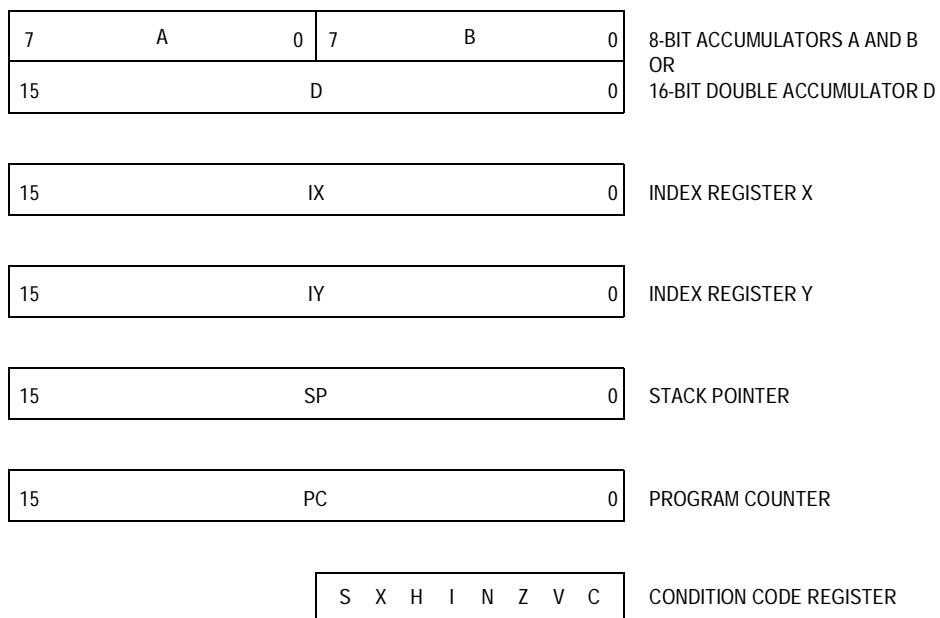


Figure 2-1. Programming Model

2.3.1 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.

2.3.2 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.

2.3.3 Stack Pointer

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

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

When a subroutine is called, the address of the instruction following the calling instruction is automatically calculated and pushed onto the stack. Normally, a return-from-subroutine (RTS) or a return-from-call (RTC)

instruction is executed at the end of a subroutine. The return instruction loads the program counter with the previously stacked return address and execution continues at that address.

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

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

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

2.3.4 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.

2.3.5 Condition Code Register

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

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

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

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

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

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

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

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

2.3.5.1 S Control Bit

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

2.3.5.2 X Mask Bit

The XIRQ input is an updated version of the 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 XIRQ pin are not recognized. An instruction must clear the X bit to enable non-maskable interrupt service requests made via the 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 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 NMI) once X has been cleared.

2.3.5.3 H Status Bit

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

2.3.5.4 *I* Mask Bit

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

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

Normally, an RTI instruction at the end of the interrupt service routine restores register values that were present before the interrupt occurred. Since the CCR is stacked before the I bit is set, the RTI normally clears the I bit, and thus re-enables interrupts. Interrupts can be re-enabled by clearing the I bit within the service routine, but implementing a nested interrupt management scheme requires great care and seldom improves system performance.

2.3.5.5 *N* Status Bit

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

2.3.5.6 *Z* Status Bit

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

2.3.5.7 V Status Bit

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

2.3.5.8 C Status Bit

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

2.4 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.

2.5 Memory Organization

The standard CPU12 address space is 64 Kbytes. Some M68HC12 devices support a paged memory expansion scheme that increases the standard space by means of predefined windows in address space. The CPU12 has special instructions that support use of expanded memory. See [Section 10. Memory Expansion](#) for more information.

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.

2.6 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.

Overview

Section 3. Addressing Modes

3.1 Contents

| | | |
|--------|---|----|
| 3.2 | Introduction | 37 |
| 3.3 | Mode Summary | 38 |
| 3.4 | Effective Address | 39 |
| 3.5 | Inherent Addressing Mode | 39 |
| 3.6 | Immediate Addressing Mode | 39 |
| 3.7 | Direct Addressing Mode | 40 |
| 3.8 | Extended Addressing Mode | 41 |
| 3.9 | Relative Addressing Mode | 41 |
| 3.10 | Indexed Addressing Modes | 43 |
| 3.10.1 | 5-Bit Constant Offset Indexed Addressing | 45 |
| 3.10.2 | 9-Bit Constant Offset Indexed Addressing | 46 |
| 3.10.3 | 16-Bit Constant Offset Indexed Addressing | 46 |
| 3.10.4 | 16-Bit Constant Indirect Indexed Addressing | 47 |
| 3.10.5 | Auto Pre/Post Decrement/Increment Indexed Addressing | 47 |
| 3.10.6 | Accumulator Offset Indexed Addressing | 49 |
| 3.10.7 | Accumulator D Indirect Indexed Addressing | 49 |
| 3.11 | Instructions Using Multiple Modes | 50 |
| 3.11.1 | Move Instructions | 50 |
| 3.11.2 | Bit Manipulation Instructions | 52 |
| 3.12 | Addressing More than 64 Kbytes | 52 |

3.2 Introduction

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

3.3 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](#).

Table 3-1. M68HC12 Addressing Mode Summary

| Addressing Mode | Source Format | Abbreviation | Description |
|--|--|--------------|---|
| Inherent | INST (no externally supplied operands) | INH | Operands (if any) are in CPU registers |
| Immediate | INST #opr8i or INST #opr16i | IMM | Operand is included in instruction stream 8- or 16-bit size implied by context |
| Direct | INST opr8a | DIR | Operand is the lower 8 bits of an address in the range \$0000–\$00FF |
| Extended | INST opr16a | EXT | Operand is a 16-bit address |
| Relative | INST rel8 or INST rel16 | REL | An 8-bit or 16-bit relative offset from the current pc is supplied in the instruction |
| Indexed (5-bit offset) | INST oprx5,xysp | IDX | 5-bit signed constant offset from X, Y, SP, or PC |
| Indexed (pre-decrement) | INST oprx3,-xys | IDX | Auto pre-decrement x, y, or sp by 1 ~ 8 |
| Indexed (pre-increment) | INST oprx3,+xys | IDX | Auto pre-increment x, y, or sp by 1 ~ 8 |
| Indexed (post-decrement) | INST oprx3,xys- | IDX | Auto post-decrement x, y, or sp by 1 ~ 8 |
| Indexed (post-increment) | INST oprx3,xys+ | IDX | Auto post-increment x, y, or sp by 1 ~ 8 |
| Indexed (accumulator offset) | INST abd,xysp | IDX | Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from X, Y, SP, or PC |
| Indexed (9-bit offset) | INST oprx9,xysp | IDX1 | 9-bit signed constant offset from X, Y, SP, or PC (lower 8 bits of offset in one extension byte) |
| Indexed (16-bit offset) | INST oprx16,xysp | IDX2 | 16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes) |
| Indexed-Indirect (16-bit offset) | INST [opr16,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 |

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 [3.11 Instructions Using Multiple Modes](#).

3.4 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.

3.5 Inherent Addressing Mode

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

Examples:

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

3.6 Immediate Addressing Mode

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

The pound symbol (#) is used to indicate an immediate addressing mode operand. One common programming error is to accidentally omit the # symbol. This causes the assembler to misinterpret the expression that follows it as an address rather than explicitly provided data. For example, LDAA #\$55 means to load the immediate value \$55 into the A accumulator, while LDAA \$55 means to load the value from address

\$0055 into the A accumulator. Without the # symbol, the instruction is erroneously interpreted as a direct addressing mode instruction.

Examples:

| | |
|------|----------|
| LDAA | ##\$55 |
| LDX | ##\$1234 |
| LDY | ##\$67 |

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

Example:

BRSET FOO, ##\$03, THERE

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

3.7 Direct Addressing Mode

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

Example:

LDAA \$55

This is a basic example of direct addressing. The value \$55 is taken to be the low-order half of an address in the range \$0000 through \$00FF.

The high order half of the address is assumed to be 0. During execution of this instruction, the CPU combines the value \$55 from the instruction with the assumed value of \$00 to form the address \$0055, which is then used to access the data to be loaded into accumulator A.

Example:

LDX \$20

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

3.8 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 [3.11 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.

Both 8-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). The numeric range of long branch offset values is \$8000 (-32,768) to \$7FFF (32,767). If the offset is 0, the CPU executes the instruction immediately following the branch instruction, regardless of the test involved.

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

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

3.10 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

Addressing Modes

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.

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) $-256 \leq n \leq 255$ 1 = 16-bit $-32,768 \leq n \leq 65,535$ 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 $-32,768 \leq n \leq 65,535$ |
| rr1pnnnn | n,-r n,r- n,r+ | Auto pre-decrement/increment or auto post-decrement/increment; 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 |

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

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 CPU reads the address PLACE2 from memory at GO2 and jumps to PLACE2. The locations of PLACE1 through PLACE3 were known at the time of program assembly but the destination of the JMP depends upon the value in D computed during program execution.

3.11 Instructions Using Multiple Modes

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

3.11.1 Move Instructions

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

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

Move instructions do not support indirect modes, 9-bit, or 16-bit offset modes requiring extra extension bytes. There are special considerations when using PC-relative addressing with move instructions.

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, move instructions pose three special problems:

1. Some moves use an indexed source and an indexed destination.
2. 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.
3. All moves do not have the indexed postbyte as the last byte of object code.

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

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

Table 3-3. PC Offsets for MOVE Instructions

| MOVE Instruction | Addressing Modes | Offset Value |
|------------------|-----------------------|---|
| MOVB | IMM \Rightarrow IDX | +1 |
| | EXT \Rightarrow IDX | +2 |
| | IDX \Rightarrow EXT | -2 |
| | IDX \Rightarrow IDX | -1 for first operand +1 for second operand |
| MOVW | IMM \Rightarrow IDX | +2 |
| | EXT \Rightarrow IDX | +2 |
| | IDX \Rightarrow EXT | -2 |
| | IDX \Rightarrow IDX | -1 for first operand +1 for second operand |

Example:

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

Moves a byte of data from \$2000 to \$1009

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

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

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

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

3.11.2 Bit Manipulation Instructions

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

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

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

3.12 Addressing More than 64 Kbytes

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

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

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

The M68HC12 system requires no external glue logic. Bank switching overhead is reduced by implementing control logic in the MCU. Interrupts do not need to be disabled during switching because switching

tasks are incorporated in special instructions that greatly simplify program access to extended memory.

MCUs with expanded memory treat the 16 Kbytes of memory space from \$8000 to \$BFFF as a program memory window.

Expanded-memory devices also have 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.

Refer to [Section 10. Memory Expansion](#) for a detailed discussion of memory expansion.

Addressing Modes

Section 4. Instruction Queue

4.1 Contents

| | | |
|---------|--|----|
| 4.2 | Introduction | 55 |
| 4.3 | Queue Description | 56 |
| 4.4 | Data Movement in the Queue. | 57 |
| 4.4.1 | No Movement. | 57 |
| 4.4.2 | Latch Data from Bus. | 57 |
| 4.4.3 | Advance and Load from Data Bus | 57 |
| 4.4.4 | Advance and Load from Buffer. | 57 |
| 4.5 | Changes in Execution Flow | 58 |
| 4.5.1 | Exceptions | 58 |
| 4.5.2 | Subroutines | 59 |
| 4.5.3 | Branches | 60 |
| 4.5.3.1 | Short Branches | 60 |
| 4.5.3.2 | Long Branches. | 61 |
| 4.5.3.3 | Bit Condition Branches | 62 |
| 4.5.3.4 | Loop Primitives | 62 |
| 4.5.4 | Jumps. | 62 |

4.2 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. Development and Debug Support](#)

contains detailed information about tracking queue operation and instruction execution.

4.3 Queue Description

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

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

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

Queue logic manages the position of program information so that the CPU itself does not deal with alignment. As it is executed, each instruction initiates at least enough program word fetches to replace its own object code in the queue.

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

Two external pins, IPIPE[1:0], provide time-multiplexed information about data movement in the queue and instruction execution. Decoding and use of these signals is discussed in [Section 8. Development and Debug Support](#).

4.4 Data Movement in the Queue

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

4.4.1 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.

4.4.2 Latch Data from Bus

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

4.4.3 Advance and Load from Data Bus

The content of queue stage 1 advances to stage 2, 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.

4.4.4 Advance and Load from Buffer

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

4.5 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.

4.5.1 Exceptions

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

1. Unimplemented opcode trap
2. Software interrupt instruction
3. X-bit interrupts
4. I-bit interrupts

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

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

4.5.2 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. The first two words fetched are queued during the second and third cycles of the sequence. The third fetch cycle is performed in anticipation of a queue advance, which may occur during the fourth cycle of the sequence. If the queue is not yet ready to advance at that time, the third word of program information is held in the buffer.

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.

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.

4.5.3 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

5.20 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:

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

4.5.3.1 Short Branches

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

The “taken” case for short branches requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is calculated using the relative offset in the instruction. Then, the address is loaded into the program counter, and the CPU performs three program word fetches at the new address. The first two words fetched are loaded into the instruction queue during the second and third cycles of the sequence. The third fetch cycle is performed in anticipation of a queue advance, which may occur during the first cycle of the next instruction. If the queue is not yet ready to advance at that time, the third word of program information is held in the buffer.

4.5.3.2 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. Optional cycles are always performed, but serve different purposes determined by instruction alignment. 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 aligned with an even byte boundary, an optional cycle is used to make an additional program word access that maintains queue order. In all other cases, the optional cycle appears as a free cycle.

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

In the “taken” case, the effective address of the branch is calculated using the 16-bit relative offset contained in the second word of the instruction. This address is loaded into the program counter, then the CPU performs three program word fetches at the new address. The first two words fetched are loaded into the instruction queue during the second and third cycles of the sequence. The third fetch cycle is performed in anticipation of a queue advance, which may occur during the first cycle of the next instruction. If the queue is not yet ready to advance, the third word of program information is held in the buffer.

4.5.3.3 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 overwritten by subsequent fetches in the “not-taken” case.

4.5.3.4 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 pre-condition is met. There are auto-increment and auto-decrement versions of the instructions. The test and increment/decrement operations are performed on internal CPU registers, and require no additional program information. To shorten execution time, these branches perform one program word fetch in anticipation of the “taken” case. The data from this fetch is overwritten by subsequent fetches in the “not-taken” case. The “taken” case performs two additional program word fetches at the new address. 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.

4.5.4 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. The first two words fetched are loaded into the instruction queue during the second and third cycles of the sequence. The third fetch cycle is performed in anticipation of a queue advance, which may occur during the first cycle of the next instruction. If the queue is not yet ready to advance, the third word of program information is held in the buffer.

Section 5. Instruction Set Overview

5.1 Contents

| | | |
|--------|--|----|
| 5.2 | Introduction | 64 |
| 5.3 | Instruction Set Description | 65 |
| 5.4 | Load and Store Instructions | 66 |
| 5.5 | Transfer and Exchange Instructions | 67 |
| 5.6 | Move Instructions | 68 |
| 5.7 | Addition and Subtraction Instructions | 69 |
| 5.8 | Binary-Coded Decimal Instructions | 70 |
| 5.9 | Decrement and Increment Instructions | 71 |
| 5.10 | Compare and Test Instructions | 72 |
| 5.11 | Boolean Logic Instructions | 73 |
| 5.12 | Clear, Complement, and Negate Instructions | 74 |
| 5.13 | Multiplication and Division Instructions | 75 |
| 5.14 | Bit Test and Manipulation Instructions | 76 |
| 5.15 | Shift and Rotate Instructions | 77 |
| 5.16 | Fuzzy Logic Instructions | 78 |
| 5.16.1 | Fuzzy Logic Membership Instruction | 78 |
| 5.16.2 | Fuzzy Logic Rule Evaluation Instructions | 78 |
| 5.16.3 | Fuzzy Logic Averaging Instruction | 79 |
| 5.17 | Maximum and Minimum Instructions | 81 |
| 5.18 | Multiply and Accumulate Instruction | 82 |
| 5.19 | Table Interpolation Instructions | 82 |
| 5.20 | Branch Instructions | 83 |
| 5.20.1 | Short Branch Instructions | 84 |
| 5.20.2 | Long Branch Instructions | 85 |
| 5.20.3 | Bit Condition Branch Instructions | 86 |

| | | |
|------|--|----|
| 5.21 | Loop Primitive Instructions | 87 |
| 5.22 | Jump and Subroutine Instructions | 88 |
| 5.23 | Interrupt Instructions | 89 |
| 5.24 | Index Manipulation Instructions | 91 |
| 5.25 | Stacking Instructions. | 92 |
| 5.26 | Pointer and Index Calculation Instructions | 93 |
| 5.27 | Condition Code Instructions | 94 |
| 5.28 | Stop and Wait Instructions | 95 |
| 5.29 | Background Mode and Null Operations | 96 |

5.2 Introduction

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

5.3 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.

In the M68HC12 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.

5.4 Load and Store Instructions

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

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

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

Table 5-1. Load and Store Instructions

| Mnemonic | Function | Operation |
|---------------------------|--------------------------------|--|
| Load Instructions | | |
| LDAA | Load A | $(M) \Rightarrow A$ |
| LDAB | Load B | $(M) \Rightarrow B$ |
| LDI | Load D | $(M : M + 1) \Rightarrow (A:B)$ |
| LDS | Load SP | $(M : M + 1) \Rightarrow SP$ |
| LDX | Load index register X | $(M : M + 1) \Rightarrow X$ |
| LDY | Load index register Y | $(M : M + 1) \Rightarrow Y$ |
| LEAS | Load effective address into SP | Effective address $\Rightarrow SP$ |
| LEAX | Load effective address into X | Effective address $\Rightarrow X$ |
| LEAY | Load effective address into Y | Effective address $\Rightarrow Y$ |
| Store Instructions | | |
| STAA | Store A | $(A) \Rightarrow M$ |
| STAB | Store B | $(B) \Rightarrow M$ |
| STD | Store D | $(A) \Rightarrow M, (B) \Rightarrow M + 1$ |
| STS | Store SP | $(SP) \Rightarrow M : M + 1$ |
| STX | Store X | $(X) \Rightarrow M : M + 1$ |
| STY | Store Y | $(Y) \Rightarrow M : M + 1$ |

5.5 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.

Exchange instructions exchange the contents of pairs of registers or accumulators.

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.

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) \Rightarrow B |
| TAP | Transfer A to CCR | (A) \Rightarrow CCR |
| TBA | Transfer B to A | (B) \Rightarrow A |
| TFR | Transfer register to register | (A, B, CCR, D, X, Y, or SP) \Rightarrow A, B, CCR, D, X, Y, or SP |
| TPA | Transfer CCR to A | (CCR) \Rightarrow A |
| TSX | Transfer SP to X | (SP) \Rightarrow X |
| TSY | Transfer SP to Y | (SP) \Rightarrow Y |
| TXS | Transfer X to SP | (X) \Rightarrow SP |
| TYS | Transfer Y to SP | (Y) \Rightarrow SP |

Table 5-2. Transfer and Exchange Instructions (Continued)

| Mnemonic | Function | Operation |
|-----------------------------------|-------------------------------|---|
| Exchange Instructions | | |
| EXG | Exchange register to register | $(A, B, CCR, D, X, Y, \text{ or } SP) \Leftrightarrow (A, B, CCR, D, X, Y, \text{ or } SP)$ |
| XGDX | Exchange D with X | $(D) \Leftrightarrow (X)$ |
| XGDY | Exchange D with Y | $(D) \Leftrightarrow (Y)$ |
| Sign Extension Instruction | | |
| SEX | Sign extend 8-Bit operand | $(A, B, CCR) \Rightarrow D, X, Y, \text{ or } SP$ |

5.6 Move Instructions

Move instructions move data bytes or words from a source ($M_1, M : M + 1_1$) to a destination ($M_2, M : M + 1_2$) in memory. Six combinations of immediate, extended, and indexed addressing are allowed to specify source and destination addresses (IMM \Rightarrow EXT, IMM \Rightarrow IDX, EXT \Rightarrow EXT, EXT \Rightarrow IDX, IDX \Rightarrow EXT, IDX \Rightarrow IDX).

Table 5-3 shows byte and word move instructions.

Table 5-3. Move Instructions

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

5.7 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 condition code register (CCR) carry bit 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 CCR carry bit facilitate multiple precision computation. Refer to [Table 5-4](#) for addition and subtraction instructions.

Table 5-4. Addition and Subtraction Instructions

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

5.8 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 | $(A) + (B) \Rightarrow A$ |
| ADCA | Add with carry to A | $(A) + (M) + C \Rightarrow A$ |
| ADCB | Add with carry to B | $(B) + (M) + C \Rightarrow B$ |
| ADDA | Add memory to A | $(A) + (M) \Rightarrow A$ |
| ADDB | Add memory to B | $(B) + (M) \Rightarrow B$ |
| DAA | Decimal adjust A | $(A)_{10}$ |

5.9 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 [5.21 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 | $(M) - \$01 \Rightarrow M$ |
| DECA | Decrement A | $(A) - \$01 \Rightarrow A$ |
| DECB | Decrement B | $(B) - \$01 \Rightarrow B$ |
| DES | Decrement SP | $(SP) - \$0001 \Rightarrow SP$ |
| DEX | Decrement X | $(X) - \$0001 \Rightarrow X$ |
| DEY | Decrement Y | $(Y) - \$0001 \Rightarrow Y$ |
| Increment Instructions | | |
| INC | Increment memory | $(M) + \$01 \Rightarrow M$ |
| INCA | Increment A | $(A) + \$01 \Rightarrow A$ |
| INCB | Increment B | $(B) + \$01 \Rightarrow B$ |
| INS | Increment SP | $(SP) + \$0001 \Rightarrow SP$ |
| INX | Increment X | $(X) + \$0001 \Rightarrow X$ |
| INY | Increment Y | $(Y) + \$0001 \Rightarrow Y$ |

5.10 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 |

5.11 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 | $(A) \bullet (M) \Rightarrow A$ |
| ANDB | AND B with memory | $(B) \bullet (M) \Rightarrow B$ |
| ANDCC | AND CCR with memory (clear CCR bits) | $(CCR) \bullet (M) \Rightarrow CCR$ |
| EORA | Exclusive OR A with memory | $(A) \oplus (M) \Rightarrow A$ |
| EORB | Exclusive OR B with memory | $(B) \oplus (M) \Rightarrow B$ |
| ORAA | OR A with memory | $(A) + (M) \Rightarrow A$ |
| ORAB | OR B with memory | $(B) + (M) \Rightarrow B$ |
| ORCC | OR CCR with memory (set CCR bits) | $(CCR) + (M) \Rightarrow CCR$ |

5.12 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 | $\$FF - (M) \Rightarrow M \text{ or } (\bar{M}) \Rightarrow M$ |
| COMA | One's complement A | $\$FF - (A) \Rightarrow A \text{ or } (\bar{A}) \Rightarrow A$ |
| COMB | One's complement B | $\$FF - (B) \Rightarrow B \text{ or } (\bar{B}) \Rightarrow B$ |
| NEG | Two's complement memory | $\$00 - (M) \Rightarrow M \text{ or } (\bar{M}) + 1 \Rightarrow M$ |
| NEGA | Two's complement A | $\$00 - (A) \Rightarrow A \text{ or } (\bar{A}) + 1 \Rightarrow A$ |
| NEGB | Two's complement B | $\$00 - (B) \Rightarrow B \text{ or } (\bar{B}) + 1 \Rightarrow B$ |

5.13 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) | $(D) \times (Y) \Rightarrow Y : D$ |
| EMULS | 16 by 16 multiply (signed) | $(D) \times (Y) \Rightarrow Y : D$ |
| MUL | 8 by 8 multiply (unsigned) | $(A) \times (B) \Rightarrow A : B$ |
| Division Instructions | | |
| EDIV | 32 by 16 divide (unsigned) | $(Y : D) \div (X)$ Quotient $\Rightarrow Y$ Remainder $\Rightarrow D$ |
| EDIVS | 32 by 16 divide (signed) | $(Y : D) \div (X)$ Quotient $\Rightarrow Y$ Remainder $\Rightarrow D$ |
| FDIV | 16 by 16 fractional divide | $(D) \div (X) \Rightarrow X$ Remainder $\Rightarrow D$ |
| IDIV | 16 by 16 integer divide (unsigned) | $(D) \div (X) \Rightarrow X$ Remainder $\Rightarrow D$ |
| IDIVS | 16 by 16 integer divide (signed) | $(D) \div (X) \Rightarrow X$ Remainder $\Rightarrow D$ |

5.14 Bit Test and Manipulation Instructions

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

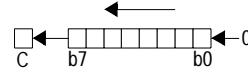
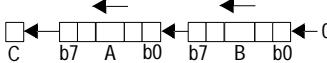
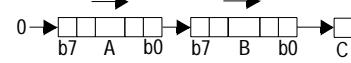
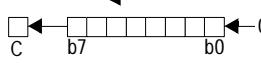
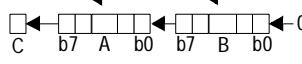
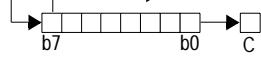
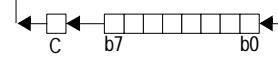
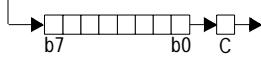
Table 5-11. Bit Test and Manipulation Instructions

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

5.15 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 memory (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 |  |

5.16 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.

5.16.1 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.

5.16.2 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.

5.16.3 Fuzzy Logic Averaging Instruction

The calculate numerator (WAV) instruction provides a facility for weighted average calculations. 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 WAVER 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 | $\mu \text{ (grade)} \Rightarrow M_{(Y)}$ $(X) + 4 \Rightarrow X; (Y) + 1 \Rightarrow Y; A \text{ unchanged}$ if $(A) < P1$ or $(A) > P2$, then $\mu = 0$, else $\mu = \text{MIN} [((A) - P1) \times S1, (P2 - (A)) \times S2, \$FF]$ where: A = current crisp input value X points to a 4-byte data structure that describes a trapezoidal membership function as base intercept points and slopes (P1, P2, S1, S2) Y points at fuzzy input (RAM location) See instruction details for special cases |

Table 5-13. Fuzzy Logic Instructions (Continued)

| Mnemonic | Function | Operation |
|----------|--|--|
| REV | MIN-MAX rule evaluation | <p>Find smallest rule input (MIN) Store to rule outputs unless fuzzy output is larger (MAX)</p> <p>Rules are unweighted</p> <p>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</p> <p>REV can be interrupted</p> |
| REWW | MIN-MAX rule evaluation | <p>Find smallest rule input (MIN) Multiply by a rule weighting factor (optional) Store to rule outputs unless fuzzy output is larger (MAX)</p> <p>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</p> <p>REWW can be interrupted</p> |
| WAV | <p>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</p> | $\sum_{i=1}^B S_i F_i \Rightarrow Y:D$ $\sum_{i=1}^B F_i \Rightarrow X$ |
| WAVER | Resumes execution of interrupted WAV instruction | Recover immediate results from stack rather than initializing them to 0. |

5.17 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)) \Rightarrow D |
| EMINM | MIN of two unsigned 16-bit values result to memory | MIN ((D), (M : M + 1)) \Rightarrow M : M+1 |
| MINA | MIN of two unsigned 8-bit values result to accumulator | MIN ((A), (M)) \Rightarrow A |
| MINM | MIN of two unsigned 8-bit values result to memory | MIN ((A), (M)) \Rightarrow M |
| Maximum Instructions | | |
| EMAXD | MAX of two unsigned 16-bit values result to accumulator | MAX ((D), (M : M + 1)) \Rightarrow D |
| EMAXM | MAX of two unsigned 16-bit values result to memory | MAX ((D), (M : M + 1)) \Rightarrow M : M + 1 |
| MAXA | MAX of two unsigned 8-bit values result to accumulator | MAX ((A), (M)) \Rightarrow A |
| MAXM | MAX of two unsigned 8-bit values result to memory | MAX ((A), (M)) \Rightarrow M |

5.18 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 × 16 bit ⇒ 32 bit | $((M_{(X)}:M_{(X+1)}) \times (M_{(Y)}:M_{(Y+1)})) + (M - M + 3) \Rightarrow M - M + 3$ |

5.19 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 change in x from the beginning of the line segment to the lookup point divided by total change in x from the beginning to the end of the line segment. B is treated as an 8-bit binary fraction with radix point left of the MSB, so each line segment is effectively divided into 256 smaller segments. During instruction execution, the change in y between the beginning and end of the segment (a signed byte for TBL or a signed word for ETBL) is multiplied by the content of the B accumulator to obtain an intermediate delta-y term. The result (stored in the A accumulator by TBL, and in the D

accumulator by ETBL) is the y-value of the beginning point plus the signed intermediate delta-y value. **Table 5-16** shows the table interpolation instructions.

Table 5-16. Table Interpolation Instructions

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

5.20 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.

5.20.1 Short Branch Instructions

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

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

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

Table 5-17. Short Branch Instructions

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

5.20.2 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 map.

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

Table 5-18. Long Branch Instructions

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

5.20.3 Bit Condition Branch Instructions

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

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

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

Table 5-19. Bit Condition Branch Instructions

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

5.21 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 8-bit offset values is \$80 (-128) to \$7F (127) 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 \Rightarrow counter If (counter) = 0, then branch; else continue to next instruction |
| DBNE | Decrement counter and branch if \neq 0 (counter = A, B, D, X, Y, or SP) | (counter) – 1 \Rightarrow 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 \Rightarrow counter If (counter) = 0, then branch; else continue to next instruction |
| IBNE | Increment counter and branch if \neq 0 (counter = A, B, D, X, Y, or SP) | (counter) + 1 \Rightarrow 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 \neq 0 (counter = A, B, D, X, Y, or SP) | If (counter) not = 0, then branch; else continue to next instruction |

5.22 Jump and Subroutine Instructions

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

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

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

Table 5-21. Jump and Subroutine Instructions

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

5.23 Interrupt Instructions

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

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

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

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

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

Table 5-22 is a summary of interrupt instructions.

Table 5-22. Interrupt Instructions

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

5.24 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 | $(B) + (X) \Rightarrow X$ |
| ABY | Add B to Y | $(B) + (Y) \Rightarrow Y$ |
| 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 | $M : M+1 \Rightarrow SP$ |
| LDX | Load X from memory | $(M : M + 1) \Rightarrow X$ |
| LDY | Load Y from memory | $(M : M + 1) \Rightarrow Y$ |
| LEAS | Load effective address into SP | Effective address $\Rightarrow SP$ |
| LEAX | Load effective address into X | Effective address $\Rightarrow X$ |
| LEAY | Load effective address into Y | Effective address $\Rightarrow Y$ |
| Store Instructions | | |
| STS | Store SP in memory | $(SP) \Rightarrow M:M+1$ |
| STX | Store X in memory | $(X) \Rightarrow M : M + 1$ |
| STY | Store Y in memory | $(Y) \Rightarrow M : M + 1$ |
| Transfer Instructions | | |
| TFR | Transfer register to register | $(A, B, CCR, D, X, Y, \text{ or } SP) \Rightarrow A, B, CCR, D, X, Y, \text{ or } SP$ |
| TSX | Transfer SP to X | $(SP) \Rightarrow X$ |
| TSY | Transfer SP to Y | $(SP) \Rightarrow Y$ |
| TXS | transfer X to SP | $(X) \Rightarrow SP$ |
| TYS | transfer Y to SP | $(Y) \Rightarrow SP$ |
| Exchange Instructions | | |
| EXG | Exchange register to register | $(A, B, CCR, D, X, Y, \text{ or } SP) \Leftrightarrow (A, B, CCR, D, X, Y, \text{ or } SP)$ |
| XGDX | EXchange D with X | $(D) \Leftrightarrow (X)$ |
| XGDY | EXchange D with Y | $(D) \Leftrightarrow (Y)$ |

5.25 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 | $(SP) - 1 \Rightarrow SP$ |
| INS | Increment SP | $(SP) + 1 \Rightarrow SP$ |
| LDS | Load SP | $(M : M + 1) \Rightarrow SP$ |
| LEAS | Load effective address into SP | Effective address $\Rightarrow SP$ |
| STS | Store SP | $(SP) \Rightarrow M : M + 1$ |
| TSX | Transfer SP to X | $(SP) \Rightarrow X$ |
| TSY | Transfer SP to Y | $(SP) \Rightarrow Y$ |
| TXS | Transfer X to SP | $(X) \Rightarrow SP$ |
| TYs | Transfer Y to SP | $(Y) \Rightarrow SP$ |
| Stack Operation Instructions | | |
| PSHA | Push A | $(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$ |
| PSHB | Push B | $(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$ |
| PSHC | Push CCR | $(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$ |
| PSHD | Push D | $(SP) - 2 \Rightarrow SP; (A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PSHX | Push X | $(SP) - 2 \Rightarrow SP; (X) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PSHY | Push Y | $(SP) - 2 \Rightarrow SP; (Y) \Rightarrow M_{(SP)} : M_{(SP+1)}$ |
| PULA | Pull A | $(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$ |
| PULB | Pull B | $(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$ |
| PULC | Pull CCR | $(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ |
| PULD | Pull D | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B; (SP) + 2 \Rightarrow SP$ |
| PULX | Pull X | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X; (SP) + 2 \Rightarrow SP$ |
| PULY | Pull Y | $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y; (SP) + 2 \Rightarrow SP$ |

5.26 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, the SP, or the PC. **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 | $r \pm \text{constant} \Rightarrow \text{SP}$ or $(r) + (\text{accumulator}) \Rightarrow \text{SP}$ $r = X, Y, \text{SP}, \text{or PC}$ |
| LEAX | Load result of indexed addressing mode effective address calculation into x index register | $r \pm \text{constant} \Rightarrow X$ or $(r) + (\text{accumulator}) \Rightarrow X$ $r = X, Y, \text{SP}, \text{or PC}$ |
| LEAY | Load result of indexed addressing mode effective address calculation into y index register | $r \pm \text{constant} \Rightarrow Y$ or $(r) + (\text{accumulator}) \Rightarrow Y$ $r = X, Y, \text{SP}, \text{or PC}$ |

5.27 Condition Code Instructions

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

Table 5-26. Condition Code Instructions

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

5.28 Stop and Wait Instructions

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

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

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

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

Table 5-27. Stop and Wait Instructions

| Mnemonic | Function | Operation |
|----------|--------------------|--|
| STOP | Stop | $SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$ Stop CPU clocks |
| WAI | Wait for interrupt | $SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 1 \Rightarrow SP; CCR \Rightarrow M_{(SP)}$ |

5.29 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. Development and Debug Support](#).

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 without actually taking the branches.

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

Table 5-28. Background Mode and Null Operation Instructions

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

Section 6. Instruction Glossary

6.1 Contents

| | | |
|-----|------------------------------------|-----|
| 6.2 | Introduction | 97 |
| 6.3 | Glossary Information | 98 |
| 6.4 | Condition Code Changes | 99 |
| 6.5 | Object Code Notation | 100 |
| 6.6 | Source Forms | 101 |
| 6.7 | Cycle-by-Cycle Execution | 104 |
| 6.8 | Glossary | 109 |

6.2 Introduction

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

6.3 Glossary Information

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

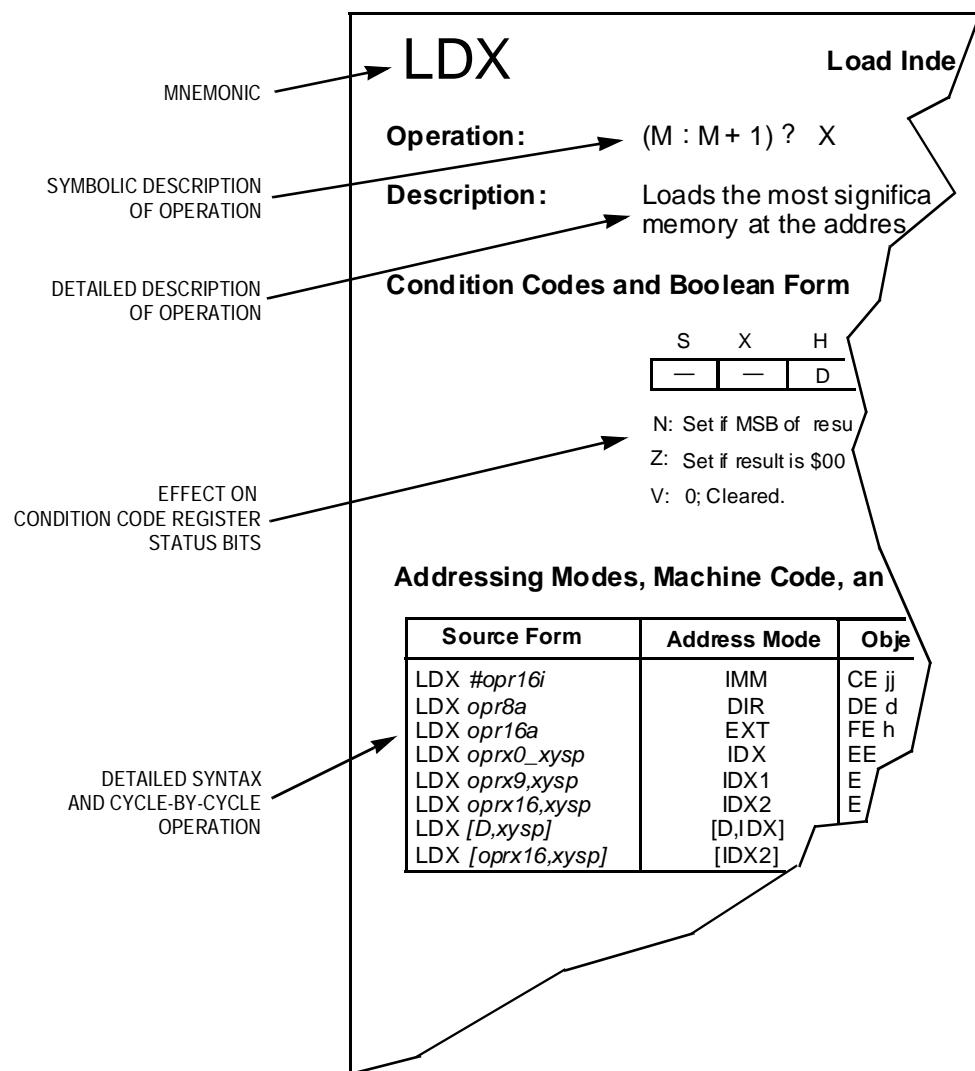


Figure 6-1. Example Glossary Page

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, cycle count, and cycle-by-cycle execution of the instruction.

6.4 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

6.5 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

6.6 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 *opr0_xysp* are both valid, but the two groups *opr0 xysp* are not valid because there is a space between them. Permitted syntax is described here.

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

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

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

instructions, but these forms are intended for 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

- opr_{x3}* — Any label or expression that evaluates to a value in the range +1 to +8
- opr_{x5}* — Any label or expression that evaluates to a 5-bit value in the range -16 to +15
- opr_{x9}* — Any label or expression that evaluates to a 9-bit value in the range -256 to +255
- opr_{x16}* — 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 -256 to +255 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 -512 to +511 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.

6.7 Cycle-by-Cycle Execution

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

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

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

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

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

- g — Read 8-bit PPAGE register. These cycles are used only with the CALL instruction to read the current value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.
- I — Read indirect pointer. Indexed indirect instructions use this 16-bit pointer from memory to address the operand for the instruction. These are always 16-bit reads but they can be either aligned or misaligned. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned access to a memory that is not designed for single-cycle misaligned access.
- i — Read indirect PPAGE value. These cycles are only used with indexed indirect versions of the CALL instruction, where the 8-bit value for the memory expansion page register of the CALL destination is fetched from an indirect memory location. These cycles are stretched only when controlled by a chip-select circuit that is programmed for slow memory.
- n — Write 8-bit PPAGE register. These cycles are used only with the CALL and RTC instructions to write the destination value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.

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

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

- U — Unstack 16-bit data. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the SP is pointing to external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single-cycle misaligned access. The internal RAM is designed to allow single-cycle misaligned word access.
- V — Vector fetch. Vectors are always aligned 16-bit words. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory.
- t — 8-bit conditional read. These cycles are either data read cycles or unused cycles, depending on the data and flow of the REVW instruction. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.
- T — 16-bit conditional read. These cycles are either data read cycles or free cycles, depending on the data and flow of the REV or REVW instruction. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single-cycle misaligned access.
- x — 8-bit conditional write. These cycles are either data write cycles or free cycles, depending on the data and flow of the REV or REVW instruction. These cycles are only stretched when controlled by a chip-select circuit programmed for slow memory.

Special Notation for Branch Taken/Not Taken Cases

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

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

6.8 Glossary

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

ABA

Add Accumulator B to Accumulator A

ABA

Operation: $(A) + (B) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $A7 \bullet B7 \bullet \overline{R7} + \overline{A7} \bullet B7 \bullet R7$

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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ABA | INH | 18 06 | 2 | 00 |

ABX

Add Accumulator B to Index Register X

ABX

Operation: $(B) + (X) \Rightarrow X$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|-------------------|
| ABX translates to... LEAX B,X | IDX | 1A E5 | 2 | PP ⁽¹⁾ |

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

ABY

Add Accumulator B to Index Register Y

ABY

Operation: $(B) + (Y) \Rightarrow Y$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|-------------------|
| ABY translates to... LEAY B,Y | IDX | 19 ED | 2 | PP ⁽¹⁾ |

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

ADCA

Add with Carry to A

ADCA

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

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|----------|---|----------|----------|----------|----------|
| - | - | Δ | - | Δ | Δ | Δ | Δ |

H: $X_3 \bullet M_3 + M_3 \bullet \overline{R}_3 + \overline{R}_3 \bullet X_3$

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: $X_7 \bullet M_7 \bullet \overline{R}_7 + \overline{X}_7 \bullet \overline{M}_7 \bullet R_7$

Set if two's complement overflow resulted from the operation; cleared otherwise

C: $X_7 \bullet M_7 + M_7 \bullet \overline{R}_7 + \overline{R}_7 \bullet X_7$

Set if there was a carry from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| ADCA #opr8 <i>i</i> | IMM | 89 ii | 1 | P |
| ADCA opr8 <i>a</i> | DIR | 99 dd | 3 | rFP |
| ADCA opr16 <i>a</i> | EXT | B9 hh ll | 3 | rOP |
| ADCA oprx0_xy <i>sp</i> | IDX | A9 xb | 3 | rFP |
| ADCA oprx9_xy <i>sp</i> | IDX1 | A9 xb ff | 3 | rPO |
| ADCA oprx16_xy <i>sp</i> | IDX2 | A9 xb ee ff | 4 | fPP |
| ADCA [D,xy <i>sp</i>] | [D,IDX] | A9 xb | 6 | fIfPfP |
| ADCA [opr16,xy <i>sp</i>] | [IDX2] | A9 xb ee ff | 6 | fIPfP |

ADCB

Add with Carry to B

ADCB**Operation:** $(B) + (M) + C \Rightarrow B$ **Description:** Adds the content of accumulator B to the content of memory location M, then adds the value of the C bit and places the result in B. This instruction affects the H status bit, so it is suitable for use in BCD arithmetic operations. See **DAA** instruction for additional information.**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet M7 \bullet \overline{R7} + \overline{X7} \bullet M7 \bullet R7$

Set if two's complement overflow resulted from the operation; cleared otherwise

C: $X7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet X7$

Set if there was a carry from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| ADCB #opr8i | IMM | C9 ii | 1 | P |
| ADCB opr8a | DIR | D9 dd | 3 | rFP |
| ADCB opr16a | EXT | F9 hh ll | 3 | rOP |
| ADCB oprx0_xySp | IDX | E9 xb | 3 | rFP |
| ADCB oprx9_xySp | IDX1 | E9 xb ff | 3 | rPO |
| ADCB oprx16_xySp | IDX2 | E9 xb ee ff | 4 | fRP |
| ADCB [D,xySp] | [D,IDX] | E9 xb | 6 | fIFrFP |
| ADCB [opr16_xySp] | [IDX2] | E9 xb ee ff | 6 | fIPrFP |

ADDA

Add without Carry to A

ADDA

Operation: $(A) + (M) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|----------|---|----------|----------|----------|----------|
| - | - | Δ | - | Δ | Δ | Δ | Δ |

H: $X_3 \bullet M_3 + M_3 \bullet \overline{R_3} + \overline{R_3} \bullet X_3$

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: $X_7 \bullet M_7 \bullet \overline{R_7} + \overline{X_7} \bullet M_7 \bullet R_7$

Set if two's complement overflow resulted from the operation; cleared otherwise

C: $X_7 \bullet M_7 + M_7 \bullet \overline{R_7} + \overline{R_7} \bullet X_7$

Set if there was a carry from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| ADDA #opr8 <i>i</i> | IMM | 8B ii | 1 | P |
| ADDA opr8 <i>a</i> | DIR | 9B dd | 3 | rFP |
| ADDA opr16 <i>a</i> | EXT | BB hh ll | 3 | rOP |
| ADDA oprx0_xy <i>sp</i> | IDX | AB xb | 3 | rFP |
| ADDA oprx9_xy <i>sp</i> | IDX1 | AB xb ff | 3 | rPO |
| ADDA oprx16_xy <i>sp</i> | IDX2 | AB xb ee ff | 4 | fRP |
| ADDA [D,xy <i>sp</i>] | [D,IDX] | AB xb | 6 | fIFrFP |
| ADDA [opr16_xy <i>sp</i>] | [IDX2] | AB xb ee ff | 6 | fIPrFP |

ADDB

Add without Carry to B

ADDB**Operation:** $(B) + (M) \Rightarrow B$ **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.**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|----------|---|----------|----------|----------|----------|
| - | - | Δ | - | Δ | Δ | Δ | Δ |

H: $X_3 \bullet M_3 + M_3 \bullet \overline{R_3} + \overline{R_3} \bullet X_3$

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: $X_7 \bullet M_7 \bullet \overline{R_7} + \overline{X_7} \bullet M_7 \bullet R_7$

Set if two's complement overflow resulted from the operation; cleared otherwise

C: $X_7 \bullet M_7 + M_7 \bullet \overline{R_7} + \overline{R_7} \bullet X_7$

Set if there was a carry from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| ADDB #opr8 <i>i</i> | IMM | CB ii | 1 | P |
| ADDB opr8 <i>a</i> | DIR | DB dd | 3 | rFP |
| ADDB opr16 <i>a</i> | EXT | FB hh ll | 3 | rOP |
| ADDB oprx0_xy <i>sp</i> | IDX | EB xb | 3 | rFP |
| ADDB oprx9_xy <i>sp</i> | IDX1 | EB xb ff | 3 | rPO |
| ADDB oprx16_xy <i>sp</i> | IDX2 | EB xb ee ff | 4 | fRP |
| ADDB [D,xy <i>sp</i>] | [D,IDX] | EB xb | 6 | fIFrFP |
| ADDB [opr16_xy <i>sp</i>] | [IDX2] | EB xb ee ff | 6 | fIPrFP |

ADDD

Add Double Accumulator

ADDD

Operation: $(A : B) + (M : M+1) \Rightarrow A : B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $D15 \bullet M15 \bullet R15 + \overline{D15} \bullet \overline{M15} \bullet R15$

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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| ADDD #opr16i | IMM | C3 jj kk | 2 | OP |
| ADDD opr8a | DIR | D3 dd | 3 | RfP |
| ADDD opr16a | EXT | F3 hh ll | 3 | ROP |
| ADDD oprx0_xysp | IDX | E3 xb | 3 | RfP |
| ADDD oprx9,xysp | IDX1 | E3 xb ff | 3 | RPO |
| ADDD oprx16,xysp | IDX2 | E3 xb ee ff | 4 | fRPP |
| ADDD [D,xysp] | [D,IDX] | E3 xb | 6 | fIfRfP |
| ADDD [opr16,xysp] | [IDX2] | E3 xb ee ff | 6 | fIPRfP |

ANDA

Logical AND A

ANDA

Operation: $(A) \bullet (M) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| ANDA #opr8 <i>i</i> | IMM | 84 ii | 1 | P |
| ANDA opr8 <i>a</i> | DIR | 94 dd | 3 | rP |
| ANDA opr16 <i>a</i> | EXT | B4 hh 11 | 3 | rOP |
| ANDA oprx0_xysp | IDX | A4 xb | 3 | rP |
| ANDA oprx9,xysp | IDX1 | A4 xb ff | 3 | rPO |
| ANDA oprx16,xysp | IDX2 | A4 xb ee ff | 4 | fPP |
| ANDA [D,xysp] | [D,IDX] | A4 xb | 6 | fIfPfP |
| ANDA [opr16,xysp] | [IDX2] | A4 xb ee ff | 6 | fIPrfP |

ANDB

Logical AND B

ANDB

Operation: $(B) \bullet (M) \Rightarrow B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| ANDB #opr8 <i>i</i> | IMM | C4 ii | 1 | P |
| ANDB opr8 <i>a</i> | DIR | D4 dd | 3 | rP |
| ANDB opr16 <i>a</i> | EXT | F4 hh ll | 3 | rOP |
| ANDB oprx0_xysp | IDX | E4 xb | 3 | rP |
| ANDB oprx9,xysp | IDX1 | E4 xb ff | 3 | rPO |
| ANDB oprx16,xysp | IDX2 | E4 xb ee ff | 4 | fPP |
| ANDB [D,xysp] | [D,IDX] | E4 xb | 6 | fIPrP |
| ANDB [opr16,xysp] | [IDX2] | E4 xb ee ff | 6 | fIPrP |

ANDCC

Logical AND CCR with Mask

ANDCC

Operation: (CCR) • (Mask) \Rightarrow 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).

**Condition Codes
and Boolean
Formulas:**

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

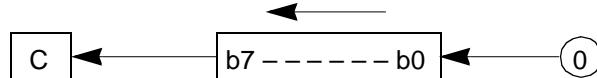
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------|--------------|-------------|--------|---------------|
| ANDCC #opr8i | IMM | 10 ii | 1 | P |

ASL

**Arithmetic Shift Left Memory
(same as LSL)**

ASL

Operation:



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

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

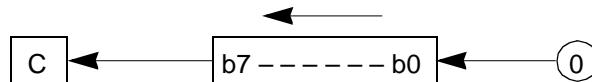
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------|--------------|-------------|--------|---------------|
| ASL opr16a | EXT | 78 hh 11 | 4 | rOPw |
| ASL oprx0_xy | IDX | 68 xb | 3 | rPw |
| ASL oprx9_xy | IDX1 | 68 xb ff | 4 | rPOw |
| ASL oprx16_xy | IDX2 | 68 xb ee ff | 5 | fRPw |
| ASL [D,xy] | [D,IDX] | 68 xb | 6 | fIfnPw |
| ASL [opr16_xy] | [IDX2] | 68 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

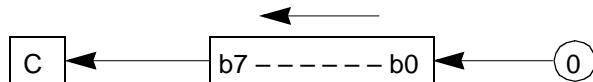
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ASLA | INH | 48 | 1 | 0 |

ASLB

Arithmetic Shift Left B (same as LSB)

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.

Condition Codes and Boolean Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

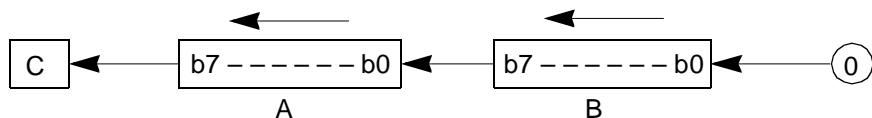
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ASLB | INH | 58 | 1 | 0 |

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

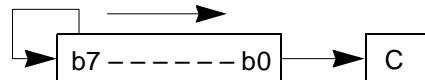
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ASLD | INH | 59 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

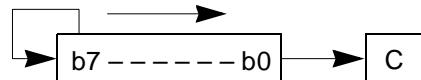
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0

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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| ASR opr16a | EXT | 77 hh 11 | 4 | rOPw |
| ASR oprx0_xysp | IDX | 67 xb | 3 | rPw |
| ASR oprx9_xysp | IDX1 | 67 xb ff | 4 | rPOw |
| ASR oprx16_xysp | IDX2 | 67 xb ee ff | 5 | fRPw |
| ASR [D,xysp] | [D,IDX] | 67 xb | 6 | fIfnPw |
| ASR [opr16,xysp] | [IDX2] | 67 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

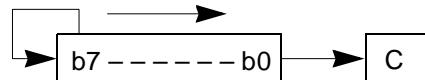
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ASRA | INH | 47 | 1 | 0 |

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ASRB | INH | 57 | 1 | 0 |

BCC

**Branch if Carry Cleared
(Same as BHS)**

BCC

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

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BCC rel8 | REL | 24 rr | 3/1 | PPP/P ⁽¹⁾ |

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

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

BCLR

Clear Bits in Memory

BCLR

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

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode ⁽¹⁾ | Object Code | Cycles | Access Detail |
|------------------------|-----------------------------|----------------|--------|---------------|
| BCLR opr8a, msk8 | DIR | 4D dd mm | 4 | rPOw |
| BCLR opr16a, msk8 | EXT | 1D hh 11 mm | 4 | rPPw |
| BCLR oprx0_xysp, msk8 | IDX | 0D xb mm | 4 | rPOw |
| BCLR oprx9,xysp, msk8 | IDX1 | 0D xb ff mm | 4 | rPwP |
| BCLR oprx16,xysp, msk8 | IDX2 | 0D xb ee ff mm | 6 | frPwOP |

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

BCS

**Branch if Carry Set
(Same as BLO)**

BCS

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

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BCS rel8 | REL | 25 rr | 3/1 | 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 |

BEQ

Branch if Equal

BEQ

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

Simple branch

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

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BEQ rel8 | REL | 27 rr | 3/1 | 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 |

BGE

Branch if Greater than or Equal to Zero

BGE

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

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

Description: If BGE 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 signed two's complement number in the accumulator is greater than or equal to the signed two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BGE rel8 | REL | 2C rr | 3/1 | 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 |

BGND

Enter Background Debug Mode

BGND

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

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

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

BDM is normally disabled to avoid accidental entry. While BDM is disabled, BGND executes as described, but the firmware causes execution to return to the user program. Refer to [Section 8.](#)

[Development and Debug Support](#) for more information concerning BDM.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| BGND | INH | 00 | 5 | 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: If BGT 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 signed two's complement number in the accumulator is greater than the signed two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BGT rel8 | REL | 2E rr | 3/1 | PPP/P ⁽¹⁾ |

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

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

BHI

Branch if Higher

BHI

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

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

Description: If BHI 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 was greater than 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BHI rel8 | REL | 22 rr | 3/1 | PPP/P ⁽¹⁾ |

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

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

BHS

**Branch if Higher or Same
(Same as BCC)**

BHS

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

For unsigned values, if (Accumulator) \geq (Memory), then branch

Description: If BHS 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 was greater than 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BHS rel8 | REL | 24 rr | 3/1 | 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 |

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| BITA #opr8 <i>i</i> | IMM | 85 ii | 1 | P |
| BITA opr8 <i>a</i> | DIR | 95 dd | 3 | rP |
| BITA opr16 <i>a</i> | EXT | B5 hh 11 | 3 | rop |
| BITA oprx0,_xysp | IDX | A5 xb | 3 | rP |
| BITA oprx9,_xysp | IDX1 | A5 xb ff | 3 | rpo |
| BITA oprx16,_xysp | IDX2 | A5 xb ee ff | 4 | fRP |
| BITA [D,_xysp] | [D,IDX] | A5 xb | 6 | fIfPfP |
| BITA [opr16,_xysp] | [IDX2] | A5 xb ee ff | 6 | fIPrfP |

BITB

Bit Test B

BITB

Operation: (B) • (M)

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| BITB #opr8 <i>i</i> | IMM | C5 ii | 1 | P |
| BITB opr8 <i>a</i> | DIR | D5 dd | 3 | rP |
| BITB opr16 <i>a</i> | EXT | F5 hh 11 | 3 | rop |
| BITB oprx0,_xysp | IDX | E5 xb | 3 | rP |
| BITB oprx9,_xysp | IDX1 | E5 xb ff | 3 | rpo |
| BITB oprx16,_xysp | IDX2 | E5 xb ee ff | 4 | fRP |
| BITB [D,_xysp] | [D,IDX] | E5 xb | 6 | fIfPfP |
| BITB [opr16,_xysp] | [IDX2] | E5 xb ee ff | 6 | fIPrfP |

BLE

Branch if Less Than or Equal to Zero

BLE

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

For signed two's complement numbers
if (Accumulator) \leq (Memory), then branch

Description: If BLE 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 two's complement number in the accumulator was less than or equal to the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|--------------|--------|----------------------|
| BLE <i>rel8</i> | REL | 2F <i>rr</i> | 3/1 | PPP/P ⁽¹⁾ |

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

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

BLO

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

BLO

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

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

Description: If BLO 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 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BLO rel8 | REL | 25 rr | 3/1 | 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 |

BLS

Branch if Lower or Same

BLS

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

For unsigned values, if (Accumulator) \leq (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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BLS rel/8 | REL | 23 rr | 3/1 | 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 |

BLT

Branch if Less than Zero

BLT

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

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

Description: If BLT 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 two's complement number in the accumulator is less than the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|--------------|--------|----------------------|
| BLT <i>rel/8</i> | REL | 2D <i>rr</i> | 3/1 | 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 |

BMI

Branch if Minus

BMI

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

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|-------------|--------|----------------------|
| BMI <i>re/8</i> | REL | 2B rr | 3/1 | 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 |

BNE

Branch if Not Equal to Zero

BNE

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

Simple branch

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

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BNE rel8 | REL | 26 rr | 3/1 | 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 |

BPL

Branch if Plus

BPL

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

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BPL rel8 | REL | 2A rr | 3/1 | 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 |

BRA

Branch Always

BRA

Operation: $(PC) + \$0002 + Rel \Rightarrow PC$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|-------------|--------|---------------|
| BRA <i>rel8</i> | REL | 20 rr | 3 | PPP |

BRCLR

Branch if Bits Cleared

BRCLR

Operation: If $(M) \bullet (\text{Mask}) = 0$, then branch

Description: Performs bitwise logical AND on 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode ⁽¹⁾ | Object Code | Cycles | Access Detail |
|-------------------------------|-----------------------------|-------------------|--------|---------------|
| BRCLR opr8a, msk8, rel8 | DIR | 4F dd mm rr | 4 | rPPP |
| BRCLR opr16a, msk8, rel8 | EXT | 1F hh 11 mm rr | 5 | rfPPP |
| BRCLR oprx0_xysp, msk8, rel8 | IDX | 0F xb mm rr | 4 | rPPP |
| BRCLR oprx9_xysp, msk8, rel8 | IDX1 | 0F xb ff mm rr | 6 | rffPPP |
| BRCLR oprx16_xysp, msk8, rel8 | IDX2 | 0F xb ee ff mm rr | 8 | frPfffPPP |

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

BRN

Branch Never

BRN

Operation: $(PC) + \$0002 \Rightarrow PC$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|-------------|--------|---------------|
| BRN <i>rel8</i> | REL | 21 rr | 1 | P |

BRSET

Branch if Bits Set

BRSET

Operation: If $(\bar{M}) \bullet (\text{Mask}) = 0$, then branch

Description: Performs bitwise logical AND on 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode ⁽¹⁾ | Object Code | Cycles | Access Detail |
|-------------------------------|-----------------------------|-------------------|--------|---------------|
| BRSET opr8a, msk8, rel8 | DIR | 4E dd mm rr | 4 | rPPP |
| BRSET opr16a, msk8, rel8 | EXT | 1E hh ll mm rr | 5 | rfPPP |
| BRSET oprx0_xySp, msk8, rel8 | IDX | 0E xb mm rr | 4 | rPPP |
| BRSET oprx9_xySp, msk8, rel8 | IDX1 | 0E xb ff mm rr | 6 | rffPPP |
| BRSET oprx16_xySp, msk8, rel8 | IDX2 | 0E xb ee ff mm rr | 8 | frPfffPPP |

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

BSET

Set Bit(s) in Memory

BSET

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

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode ⁽¹⁾ | Object Code | Cycles | Access Detail |
|------------------------|-----------------------------|----------------|--------|---------------|
| BSET opr8a, msk8 | DIR | 4C dd mm | 4 | rPOw |
| BSET opr16a, msk8 | EXT | 1C hh 11 mm | 4 | rPPw |
| BSET oprx0_xysp, msk8 | IDX | 0C xb mm | 4 | rPOw |
| BSET oprx9,xysp, msk8 | IDX1 | 0C xb ff mm | 4 | rPwP |
| BSET oprx16,xysp, msk8 | IDX2 | 0C xb ee ff mm | 6 | fPwOP |

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

BSR

Branch to Subroutine

BSR

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

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|-------------|--------|---------------|
| BSR <i>rel8</i> | REL | 07 rr | 4 | PPPS |

BVC

Branch if Overflow Cleared

BVC

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

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------|--------------|-------------|--------|----------------------|
| BVC <i>rel8</i> | REL | 28 rr | 3/1 | 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 | $Z + (N \oplus V) = 0$ | $r \leq m$ | BLE | 2F | Signed |
| $r \geq m$ | BGE | 2C | $N \oplus V = 0$ | $r < m$ | BLT | 2D | Signed |
| $r = m$ | BEQ | 27 | $Z = 1$ | $r \neq m$ | BNE | 26 | Signed |
| $r \leq m$ | BLE | 2F | $Z + (N \oplus V) = 1$ | $r > m$ | BGT | 2E | Signed |
| $r < m$ | BLT | 2D | $N \oplus V = 1$ | $r \geq m$ | BGE | 2C | Signed |
| $r > m$ | BHI | 22 | $C + Z = 0$ | $r \leq m$ | BLS | 23 | Unsigned |
| $r \geq m$ | BHS/BCC | 24 | $C = 0$ | $r < m$ | BLO/BCS | 25 | Unsigned |
| $r = m$ | BEQ | 27 | $Z = 1$ | $r \neq m$ | BNE | 26 | Unsigned |
| $r \leq m$ | BLS | 23 | $C + Z = 1$ | $r > m$ | BHI | 22 | Unsigned |
| $r < m$ | BLO/BCS | 25 | $C = 1$ | $r \geq m$ | 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 | $Z = 1$ | $r \neq 0$ | BNE | 26 | Simple |
| Always | BRA | 20 | — | Never | BRN | 21 | Unconditional |

BVS

Branch if Overflow Set

BVS

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

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|----------------------|
| BVS rel8 | REL | 29 rr | 3/1 | 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 |

CALL

Call Subroutine in Expanded Memory

CALL

Operation: $(SP) - \$0002 \Rightarrow SP$
 $RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP + 1)}$
 $(SP) - \$0001 \Rightarrow SP$
 $(PPAGE) \Rightarrow M_{(SP)}$
page $\Rightarrow PPAGE$
Subroutine Address $\Rightarrow PC$

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, 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.

Decrements the SP by one, to allow the current memory page value in the PPAGE register to be stacked.

Stacks the content of PPAGE.

Writes a new page value supplied by the instruction to PPAGE.

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

CALL

Call Subroutine in Expanded Memory (Continued)

CALL

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------------------|--------------|----------------|--------|---------------|
| CALL <i>opr16a, page</i> | EXT | 4A hh ll pg | 8 | gnfSsPPP |
| CALL <i>opr16a, page</i> | IDX | 4B xb pg | 8 | gnfSsPPP |
| CALL <i>opr16a, page</i> | IDX1 | 4B xb ff pg | 8 | gnfSsPPP |
| CALL <i>opr16a, page</i> | IDX2 | 4B xb ee ff pg | 9 | fgnfSsPPP |
| CALL [D,xySP] | [D,IDX] | 4B xb | 10 | fIignSsPPP |
| CALL [opr16a, xySP] | [IDX2] | 4B xb ee ff | 10 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $\overline{A7} \bullet B7 + B7 \bullet R7 + R7 + \overline{A7}$

Set if there was a borrow from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| CBA | INH | 18 17 | 2 | 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 0 |

C: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------------------------|--------------|-------------|--------|---------------|
| CLC translates to... ANDCC #\$FE | IMM | 10 FE | 1 | 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | 0 | - | - | - | - |

I: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------------------------|--------------|-------------|--------|---------------|
| CLI translates to... ANDCC #\$EF | IMM | 10 EF | 1 | P |

CLR

Clear Memory

CLR

Operation: $0 \Rightarrow M$

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

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | 1 | 0 | 0 |

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| CLR <i>opr16a</i> | EXT | 79 hh ll | 3 | wOP |
| CLR <i>opr16,xysp</i> | IDX | 69 xb | 2 | Pw |
| CLR <i>opr9,xysp</i> | IDX1 | 69 xb ff | 3 | PwO |
| CLR <i>opr16,xysp</i> | IDX2 | 69 xb ee ff | 3 | PwP |
| CLR [D, <i>xysp</i>] | [D,IDX] | 69 xb | 5 | PIfPw |
| CLR [<i>opr16,xysp</i>] | [IDX2] | 69 xb ee ff | 5 | PIPPw |

CLRA

Clear A

CLRA

Operation: $0 \Rightarrow A$

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

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | 1 | 0 | 0 |

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| CLRA | INH | 87 | 1 | 0 |

CLRB

Clear B

CLRB

Operation: $0 \Rightarrow B$

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

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | 1 | 0 | 0 |

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| CLRB | INH | C7 | 1 | 0 |

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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | 0 | - |

V: 0; cleared

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------------------------|--------------|-------------|--------|---------------|
| CLV translates to... ANDCC #\$FD | IMM | 10 FD | 1 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + X7 \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 + \overline{X7}$

Set if there was a borrow from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| CMPA #opr8 <i>i</i> | IMM | 81 ii | 1 | P |
| CMPA opr8 <i>a</i> | DIR | 91 dd | 3 | rP |
| CMPA opr16 <i>a</i> | EXT | B1 hh ll | 3 | rOP |
| CMPA oprx0_xy <i>sp</i> | IDX | A1 xb | 3 | rP |
| CMPA oprx9,xy <i>sp</i> | IDX1 | A1 xb ff | 3 | rPO |
| CMPA oprx16,xy <i>sp</i> | IDX2 | A1 xb ee ff | 4 | fPP |
| CMPA [D,xy <i>sp</i>] | [D,IDX] | A1 xb | 6 | fIPrfP |
| CMPA [opr16,xy <i>sp</i>] | [IDX2] | A1 xb ee ff | 6 | fIPrfP |

CMPB

Compare B

CMPB

Operation: (B) – (M)

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

**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + \overline{X7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 + \overline{X7}$

Set if there was a borrow from the MSB of the result; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| CMPB #opr8 <i>i</i> | IMM | C1 ii | 1 | P |
| CMPB opr8 <i>a</i> | DIR | D1 dd | 3 | rP |
| CMPB opr16 <i>a</i> | EXT | F1 hh ll | 3 | rOP |
| CMPB oprx0_xy <i>sp</i> | IDX | E1 xb | 3 | rP |
| CMPB oprx9,xy <i>sp</i> | IDX1 | E1 xb ff | 3 | rPO |
| CMPB oprx16,xy <i>sp</i> | IDX2 | E1 xb ee ff | 4 | fPP |
| CMPB [D,xy <i>sp</i>] | [D,IDX] | E1 xb | 6 | fIPrfP |
| CMPB [opr16,xy <i>sp</i>] | [IDX2] | E1 xb ee ff | 6 | fIPrfP |

COM**Complement Memory****COM****Operation:** $(\bar{M}) = \$FF - (M) \Rightarrow M$ **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.**Condition Codes
and Boolean
Formulas:**

| 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)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| COM <i>opr16a</i> | EXT | 71 hh 11 | 4 | rOPw |
| COM <i>opr0_xysp</i> | IDX | 61 xb | 3 | rPw |
| COM <i>opr9,xysp</i> | IDX1 | 61 xb ff | 4 | rPOw |
| COM <i>opr16,xysp</i> | IDX2 | 61 xb ee ff | 5 | fPPw |
| COM [D, <i>xysp</i>] | [D,IDX] | 61 xb | 6 | fIfPw |
| COM [<i>opr16,xysp</i>] | [IDX2] | 61 xb ee ff | 6 | fIPrPw |

COMA

Complement A

COMA**Operation:** $(\bar{A}) = \$FF - (A) \Rightarrow A$ **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.**Condition Codes
and Boolean
Formulas:**

| 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)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| COMA | INH | 41 | 1 | O |

COMB

Complement B

COMB

Operation: $(\bar{B}) = \$FF - (B) \Rightarrow B$ **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.**Condition Codes
and Boolean
Formulas:**

| 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)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| COMB | INH | 51 | 1 | O |

CPD

Compare Double Accumulator

CPD

Operation: $(A : B) - (M : M + 1)$

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

**Condition Codes
and Boolean
Formulas:**

| 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: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 + \overline{D15}$

Set if the absolute value of the content of memory is larger than the absolute value of the accumulator; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| CPD #opr16i | IMM | 8C jj kk | 2 | OP |
| CPD opr8a | DIR | 9C dd | 3 | RfP |
| CPD opr16a | EXT | BC hh ll | 3 | ROP |
| CPD oprx0_xysp | IDX | AC xb | 3 | RfP |
| CPD oprx9_xysp | IDX1 | AC xb ff | 3 | RPO |
| CPD oprx16_xysp | IDX2 | AC xb ee ff | 4 | fRPP |
| CPD [D,xysp] | [D,IDX] | AC xb | 6 | fIfRfP |
| CPD [opr16,xysp] | [IDX2] | AC xb ee ff | 6 | fIPRfP |

CPS

Compare Stack Pointer

CPS

Operation: $(SP) - (M : M + 1)$

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

**Condition Codes
and Boolean
Formulas:**

| 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: $\overline{S15} \bullet M15 + M15 \bullet R15 + R15 + \overline{S15}$

Set if the absolute value of the content of memory is larger than the absolute value of the SP; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| CPS #opr16 <i>i</i> | IMM | 8F jj kk | 2 | OP |
| CPS opr8 <i>a</i> | DIR | 9F dd | 3 | RfP |
| CPS opr16 <i>a</i> | EXT | BF hh ll | 3 | ROP |
| CPS oprx0_xysp | IDX | AF xb | 3 | RfP |
| CPS oprx9_xysp | IDX1 | AF xb ff | 3 | RPO |
| CPS oprx16_xysp | IDX2 | AF xb ee ff | 4 | fRPP |
| CPS [D,xysp] | [D,IDX] | AF xb | 6 | fIfRfP |
| CPS [opr16,xysp] | [IDX2] | AF xb ee ff | 6 | fIPRfP |

CPX

Compare Index Register X

CPX

Operation: $(X) - (M : M + 1)$

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

**Condition Codes
and Boolean
Formulas:**

| 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: $X_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + X_{15} \bullet M_{15} \bullet R_{15}$
Set if two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} + \overline{X_{15}}$
Set if the absolute value of the content of memory is larger than the absolute value of the index register; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| CPX #opr16 <i>i</i> | IMM | 8E jj kk | 2 | OP |
| CPX opr8 <i>a</i> | DIR | 9E dd | 3 | RfP |
| CPX opr16 <i>a</i> | EXT | BE hh ll | 3 | ROP |
| CPX oprx0_xy <i>sp</i> | IDX | AE xb | 3 | RfP |
| CPX oprx9_xy <i>sp</i> | IDX1 | AE xb ff | 3 | RPO |
| CPX oprx16_xy <i>sp</i> | IDX2 | AE xb ee ff | 4 | fRPP |
| CPX [D,xy <i>sp</i>] | [D,IDX] | AE xb | 6 | fIfRfP |
| CPX [opr16,xy <i>sp</i>] | [IDX2] | AE xb ee ff | 6 | fIPRfP |

CPY

Compare Index Register Y

CPY

Operation: $(Y) - (M : M + 1)$

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

**Condition Codes
and Boolean
Formulas:**

| 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: $Y_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{Y_{15}} \bullet M_{15} \bullet R_{15}$

Set if two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{Y_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} + \overline{Y_{15}}$

Set if the absolute value of the content of memory is larger than the absolute value of the index register; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| CPY #opr16 <i>i</i> | IMM | 8D jj kk | 2 | OP |
| CPY opr8 <i>a</i> | DIR | 9D dd | 3 | RfP |
| CPY opr16 <i>a</i> | EXT | BD hh ll | 3 | ROP |
| CPY oprx0_xysp | IDX | AD xb | 3 | RfP |
| CPY oprx9_xysp | IDX1 | AD xb ff | 3 | RPO |
| CPY oprx16_xysp | IDX2 | AD xb ee ff | 4 | fRPP |
| CPY [D,xysp] | [D,IDX] | AD xb | 6 | fIfRfP |
| CPY [opr16,xysp] | [IDX2] | AD xb ee ff | 6 | fIPRfP |

DAA**Decimal Adjust A****DAA**

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

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

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

DAA

Decimal Adjust A (Continued)

DAA

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| DAA | INH | 18 07 | 3 | OFO |

DBEQ

Decrement and Branch if Equal to Zero

DBEQ

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

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

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

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|-------------------|--------------|----------------------------|--------|---------------|
| DBEQ abdxys, rel9 | REL | 04 1b rr | 3/3 | PPP |

1. Encoding for 1b 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: $(\text{Counter}) - 1 \Rightarrow \text{Counter}$
If $(\text{Counter}) \neq 0$, then $(\text{PC}) + \$0003 + \text{Rel} \Rightarrow \text{PC}$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|-------------------|--------------|----------------------------|--------|---------------|
| DBNE abdxys, rel9 | REL | 04 1b rr | 3/3 | PPP |

1. Encoding for 1b 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: $(M) - \$01 \Rightarrow M$

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.

**Condition Codes
and Boolean**

Formulas:

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| DEC <i>opr16a</i> | EXT | 73 hh 11 | 4 | rOPw |
| DEC <i>opr0_xysp</i> | IDX | 63 xb | 3 | rPw |
| DEC <i>opr9,xysp</i> | IDX1 | 63 xb ff | 4 | rPOw |
| DEC <i>opr16,xysp</i> | IDX2 | 63 xb ee ff | 5 | fPPw |
| DEC [D, <i>xysp</i>] | [D,IDX] | 63 xb | 6 | fIfnPw |
| DEC [<i>opr16,xysp</i>] | [IDX2] | 63 xb ee ff | 6 | fIPrPw |

DECA

Decrement A

DECA

Operation: $(A) - \$01 \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| DECA | INH | 43 | 1 | O |

DECB

Decrement B

DECB

Operation: $(B) - \$01 \Rightarrow B$

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.

Condition Codes

and Boolean

Formulas:

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| DECB | INH | 53 | 1 | 0 |

DES

Decrement Stack Pointer

DES

Operation: $(SP) - \$0001 \Rightarrow SP$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------------------|--------------|-------------|--------|-------------------|
| DES translates to... LEAS –1,SP | IDX | 1B 9F | 2 | PP ⁽¹⁾ |

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

DEX

Decrement Index Register X

DEX

Operation: $(X) - \$0001 \Rightarrow X$

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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|----------|---|---|
| - | - | - | - | - | Δ | - | - |

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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| DEX | INH | 09 | 1 | 0 |

DEY

Decrement Index Register Y

DEY

Operation: $(Y) - \$0001 \Rightarrow Y$

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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | Δ | - | - |

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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| DEY | INH | 03 | 1 | 0 |

EDIV

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

EDIV

Operation: $(Y : D) \div (X) \Rightarrow Y; \text{Remainder} \Rightarrow D$

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| EDIV | INH | 11 | 11 | fffffffffffo |

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

Operation: $(Y : D) \div (X) \Rightarrow Y; \text{Remainder} \Rightarrow D$

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|------------------|
| EDIVS | INH | 18 14 | 12 | 0xfffffffffffff0 |

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.

**Condition Codes
and Boolean**

Formulas:

| 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: $M31 \bullet I31 \bullet \overline{R31} + \overline{M31} \bullet I31 \bullet R31$

Set if result > \$FFFFFF (+ 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form ⁽¹⁾ | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| EMACS <i>opr16a</i> | Special | 18 12 hh 11 | 13 | 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

Operation: MAX ((D), (M : M + 1)) \Rightarrow 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.

Condition Codes and Boolean Formulas:

| 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 ($R = D - M : M + 1$)

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------------|--------------|----------------|--------|---------------|
| EMAXD oprx0_xysp | IDX | 18 1A xb | 4 | ORfP |
| EMAXD oprx9,xysp | IDX1 | 18 1A xb ff | 4 | ORPO |
| EMAXD oprx16,xysp | IDX2 | 18 1A xb ee ff | 5 | OfRPP |
| EMAXD [D,xysp] | [D,IDX] | 18 1A xb | 7 | OfIFRfP |
| EMAXD [opr16,xysp] | [IDX2] | 18 1A xb ee ff | 7 | OfIPRfP |

EMAXM

Place Larger of Two
Unsigned 16-Bit Values
in Memory

EMAXM

Operation: MAX ((D), (M : M + 1)) \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $D_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{D_{15}} \bullet M_{15} \bullet R_{15}$
Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} \bullet \overline{D_{15}}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = D - M : M + 1$)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------|--------------|----------------|--------|---------------|
| EMAXM <i>opr0_xysp</i> | IDX | 18 1E xb | 4 | ORPW |
| EMAXM <i>opr9_xysp</i> | IDX1 | 18 1E xb ff | 5 | ORPWO |
| EMAXM <i>opr16_xysp</i> | IDX2 | 18 1E xb ee ff | 6 | OFRPWP |
| EMAXM [D, <i>xysp</i>] | [D,IDX] | 18 1E xb | 7 | OFlFRPW |
| EMAXM [<i>opr16,xysp</i>] | [IDX2] | 18 1E xb ee ff | 7 | OFlPRPW |

EMIND

Place Smaller of Two Unsigned 16-Bit Values in Accumulator D

EMIND

Operation: MIN ((D), (M : M + 1)) \Rightarrow 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.

Condition Codes and Boolean Formulas:

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

Condition codes reflect internal subtraction ($R = D - M : M + 1$)

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|----------------|--------|---------------|
| EMIND oprx0_xysp | IDX | 18 1B xb | 4 | ORFP |
| EMIND oprx9,xysp | IDX1 | 18 1B xb ff | 4 | ORPO |
| EMIND oprx16,xysp | IDX2 | 18 1B xb ee ff | 5 | OFRPP |
| EMIND [D,xysp] | [D,IDX] | 18 1B xb | 7 | OFIFRFP |
| EMIND [oprx16,xysp] | [IDX2] | 18 1B xb ee ff | 7 | OFIPRFP |

EMINM

Place Smaller of Two
Unsigned 16-Bit Values
in Memory

EMINM

Operation: $\text{MIN}((D), (M : M + 1)) \Rightarrow M : M + 1$

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.

**Condition Codes
and Boolean
Formulas:**

| 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 ($R = D - M : M + 1$)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|----------------|--------|---------------|
| EMINM <i>opr</i> x0_xysp | IDX | 18 1F xb | 4 | ORPW |
| EMINM <i>opr</i> x9_xysp | IDX1 | 18 1F xb ff | 5 | ORPWO |
| EMINM <i>opr</i> x16_xysp | IDX2 | 18 1F xb ee ff | 6 | OFRPWP |
| EMINM [D,xy] | [D,IDX] | 18 1F xb | 7 | OFIFRPW |
| EMINM [<i>opr</i> x16_xy] | [IDX2] | 18 1F xb ee ff | 7 | OFIPRPW |

EMUL

**Extended Multiply
16-Bit by 16-Bit (Unsigned)**

EMUL

Operation: $(D) \times (Y) \Rightarrow Y : D$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| EMUL | INH | 13 | 3 | ff0 |

EMULS

Extended Multiply
16-Bit by 16-Bit (Signed)

EMULS

Operation: $(D) \times (Y) \Rightarrow Y : D$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| EMULS | INH | 18 13 | 3 | ofo |

EORA

Exclusive OR A

EORA

Operation: $(A) \oplus (M) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| EORA #opr8 <i>i</i> | IMM | 88 ii | 1 | P |
| EORA opr8 <i>a</i> | DIR | 98 dd | 3 | rfP |
| EORA opr16 <i>a</i> | EXT | B8 hh ll | 3 | rOP |
| EORA oprx0_xysp | IDX | A8 xb | 3 | rfP |
| EORA oprx9,xysp | IDX1 | A8 xb ff | 3 | rPO |
| EORA oprx16,xysp | IDX2 | A8 xb ee ff | 4 | frPP |
| EORA [D,xysp] | [D,IDX] | A8 xb | 6 | fIfrfP |
| EORA [opr16,xysp] | [IDX2] | A8 xb ee ff | 6 | fIPrfP |

EORB

Exclusive OR B

EORB

Operation: $(B) \oplus (M) \Rightarrow B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| EORB #opr8 <i>i</i> | IMM | C8 ii | 1 | P |
| EORB opr8 <i>a</i> | DIR | D8 dd | 3 | rFP |
| EORB opr16 <i>a</i> | EXT | F8 hh ll | 3 | rOP |
| EORB oprx0_xy <i>sp</i> | IDX | E8 xb | 3 | rFP |
| EORB oprx9_xy <i>sp</i> | IDX1 | E8 xb ff | 3 | rPO |
| EORB oprx16_xy <i>sp</i> | IDX2 | E8 xb ee ff | 4 | fRPP |
| EORB [D,xy <i>sp</i>] | [D,IDX] | E8 xb | 6 | fIfRFP |
| EORB [opr16,xy <i>sp</i>] | [IDX2] | E8 xb ee ff | 6 | fIPrFP |

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 points in the table represent the 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, set up an index register so that it points to the starting point (X1) of a line segment when the instruction is executed. X1 is the table entry closest to, but less than or equal to, the desired lookup value. The next table entry after X1 is X2. XL is the distance in X between X1 and X2. Load accumulator B with a binary fraction (radix point to left of MSB) representing the ratio $(XL - X1) \div (X2 - X1)$.

The 16-bit unrounded result is calculated using the following expression:

$$D = Y1 + [(B) \times (Y2 - Y1)]$$

Where:

$$(B) = (XL - X1) \div (X2 - X1)$$

Y1 = 16-bit data entry pointed to by <effective address>

Y2 = 16-bit data entry pointed to by <effective address> + 2

The intermediate value $[(B) \times (Y2 - Y1)]$ 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.

Condition Codes and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | Δ | Δ | - | ? |

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

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

C: Undefined

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| ETBL oprx0_xyssp | IDX | 18 3F xb | 10 | ORRfffffP |

EXG

Exchange Register Contents

EXG

Operation: See table

Description: Exchanges the contents of registers specified in the instruction as shown below. Note that the order in which exchanges between 8-bit and 16-bit registers are specified affects the high byte of the 16-bit registers differently. Exchanges of D with A or B are ambiguous. Cases involving TMP2 and TMP3 are reserved for Motorola use, so some assemblers may not permit their use, but it is possible to generate these cases by using DC.B or DC.W assembler directives.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

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 XIRQ interrupt.

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|---------------------|--------------|----------------------------|--------|---------------|
| EXG abcdxys,abcdxys | INH | B7 eb | 1 | P |

1. Legal coding for eb is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit (bit 3 is a don't care). Values are in hexadecimal.

| | 8 | 9 | A | B | C | D | E | F |
|---|--|--|--|--|-------------------------|--------------------------------------|--------------------------------------|--|
| 0 | A ⇌ A | B ⇌ A | CCR ⇌ A | TMP3 _L ⇒ A \$00:A ⇒ TMP3 | B ⇒ A A ⇒ B | X _L ⇒ A \$00:A ⇒ X | Y _L ⇒ A \$00:A ⇒ Y | SP _L ⇒ A \$00:A ⇒ SP |
| 1 | A ⇌ B | B ⇌ B | CCR ⇌ B | TMP3 _L ⇒ B \$FF:B ⇒ TMP3 | B ⇒ B \$FF ⇒ A | X _L ⇒ B \$FF:B ⇒ X | Y _L ⇒ B \$FF:B ⇒ Y | SP _L ⇒ B \$FF:B ⇒ SP |
| 2 | A ⇌ CCR | B ⇌ CCR | CCR ⇌ CCR | TMP3 _L ⇒ CCR \$FF:CCR ⇒ TMP3 | B ⇒ CCR \$FF:CCR ⇒ D | X _L ⇒ CCR \$FF:CCR ⇒ X | Y _L ⇒ CCR \$FF:CCR ⇒ Y | SP _L ⇒ CCR \$FF:CCR ⇒ SP |
| 3 | \$00:A ⇒ TMP2 TMP2 _L ⇒ A | \$00:B ⇒ TMP2 TMP2 _L ⇒ B | \$00:CCR ⇒ TMP2 TMP2 _L ⇒ CCR | TMP3 ⇌ TMP2 | D ⇌ TMP2 | X ⇌ TMP2 | Y ⇌ TMP2 | SP ⇌ TMP2 |
| 4 | \$00:A ⇒ D | \$00:B ⇒ D | \$00:CCR ⇒ D B ⇒ CCR | TMP3 ⇌ D | D ⇌ D | X ⇌ D | Y ⇌ D | SP ⇌ D |
| 5 | \$00:A ⇒ X X _L ⇒ A | \$00:B ⇒ X X _L ⇒ B | \$00:CCR ⇒ X X _L ⇒ CCR | TMP3 ⇌ X | D ⇌ X | X ⇌ X | Y ⇌ X | SP ⇌ X |
| 6 | \$00:A ⇒ Y Y _L ⇒ A | \$00:B ⇒ Y Y _L ⇒ B | \$00:CCR ⇒ Y Y _L ⇒ CCR | TMP3 ⇌ Y | D ⇌ Y | X ⇌ Y | Y ⇌ Y | SP ⇌ Y |
| 7 | \$00:A ⇒ SP SP _L ⇒ A | \$00:B ⇒ SP SP _L ⇒ B | \$00:CCR ⇒ SP SP _L ⇒ CCR | TMP3 ⇌ SP | D ⇌ SP | X ⇌ SP | Y ⇌ SP | SP ⇌ SP |

FDIV

Fractional Divide

FDIV

Operation: $(D) \div (X) \Rightarrow X; \text{Remainder} \Rightarrow D$

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 32×16 -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.

Condition Codes and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | Δ | Δ | Δ |

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

V: 1 if $X \leq D$

Set if the denominator was less than or equal to the numerator; cleared otherwise

C: $\overline{X_{15}} \bullet \overline{X_{14}} \bullet \overline{X_{13}} \bullet \overline{X_{12}} \bullet \dots \bullet \overline{X_3} \bullet \overline{X_2} \bullet \overline{X_1} \bullet \overline{X_0}$

Set if denominator was \$0000; cleared otherwise

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-----------------|
| FDIV | INH | 18 11 | 12 | 0xffffffffffff0 |

IBEQ

Increment and Branch if Equal to Zero

IBEQ

Operation: $(\text{Counter}) + 1 \Rightarrow \text{Counter}$
 If $(\text{Counter}) = 0$, then $(\text{PC}) + \$0003 + \text{Rel} \Rightarrow \text{PC}$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|-------------------|--------------|----------------------------|--------|---------------|
| IBEQ abdxys, rel9 | REL | 04 1b rr | 3/3 | PPP |

1. Encoding for 1b 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: $(\text{Counter}) + 1 \Rightarrow \text{Counter}$
 If $(\text{Counter}) \neq 0$, then $(\text{PC}) + \$0003 + \text{Rel} \Rightarrow \text{PC}$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|-------------------|--------------|----------------------------|--------|---------------|
| IBNE abdxys, rel9 | REL | 04 1b rr | 3/3 | PPP |

1. Encoding for 1b 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: $(D) \div (X) \Rightarrow X; \text{Remainder} \Rightarrow D$

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, the quotient is set to \$FFFF, and the remainder is indeterminate.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|----------|---|----------|
| - | - | - | - | - | Δ | 0 | Δ |

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

V: 0; cleared

C: $\overline{X_{15}} \bullet \overline{X_{14}} \bullet \overline{X_{13}} \bullet \overline{X_{12}} \bullet \dots \bullet \overline{X_3} \bullet \overline{X_2} \bullet \overline{X_1} \bullet \overline{X_0}$
Set if denominator was \$0000; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| IDIV | INH | 18 10 | 12 | 0ffffffffff0 |

IDIVS

Integer Divide (Signed)

IDIVS

Operation: $(D \div X) \Rightarrow X; \text{Remainder} \Rightarrow D$

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, but the values of the N, Z, and V status bits are undefined.

Other than division by zero, which is not legal and causes the C status bit to be set, the only overflow case is:

$$\frac{\$8000}{\$FFFF} = \frac{-32,768}{-1} = +32,768$$

But the highest positive value that can be represented in a 16-bit two's complement number is 32,767 (\$7FFF).

Condition Codes

and Boolean Formulas:

| 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: $\overline{X_{15}} \bullet \overline{X_{14}} \bullet \overline{X_{13}} \bullet \overline{X_{12}} \bullet \dots \bullet \overline{X_3} \bullet \overline{X_2} \bullet \overline{X_1} \bullet \overline{X_0}$
Set if denominator was \$0000; cleared otherwise

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| IDIVS | INH | 18 15 | 12 | 0ffffffffff0 |

INC

Increment Memory

INC

Operation: $(M) + \$01 \Rightarrow M$

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.

Condition Codes

and Boolean Formulas:

| 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.

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| INC opr16a | EXT | 72 hh ll | 4 | rOPw |
| INC oprx0_xysp | IDX | 62 xb | 3 | rPw |
| INC oprx9,xysp | IDX1 | 62 xb ff | 4 | rPOw |
| INC oprx16,xysp | IDX2 | 62 xb ee ff | 5 | fPPw |
| INC [D,xysp] | [D,IDX] | 62 xb | 6 | fIfPrPw |
| INC [opr16,xysp] | [IDX2] | 62 xb ee ff | 6 | fIPrPw |

INCA

Increment A

INCA

Operation: $(A) + \$01 \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| INCA | INH | 42 | 1 | 0 |

INCB

Increment B

INCB

Operation: $(B) + \$01 \Rightarrow B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| INCB | INH | 52 | 1 | 0 |

INS

Increment Stack Pointer

INS

Operation: $(SP) + \$0001 \Rightarrow SP$

Description: Add one to the SP. This instruction is assembled to LEAS 1,SP. The LEAS instruction does not affect condition codes as an INX orINY instruction would.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------------|--------------|-------------|--------|-------------------|
| INS translates to... LEAS 1,SP | IDX | 1B 81 | 2 | PP ⁽¹⁾ |

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

INX**Increment Index Register X****INX**

Operation: $(X) + \$0001 \Rightarrow X$

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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | Δ | - | - |

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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| INX | INH | 08 | 1 | 0 |

INY

Increment Index Register Y

INY

Operation: $(Y) + \$0001 \Rightarrow Y$

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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|----------|---|---|
| - | - | - | - | - | Δ | - | - |

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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| INY | INH | 02 | 1 | 0 |

JMP

Jump

JMP

Operation: Effective Address \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| JMP opr16a | EXT | 06 hh 11 | 3 | PPP |
| JMP oprx0_xysp | IDX | 05 xb | 3 | PPP |
| JMP oprx9,xysp | IDX1 | 05 xb ff | 3 | PPP |
| JMP oprx16,xysp | IDX2 | 05 xb ee ff | 4 | fPPP |
| JMP [D,xysp] | [D,IDX] | 05 xb | 6 | fIfPPP |
| JMP [oprx16,xysp] | [IDX2] | 05 xb ee ff | 6 | fIfPPP |

JSR

Jump to Subroutine

JSR

Operation: $(SP) - \$0002 \Rightarrow SP$

$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP + 1)}$
Subroutine Address $\Rightarrow PC$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| JSR <i>opr8a</i> | DIR | 17 dd | 4 | PPPS |
| JSR <i>opr16a</i> | EXT | 16 hh 11 | 4 | PPPS |
| JSR <i>opr0_xysp</i> | IDX | 15 xb | 4 | PPPS |
| JSR <i>opr9,xysp</i> | IDX1 | 15 xb ff | 4 | PPPS |
| JSR <i>opr16,xysp</i> | IDX2 | 15 xb ee ff | 5 | fPPPS |
| JSR [D, <i>xysp</i>] | [D,IDX] | 15 xb | 7 | fIfPPPS |
| JSR [<i>opr16,xysp</i>] | [IDX2] | 15 xb ee ff | 7 | fIfPPPS |

LBCC

**Long Branch if Carry Cleared
(Same as LBHS)**

LBCC

Operation: If C = 0, then (PC) + \$0004 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBCC rel/16 | REL | 18 24 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBCS

**Long Branch if Carry Set
(Same as LBLO)**

LBCS

Operation: If C = 1, then (PC) + \$0004 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBCS rel16 | REL | 18 25 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBEQ

Long Branch if Equal

LBEQ

Operation: If Z = 1, (PC) + \$0004 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBEQ rel/16 | REL | 18 27 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | Z + (N \oplus V) = 0 | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | N \oplus V = 0 | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | Z + (N \oplus V) = 1 | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | N \oplus V = 1 | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | C + Z = 0 | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | C = 0 | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | C + Z = 1 | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | C = 1 | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | C = 1 | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | N = 1 | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | V = 1 | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | Z = 1 | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBGE

Long Branch if Greater Than or Equal to Zero

LBGE

Operation: If $N \oplus V = 0$, $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers,
if (Accumulator) \geq Memory), then branch

Description: If LBGE 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 two's complement number in the accumulator was greater than or equal to the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBGE rel/16 | REL | 18 2C qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | $r \leq m$ | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | $r < m$ | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | $r \neq m$ | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | $r > m$ | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | $r \geq m$ | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | $r \leq m$ | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | $r < m$ | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | $r \neq m$ | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | $r > m$ | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | $r \geq m$ | 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 | $Z = 1$ | $r \neq 0$ | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBGT

Long Branch if Greater Than Zero

LBGT

Operation: If $Z + (N \oplus V) = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers,
If (Accumulator) > (Memory), then branch

Description: If LBGT 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 two's complement number in the accumulator was greater than the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBGT rel/16 | REL | 18 2E qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBHI

Long Branch if Higher

LBHI

Operation: If $C + Z = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if (Accumulator) > (Memory), then branch

Description: If LBHI 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 was greater than the unsigned binary number in memory. This instruction is 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.

**Condition Codes
and Boolean**

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBHI rel/16 | REL | 18 22 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|------------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | $r \leq m$ | LBLE | 18 2F | Signed |
| $r \geq m$ | LBGE | 18 2C | $N \oplus V = 0$ | $r < m$ | LBLT | 18 2D | Signed |
| $r = m$ | LBEQ | 18 27 | $Z = 1$ | $r \neq m$ | LBNE | 18 26 | Signed |
| $r \leq m$ | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | $r > m$ | LBGT | 18 2E | Signed |
| $r < m$ | LBLT | 18 2D | $N \oplus V = 1$ | $r \geq m$ | LBGE | 18 2C | Signed |
| $r > m$ | LBHI | 18 22 | $C + Z = 0$ | $r \leq m$ | LBLS | 18 23 | Unsigned |
| $r \geq m$ | LBHS/LBCC | 18 24 | $C = 0$ | $r < m$ | LBLO/LBCS | 18 25 | Unsigned |
| $r = m$ | LBEQ | 18 27 | $Z = 1$ | $r \neq m$ | LBNE | 18 26 | Unsigned |
| $r \leq m$ | LBLS | 18 23 | $C + Z = 1$ | $r > m$ | LBHI | 18 22 | Unsigned |
| $r < m$ | LBLO/LBCS | 18 25 | $C = 1$ | $r \geq m$ | 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 | $Z = 1$ | $r \neq 0$ | 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 \Rightarrow PC

For unsigned binary numbers, if (Accumulator) \geq (Memory), then branch

Description: If LBHS 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 was greater than or equal to the unsigned binary number in memory. This instruction is 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBHS rel/16 | REL | 18 24 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBLE

Long Branch if Less Than or Equal to Zero

LBLE

Operation: If $Z + (N \oplus V) = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers,
if (Accumulator) \leq (Memory), then branch

Description: If LBLE 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 two's complement number in the accumulator was less than or equal to the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBLE rel/16 | REL | 18 2F qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCS | 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/LBCS | 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 \Rightarrow PC

For unsigned binary numbers, if (Accumulator) < (Memory), then branch

Description: If LBLO 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 was less than the unsigned binary number in memory. This instruction is 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBLO rel/16 | REL | 18 25 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | Z + (N \oplus V) = 0 | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | N \oplus V = 0 | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | Z + (N \oplus V) = 1 | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | N \oplus V = 1 | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | C + Z = 0 | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | C = 0 | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | C + Z = 1 | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | C = 1 | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | C = 1 | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | N = 1 | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | V = 1 | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | Z = 1 | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBLS

Long Branch if Lower or Same

LBLS

Operation: If $C + Z = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$
For unsigned binary numbers, if (Accumulator) \leq (Memory), then branch

Description: If LBLS 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 was less than or equal to the unsigned binary number in memory. This instruction is 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBLS rel/16 | REL | 18 23 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBLT

Long Branch if Less Than Zero

LBLT

Operation: If $N \oplus V = 1$, $(PC) + \$0004 + Rel \Rightarrow PC$

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

Description: If LBLT 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 two's complement number in the accumulator was less than the two's complement number in memory.

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBLT rel/16 | REL | 18 2D qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBMI

Long Branch if Minus

LBMI

Operation: If N = 1, then (PC) + \$0004 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBMI rel/16 | REL | 18 2B qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | Z + (N \oplus V) = 0 | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | N \oplus V = 0 | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | Z + (N \oplus V) = 1 | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | N \oplus V = 1 | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | C + Z = 0 | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | C = 0 | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | C + Z = 1 | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | C = 1 | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCC | 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 $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

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

See [3.9 Relative Addressing Mode](#) for details of branch execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBNE rel16 | REL | 18 26 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCC | 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 \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBPL rel/16 | REL | 18 2A qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBRA

Long Branch Always

LBRA

Operation: $(PC) + \$0004 + Rel \Rightarrow PC$

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.

**Condition Codes
and Boolean**

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LBRA rel/16 | REL | 18 20 qq rr | 4 | OPPP |

LBRN

Long Branch Never

LBRN

Operation: $(PC) + \$0004 \Rightarrow PC$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| LBRN <i>rel16</i> | REL | 18 21 qq rr | 3 | OPO |

LBVC

Long Branch if Overflow Cleared

LBVC

Operation: If $V = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBVC rel/16 | REL | 18 28 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | $Z + (N \oplus V) = 0$ | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | $N \oplus V = 0$ | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | $Z + (N \oplus V) = 1$ | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | $N \oplus V = 1$ | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | $C + Z = 0$ | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | $C = 0$ | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | $Z = 1$ | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | $C + Z = 1$ | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | $C = 1$ | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | $C = 1$ | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | $N = 1$ | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | $V = 1$ | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | $Z = 1$ | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LBVS

Long Branch if Overflow Set

LBVS

Operation: If V = 1, then (PC) + \$0004 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|-------------------------|
| LBVS rel/16 | REL | 18 29 qq rr | 4/3 | OPPP/OPO ⁽¹⁾ |

1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

| Branch | | | | Complementary Branch | | | |
|----------|-----------|--------|------------------------|----------------------|-----------|--------|---------------|
| Test | Mnemonic | Opcode | Boolean | Test | Mnemonic | Opcode | Comment |
| r>m | LBGT | 18 2E | Z + (N \oplus V) = 0 | r≤m | LBLE | 18 2F | Signed |
| r≥m | LBGE | 18 2C | N \oplus V = 0 | r<m | LBLT | 18 2D | Signed |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Signed |
| r≤m | LBLE | 18 2F | Z + (N \oplus V) = 1 | r>m | LBGT | 18 2E | Signed |
| r<m | LBLT | 18 2D | N \oplus V = 1 | r≥m | LBGE | 18 2C | Signed |
| r>m | LBHI | 18 22 | C + Z = 0 | r≤m | LBLS | 18 23 | Unsigned |
| r≥m | LBHS/LBCC | 18 24 | C = 0 | r<m | LBLO/LBCS | 18 25 | Unsigned |
| r=m | LBEQ | 18 27 | Z = 1 | r≠m | LBNE | 18 26 | Unsigned |
| r≤m | LBLS | 18 23 | C + Z = 1 | r>m | LBHI | 18 22 | Unsigned |
| r<m | LBLO/LBCS | 18 25 | C = 1 | r≥m | LBHS/LBCC | 18 24 | Unsigned |
| Carry | LBCS | 18 25 | C = 1 | No Carry | LBCC | 18 24 | Simple |
| Negative | LBMI | 18 2B | N = 1 | Plus | LBPL | 18 2A | Simple |
| Overflow | LBVS | 18 29 | V = 1 | No Overflow | LBVC | 18 28 | Simple |
| r=0 | LBEQ | 18 27 | Z = 1 | r≠0 | LBNE | 18 26 | Simple |
| Always | LBRA | 18 20 | — | Never | LBRN | 18 21 | Unconditional |

LDAA

Load Accumulator A

LDAA

Operation: $(M) \Rightarrow A$

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| LDAA #opr8 <i>i</i> | IMM | 86 ii | 1 | P |
| LDAA opr8 <i>a</i> | DIR | 96 dd | 3 | rfP |
| LDAA opr16 <i>a</i> | EXT | B6 hh ll | 3 | rOP |
| LDAA oprx0_xy <i>sp</i> | IDX | A6 xb | 3 | rfP |
| LDAA oprx9_xy <i>sp</i> | IDX1 | A6 xb ff | 3 | rPO |
| LDAA oprx16_xy <i>sp</i> | IDX2 | A6 xb ee ff | 4 | frPP |
| LDAA [D,xy <i>sp</i>] | [D,IDX] | A6 xb | 6 | fIfrfP |
| LDAA [opr16,xy <i>sp</i>] | [IDX2] | A6 xb ee ff | 6 | fIPrfP |

LDAB

Load Accumulator B

LDAB

Operation: $(M) \Rightarrow B$

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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| LDAB #opr8 <i>i</i> | IMM | C6 ii | 1 | P |
| LDAB opr8 <i>a</i> | DIR | D6 dd | 3 | rfP |
| LDAB opr16 <i>a</i> | EXT | F6 hh ll | 3 | rOP |
| LDAB oprx0_xy <i>sp</i> | IDX | E6 xb | 3 | rfP |
| LDAB oprx9_xy <i>sp</i> | IDX1 | E6 xb ff | 3 | rPO |
| LDAB oprx16_xy <i>sp</i> | IDX2 | E6 xb ee ff | 4 | frPP |
| LDAB [D,xy <i>sp</i>] | [D,IDX] | E6 xb | 6 | fIfrfP |
| LDAB [opr16,xy <i>sp</i>] | [IDX2] | E6 xb ee ff | 6 | fIPrfP |

LDD

Load Double Accumulator

LDD

Operation: $(M : M + 1) \Rightarrow A : B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| LDD #opr16 <i>i</i> | IMM | CC jj kk | 2 | OP |
| LDD opr8a | DIR | DC dd | 3 | RfP |
| LDD opr16a | EXT | FC hh ll | 3 | ROP |
| LDD oprx0_xysp | IDX | EC xb | 3 | RfP |
| LDD oprx9,xysp | IDX1 | EC xb ff | 3 | RPO |
| LDD oprx16,xysp | IDX2 | EC xb ee ff | 4 | fRPP |
| LDD [D,xysp] | [D,IDX] | EC xb | 6 | fIfRfP |
| LDD [opr16,xysp] | [IDX2] | EC xb ee ff | 6 | fIPRfP |

LDS

Load Stack Pointer

LDS

Operation: $(M : M+1) \Rightarrow SP$

Description: Loads the most significant byte of the SP with the content of memory location M, and loads the least significant byte of the SP with the content of the next byte of memory at M + 1.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| LDS #opr16 <i>i</i> | IMM | CF jj kk | 2 | OP |
| LDS opr8 <i>a</i> | DIR | DF dd | 3 | RfP |
| LDS opr16 <i>a</i> | EXT | FF hh ll | 3 | ROP |
| LDS oprx0_xysp | IDX | EF xb | 3 | RfP |
| LDS oprx9,xysp | IDX1 | EF xb ff | 3 | RPO |
| LDS oprx16,xysp | IDX2 | EF xb ee ff | 4 | fRPP |
| LDS [D,xysp] | [D,IDX] | EF xb | 6 | fIfRfP |
| LDS [opr16,xysp] | [IDX2] | EF xb ee ff | 6 | fIPRfP |

LDX

Load Index Register X

LDX

Operation: $(M : M + 1) \Rightarrow X$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| LDX #opr16 <i>i</i> | IMM | CE jj kk | 2 | OP |
| LDX opr8 <i>a</i> | DIR | DE dd | 3 | RfP |
| LDX opr16 <i>a</i> | EXT | FE hh ll | 3 | ROP |
| LDX oprx0_xysp | IDX | EE xb | 3 | RfP |
| LDX oprx9,xysp | IDX1 | EE xb ff | 3 | RPO |
| LDX oprx16,xysp | IDX2 | EE xb ee ff | 4 | fRPP |
| LDX [D,xysp] | [D,IDX] | EE xb | 6 | fIfRfP |
| LDX [opr16,xysp] | [IDX2] | EE xb ee ff | 6 | fIPRfP |

LDY

Load Index Register Y

LDY

Operation: $(M : M + 1) \Rightarrow Y$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| LDY #opr16 <i>i</i> | IMM | CD jj kk | 2 | OP |
| LDY opr8 <i>a</i> | DIR | DD dd | 3 | RfP |
| LDY opr16 <i>a</i> | EXT | FD hh ll | 3 | ROP |
| LDY oprx0,_xysp | IDX | ED xb | 3 | RfP |
| LDY oprx9,_xysp | IDX1 | ED xb ff | 3 | RPO |
| LDY oprx16,_xysp | IDX2 | ED xb ee ff | 4 | fRPP |
| LDY [D,_xysp] | [D,IDX] | ED xb | 6 | fIfRfP |
| LDY [opr16,_xysp] | [IDX2] | ED xb ee ff | 6 | fIPRfP |

LEAS**Load Stack Pointer with Effective Address****LEAS****Operation:** Effective Address \Rightarrow 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.

Condition Codes**and Boolean****Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------|--------------|-------------|--------|-------------------|
| LEAS oprx0_xy | IDX | 1B xb | 2 | PP ⁽¹⁾ |
| LEAS oprx9_xy | IDX1 | 1B xb ff | 2 | PO |
| LEAS oprx16_xy | IDX2 | 1B xb ee ff | 2 | PP |

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

LEAX

Load X with Effective Address

LEAX

Operation: Effective Address \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------|--------------|-------------|--------|-------------------|
| LEAX oprx0_xy | IDX | 1A xb | 2 | PP ⁽¹⁾ |
| LEAX oprx9_xy | IDX1 | 1A xb ff | 2 | PO |
| LEAX oprx16_xy | IDX2 | 1A xb ee ff | 2 | PP |

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

LEAY

Load Y with Effective Address

LEAY

Operation: Effective Address \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|-------------------|
| LEAY oprx0_xysp | IDX | 19 xb | 2 | PP ⁽¹⁾ |
| LEAY oprx9,xysp | IDX1 | 19 xb ff | 2 | PO |
| LEAY oprx16,xysp | IDX2 | 19 xb ee ff | 2 | PP |

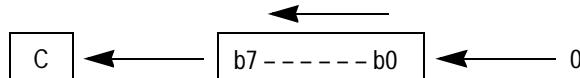
1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.

LSL

**Logical Shift Left Memory
(Same as ASL)**

LSL

Operation:



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

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

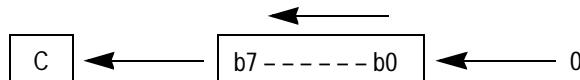
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| LSL opr16a | EXT | 78 hh 11 | 4 | rOPw |
| LSL oprx0_xysp | IDX | 68 xb | 3 | rPw |
| LSL oprx9,xysp | IDX1 | 68 xb ff | 4 | rPOw |
| LSL oprx16,xysp | IDX2 | 68 xb ee ff | 5 | frPPw |
| LSL [D,xysp] | [D,IDX] | 68 xb | 6 | fIfPrPw |
| LSL [opr16,xysp] | [IDX2] | 68 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

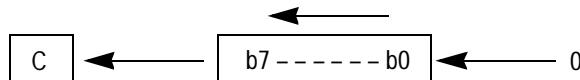
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSLA | INH | 48 | 1 | 0 |

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

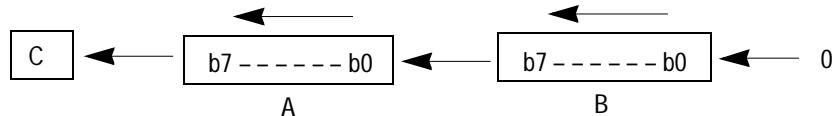
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSLB | INH | 58 | 1 | 0 |

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.

Condition Codes

and Boolean

Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

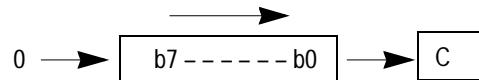
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSLD | INH | 59 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | Δ | Δ | Δ |

N: 0; cleared

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

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

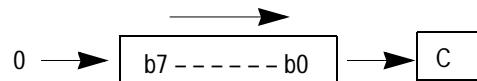
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| LSR opr16a | EXT | 74 hh 11 | 4 | rOPw |
| LSR oprx0_xysp | IDX | 64 xb | 3 | rPw |
| LSR oprx9,xysp | IDX1 | 64 xb ff | 4 | rPOw |
| LSR oprx16,xysp | IDX2 | 64 xb ee ff | 5 | fPPw |
| LSR [D,xysp] | [D,IDX] | 64 xb | 6 | fIfPrPw |
| LSR [opr16,xysp] | [IDX2] | 64 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | Δ | Δ | Δ |

N: 0; cleared

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

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

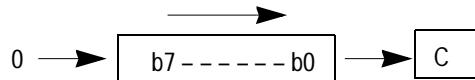
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSRA | INH | 44 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | 0 | Δ | Δ | Δ |

N: 0; cleared

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

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

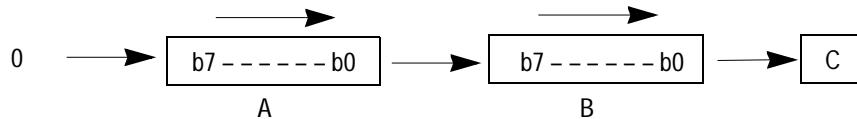
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSRB | INH | 54 | 1 | 0 |

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).

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| LSRD | INH | 49 | 1 | 0 |

MAXA

**Place Larger of Two Unsigned 8-Bit Values
in Accumulator A**

MAXA

Operation: MAX ((A), (M)) \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $X_7 \bullet M_7 \bullet R_7 + \overline{X}_7 \bullet M_7 \bullet R_7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X}_7 \bullet M_7 + M_7 \bullet R_7 + R_7 \bullet \overline{X}_7$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|----------------|--------|---------------|
| MAXA oprx0_xysp | IDX | 18 18 xb | 4 | OrfP |
| MAXA oprx9,xysp | IDX1 | 18 18 xb ff | 4 | OrPO |
| MAXA oprx16,xysp | IDX2 | 18 18 xb ee ff | 5 | OfrPP |
| MAXA [D,xysp] | [D,IDX] | 18 18 xb | 7 | OfIfrfP |
| MAXA [opr16,xysp] | [IDX2] | 18 18 xb ee ff | 7 | OfIPrfP |

MAXM

**Place Larger of Two Unsigned 8-Bit Values
in Memory**

MAXM

Operation: MAX ((A), (M)) \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

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

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|----------------|--------|---------------|
| MAXM <i>opr0_xysp</i> | IDX | 18 1C xb | 4 | OrPw |
| MAXM <i>opr9_xysp</i> | IDX1 | 18 1C xb ff | 5 | OrPwO |
| MAXM <i>opr16_xysp</i> | IDX2 | 18 1C xb ee ff | 6 | OfrPwP |
| MAXM [D, <i>xysp</i>] | [D,IDX] | 18 1C xb | 7 | OfIfrPw |
| MAXM [<i>opr16,xysp</i>] | [IDX2] | 18 1C xb ee ff | 7 | 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: Accumulator A and index registers X and Y must be set up as follows before executing MEM.

A must hold the current crisp value of a system input variable.

X must point to a 4-byte data structure that describes the trapezoidal membership function for a label of the system input.

Y must point to the fuzzy input (RAM location) where the resulting grade of membership is to be stored.

The 4-byte membership function data structure consists of Point_1, Point_2, Slope_1, and Slope_2, in that order.

Point_1 is the X-axis starting point for the leading side of the trapezoid, and Slope_1 is the slope of the leading side of the trapezoid.

Point_2 is the X-axis position of the rightmost point of the trapezoid, and Slope_2 is the slope of the trailing side of the trapezoid. The trailing side slopes up and left from Point_2.

A Slope_1 or Slope_2 value of \$00 indicates a special case where 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).

When MEM is executed, X points at Point_1 and Slope_2 is at X + 3. After execution, the content of A is unchanged. X has been incremented by four to point to the next set of membership function points and slopes. The fuzzy input (RAM location) to which Y pointed contains the grade of membership that was calculated by MEM, and Y has been incremented by one so it points to the next fuzzy input.

MEM

Determine Grade of Membership
(Fuzzy Logic)
(Continued)

MEM

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | ? | - | ? | ? | ? | ? |

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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| MEM | Special | 01 | 5 | RRfOw |

MINA

Place Smaller of Two Unsigned 8-Bit Values in Accumulator A

MINA

Operation: MIN ((A), (M)) \Rightarrow 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.

Condition Codes and Boolean Formulas:

| 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: $X_7 \bullet \overline{M}_7 \bullet \overline{R}_7 + \overline{X}_7 \bullet M_7 \bullet R_7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X}_7 \bullet M_7 + M_7 \bullet R_7 + R_7 \bullet \overline{X}_7$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|----------------|--------|---------------|
| MINA oprx0_xy | IDX | 18 19 xb | 4 | OrfP |
| MINA oprx9_xy | IDX1 | 18 19 xb ff | 4 | OrPO |
| MINA oprx16_xy | IDX2 | 18 19 xb ee ff | 5 | OfrPP |
| MINA [D,xy] | [D,IDX] | 18 19 xb | 7 | OfIfrfP |
| MINA [oprx16,xy] | [IDX2] | 18 19 xb ee ff | 7 | OfIPrfP |

MINM

Place Smaller of Two Unsigned 8-Bit Values in Memory

MINM

Operation: MIN ((A), (M)) \Rightarrow 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.

Condition Codes and Boolean Formulas:

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + \overline{X7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------------|--------------|----------------|--------|---------------|
| MINM oprx0_xysp | IDX | 18 1D xb | 4 | OrPw |
| MINM oprx9,xysp | IDX1 | 18 1D xb ff | 5 | OrPwO |
| MINM oprx16,xysp | IDX2 | 18 1D xb ee ff | 6 | OfrPwP |
| MINM [D,xysp] | [D,IDX] | 18 1D xb | 7 | OfIfrPw |
| MINM [oprx16,xysp] | [IDX2] | 18 1D xb ee ff | 7 | OfIPrPw |

MOVB

**Move a Byte of Data
from One Memory Location to Another**

MOVB

Operation: $(M_1) \Rightarrow M_2$

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

Move instructions use separate addressing modes to access the source and destination of a move. The following combinations of addressing modes are supported: IMM-EXT, IMM-IDX, EXT-EXT, EXT-IDX, IDX-EXT, and IDX-IDX. IDX operands allow indexed addressing mode specifications that fit in a single postbyte including 5-bit constant, accumulator offsets, and auto increment/decrement modes. Nine-bit and 16-bit constant offsets would require additional extension bytes and are not allowed. Indexed indirect modes (for example [D,r]) are also not allowed.

There are special considerations when using PC-relative addressing with move instructions. These are discussed in [3.11 Instructions Using Multiple Modes](#).

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form ⁽¹⁾ | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------|--------------|-------------------|--------|---------------|
| MOVB #opr8, opr16a | IMM-EXT | 18 0B ii hh ll | 4 | OPwP |
| MOVB #opr8i, oprx0_xysp | IMM-IDX | 18 08 xb ii | 4 | OPwO |
| MOVB opr16a, opr16a | EXT-EXT | 18 0C hh ll hh ll | 6 | OrPwPO |
| MOVB opr16a, oprx0_xysp | EXT-IDX | 18 09 xb hh ll | 5 | OPrPwP |
| MOVB oprx0_xysp, opr16a | IDX-EXT | 18 0D xb hh ll | 5 | OrPwP |
| MOVB oprx0_xysp, oprx0_xysp | IDX-IDX | 18 0A xb xb | 5 | OrPwO |

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

MOVW

**Move a Word of Data
from One Memory Location to Another**

MOVW

Operation: $(M : M + 1_1) \Rightarrow M : M + 1_2$

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

Move instructions use separate addressing modes to access the source and destination of a move. The following combinations of addressing modes are supported: IMM-EXT, IMM-IDX, EXT-EXT, EXT-IDX, IDX-EXT, and IDX-IDX. IDX operands allow indexed addressing mode specifications that fit in a single postbyte including 5-bit constant, accumulator offsets, and auto increment/decrement modes. Nine-bit and 16-bit constant offsets would require additional extension bytes and are not allowed. Indexed indirect modes (for example [D,r]) are also not allowed.

There are special considerations when using PC-relative addressing with move instructions. These are discussed in [3.11 Instructions Using Multiple Modes](#).

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form ⁽¹⁾ | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------|--------------|-------------------|--------|---------------|
| MOVW #opr16i, opr16a | IMM-EXT | 18 03 jj kk hh ll | 5 | OPWPO |
| MOVW #opr16i, oprx0_xysp | IMM-IDX | 18 00 xb jj kk | 4 | OPPW |
| MOVW opr16a, opr16a | EXT-EXT | 18 04 hh ll hh ll | 6 | ORPWPO |
| MOVW opr16a, oprx0_xysp | EXT-IDX | 18 01 xb hh ll | 5 | OPRPW |
| MOVW oprx0_xysp, opr16a | IDX-EXT | 18 05 xb hh ll | 5 | ORPWP |
| MOVW oprx0_xysp, oprx0_xysp | IDX-IDX | 18 02 xb xb | 5 | ORPWO |

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

MUL

**Multiply
8-Bit by 8-Bit (Unsigned)**

MUL

Operation: $(A) \times (B) \Rightarrow A : B$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| MUL | INH | 12 | 3 | ff0 |

NEG

Negate Memory

NEG

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

Description: Replaces the content of memory location M with its two's complement (the value \$80 is left unchanged).

**Condition Codes
and Boolean
Formulas:**

| 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 $(M) = \$00$.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| NEG opr16a | EXT | 70 hh 11 | 4 | rOPw |
| NEG oprx0_xysp | IDX | 60 xb | 3 | rPw |
| NEG oprx9,xysp | IDX1 | 60 xb ff | 4 | rPOw |
| NEG oprx16,xysp | IDX2 | 60 xb ee ff | 5 | frPPw |
| NEG [D,xysp] | [D,IDX] | 60 xb | 6 | fIfrrPw |
| NEG [opr16,xysp] | [IDX2] | 60 xb ee ff | 6 | fIPrPw |

NEGA

Negate A

NEGA

Operation: $0 - (A) = (\bar{A}) + 1 \Rightarrow A$

Description: Replaces the content of accumulator A with its two's complement (the value \$80 is left unchanged).

**Condition Codes
and Boolean
Formulas:**

| 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 $(A) = \$00$

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| NEGA | INH | 40 | 1 | O |

NEGB

Negate B

NEGB

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

Description: Replaces the content of accumulator B with its two's complement (the value \$80 is left unchanged).

**Condition Codes
and Boolean
Formulas:**

| 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 $(B) = \$00$

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| NEGB | INH | 50 | 1 | 0 |

NOP

Null Operation

NOP

Operation: No operation

Description: This single-byte instruction increments the PC and does nothing else. No other CPU registers are affected. NOP is typically used to produce a time delay, although some software disciplines discourage CPU frequency-based time delays. During debug, NOP instructions are sometimes used to temporarily replace other machine code instructions, thus disabling the replaced instruction(s).

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| NOP | INH | A7 | 1 | 0 |

ORAA

Inclusive OR A

ORAA

Operation: $(A) + (M) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| ORAA #opr8 <i>i</i> | IMM | 8A ii | 1 | P |
| ORAA opr8 <i>a</i> | DIR | 9A dd | 3 | rfP |
| ORAA opr16 <i>a</i> | EXT | BA hh ll | 3 | rOP |
| ORAA oprx0_xysp | IDX | AA xb | 3 | rfP |
| ORAA oprx9,xysp | IDX1 | AA xb ff | 3 | rPO |
| ORAA oprx16,xysp | IDX2 | AA xb ee ff | 4 | frPP |
| ORAA [D,xysp] | [D,IDX] | AA xb | 6 | fIfrfP |
| ORAA [opr16,xysp] | [IDX2] | AA xb ee ff | 6 | fIPrfP |

ORAB

Inclusive OR B

ORAB

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

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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| ORAB #opr8 <i>i</i> | IMM | CA ii | 1 | P |
| ORAB opr8 <i>a</i> | DIR | DA dd | 3 | rfP |
| ORAB opr16 <i>a</i> | EXT | FA hh ll | 3 | rOP |
| ORAB oprx0_xysp | IDX | EA xb | 3 | rfP |
| ORAB oprx9_xysp | IDX1 | EA xb ff | 3 | rPO |
| ORAB oprx16_xysp | IDX2 | EA xb ee ff | 4 | frPP |
| ORAB [D,xysp] | [D,IDX] | EA xb | 6 | fIfrfP |
| ORAB [opr16,xysp] | [IDX2] | EA xb ee ff | 6 | fIPrfP |

ORCC

Logical OR CCR with Mask

ORCC

Operation: (CCR) + (M) \Rightarrow 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.

Condition Codes

and Boolean

Formulas:

| 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.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ORCC #opr8i | IMM | 14 ii | 1 | P |

PSHA

Push A onto Stack

PSHA

Operation: $(SP) - \$0001 \Rightarrow SP$
 $(A) \Rightarrow M_{(SP)}$

Description: Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stacked at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHA | INH | 36 | 2 | 0s |

PSHB

Push B onto Stack

PSHB

Operation: $(SP) - \$0001 \Rightarrow SP$
 $(B) \Rightarrow M_{(SP)}$

Description: Stacks the content of accumulator B. The stack pointer is decremented by one. The content of B is then stacked at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHB | INH | 37 | 2 | 0s |

PSHC

Push CCR onto Stack

PSHC

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

Description: Stacks the content of the condition codes register. The stack pointer is decremented by one. The content of the CCR is then stacked at the address to which the SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHC | INH | 39 | 2 | 0s |

PSHD

Push Double Accumulator onto Stack

PSHD

Operation: $(SP) - \$0002 \Rightarrow SP$
 $(A : B) \Rightarrow M_{(SP)} : M_{(SP + 1)}$

Description: Stacks the content of double accumulator D. The stack pointer is decremented by two, then the contents of accumulators A and B are stacked at the location to which the SP points.

After PSHD executes, the SP points to the stacked value of accumulator A. This stacking order is the opposite of the order in which A and B are stacked when an interrupt is recognized. The interrupt stacking order is backward-compatible with the M6800, which had no 16-bit accumulator.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHD | INH | 3B | 2 | os |

PSHX

Push Index Register X onto Stack

PSHX

Operation: $(SP) - \$0002 \Rightarrow SP$
 $(X_H : X_L) \Rightarrow M_{(SP)} : M_{(SP + 1)}$

Description: Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stacked at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHX | INH | 34 | 2 | OS |

PSHY

Push Index Register Y onto Stack

PSHY

Operation: $(SP) - \$0002 \Rightarrow SP$
 $(Y_H : Y_L) \Rightarrow M_{(SP)} : M_{(SP + 1)}$

Description: Stacks the content of index register Y. The stack pointer is decremented by two. The content of Y is then stacked at the address to which the SP points. After PSHY executes, the SP points to the stacked value of the high-order half of Y.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PSHY | INH | 35 | 2 | OS |

PULA

Pull A from Stack

PULA

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

Description: Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULA | INH | 32 | 3 | uFO |

PULB

Pull B from Stack

PULB

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

Description: Accumulator B is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULB | INH | 33 | 3 | uFO |

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 CPU registers that were pushed onto the stack before subroutine execution.

**Condition Codes
and Boolean
Formulas:**

| 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 XIRQ interrupt.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULC | INH | 38 | 3 | uFO |

PULD**Pull Double Accumulator from Stack****PULD**

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B$
 $(SP) + \$0002 \Rightarrow SP$

Description: Double accumulator D is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

The order in which A and B are pulled from the stack is the opposite of the order in which A and B are pulled when an RTI instruction is executed. The interrupt stacking order for A and B is backward-compatible with the M6800, which had no 16-bit accumulator.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

Condition Codes**and Boolean****Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULD | INH | 3A | 3 | UFO |

PULX

Pull Index Register X from Stack

PULX

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$
 $(SP) + \$0002 \Rightarrow SP$

Description: Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULX | INH | 30 | 3 | UFO |

PULY

Pull Index Register Y from Stack

PULY

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L$
 $(SP) + \$0002 \Rightarrow SP$

Description: Index register Y is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| PULY | INH | 31 | 3 | UFO |

REV

Fuzzy Logic Rule Evaluation

REV

Operation: MIN – MAX Rule Evaluation

Description: Performs an unweighted evaluation of a list of rules, using fuzzy input values to produce fuzzy outputs. REV can be interrupted, so it does not adversely affect interrupt latency.

The REV instruction uses an 8-bit offset from a base address stored in index register Y to determine the address of each fuzzy input and fuzzy output. For REV to execute correctly, each rule in the knowledge base must consist of a table of 8-bit antecedent offsets followed by a table of 8-bit consequent offsets. The value \$FE marks boundaries between antecedents and consequents and between successive rules. The value \$FF marks the end of the rule list. REV can evaluate any number of rules with any number of inputs and outputs.

Beginning with the address pointed to by the first rule antecedent, REV evaluates each successive fuzzy input value until it encounters an \$FE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. Then, beginning with the address pointed to by the first rule consequent, the truth value is compared to each successive fuzzy output value until another \$FE separator is encountered; if the truth value is greater than the current output value, it is written to the output. Operation is similar to that of a MAXM instruction. Rules are processed until an \$FF terminator is encountered.

Before executing REV, perform these set up operations.

- X must point to the first 8-bit element in the rule list.
- Y must point to the base address for fuzzy inputs and fuzzy outputs.
- A must contain the value \$FF, and the CCR V bit must = 0.
(LDAA #\$FF places the correct value in A and clears V.)
- Clear fuzzy outputs to 0s.

Index register X points to the element in the rule list that is being evaluated. X is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, X points to the next address after the \$FF separator at the end of the rule list.

REV**Fuzzy Logic Rule Evaluation
(Continued)****REV**

Index register Y points to the base address for the fuzzy inputs and fuzzy outputs. The value in Y does not change during execution.

Accumulator A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A, and writes the smaller of the two to A. When all antecedents have been evaluated, A contains the smallest input value. This is the truth value used during consequent processing. Accumulator A must be initialized to \$FF for the MIN function to evaluate the inputs of the first rule correctly. For subsequent rules, the value \$FF is written to A when an \$FE marker is encountered. At the end of execution, accumulator A holds the truth value for the last rule.

The V status bit signals whether antecedents (0) or consequents (1) are being processed. V must be initialized to 0 for processing to begin with the antecedents of the first rule. Once execution begins, the value of V is automatically changed as \$FE separators are encountered. At the end of execution, V should equal 1, because the last element before the \$FF end marker should be a rule consequent. If V is equal to 0 at the end of execution, the rule list is incorrect.

Fuzzy outputs must be cleared to \$00 before processing begins in order for the MAX algorithm used during consequent processing to work correctly. Residual output values would cause incorrect comparison.

Refer to [Section 9. Fuzzy Logic Support](#) for details.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------|--------------|-------------|-------------------------|-----------------------|
| REV (add if interrupted) | Special | 18 3A | see note ⁽¹⁾ | Orf(ttx)O ff + Orf |

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.

REVV

Fuzzy Logic Rule Evaluation (Weighted)

REVV

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.

REVIEW

Fuzzy Logic Rule Evaluation (Weighted) (Continued)

REVIEW

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 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

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--|--------------|-------------|-------------------------|-------------------------------------|
| REVW (add 2 at end of ins if wts) (add if interrupted) | Special | 18 3B | See note ⁽¹⁾ | ORf(tTx)O (rFFFrf) fff + ORft |

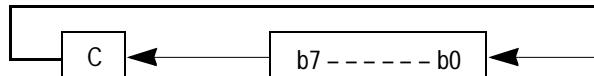
1. The 3-cycle loop in parentheses expands to five cycles for separators when weighting is enabled. The loop is executed once for each element in the rule list. When an interrupt occurs, there is a 2-cycle exit sequence, a 4-cycle re-entry sequence, then execution resumes with a prefetch of the last 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

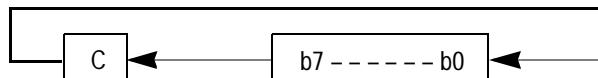
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| ROL opr16a | EXT | 75 hh 11 | 4 | rOPw |
| ROL oprx0_xysp | IDX | 65 xb | 3 | rPw |
| ROL oprx9,xysp | IDX1 | 65 xb ff | 4 | rPOw |
| ROL oprx16,xysp | IDX2 | 65 xb ee ff | 5 | frPPw |
| ROL [D,xysp] | [D,IDX] | 65 xb | 6 | fIfnPw |
| ROL [opr16,xysp] | [IDX2] | 65 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

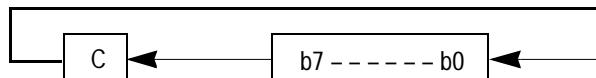
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ROLA | INH | 45 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

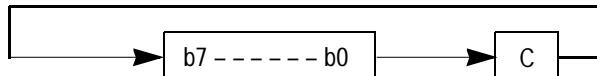
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| ROLB | INH | 55 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

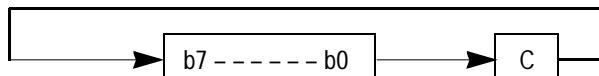
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| ROR opr16a | EXT | 76 hh 11 | 4 | rOPw |
| ROR oprx0_xysp | IDX | 66 xb | 3 | rPw |
| ROR oprx9,xysp | IDX1 | 66 xb ff | 4 | rPOw |
| ROR oprx16,xysp | IDX2 | 66 xb ee ff | 5 | frPPw |
| ROR [D,xysp] | [D,IDX] | 66 xb | 6 | fIfrPw |
| ROR [opr16,xysp] | [IDX2] | 66 xb ee ff | 6 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

**Addressing
Modes, Machine
Code, and
Execution Times:**

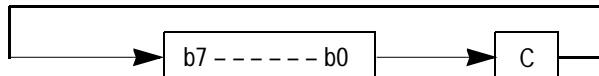
| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| RORA | INH | 46 | 1 | 0 |

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.

Condition Codes and Boolean Formulas:

| 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: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| RORB | INH | 56 | 1 | 0 |

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 M68HC12 devices that do not have expanded memory capability.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| RTC | INH | 0A | 6 | uUnPPP |

RTI

Return from Interrupt

RTI

Operation: $(M_{(SP)}) \Rightarrow CCR; (SP) + \$0001 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) - \$0002 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

Description: Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

If another interrupt is pending when RTI has finished restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched. This operation is functionally identical to the same operation in the M68HC11, where registers actually are re-stacked, but is faster.

**Condition Codes
and Boolean
Formulas:**

| 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 XIRQ interrupt.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------------|--------------|-------------|---------|-------------------------|
| RTI (with interrupt pending) | INH | 0B | 8 10 | uUUUUPPP uUUUUUVfPPP |

RTS

Return from Subroutine

RTS

Operation: $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$

Description: Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| RTS | INH | 3D | 5 | UFPPP |

SBA

Subtract Accumulators

SBA

Operation: $(A) - (B) \Rightarrow A$

Description: Subtracts the content of accumulator B from the content of accumulator A and places the result in A. The content of B is not affected. For subtraction instructions, the C status bit represents a borrow.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| SBA | INH | 18 16 | 2 | 00 |

SBCA

Subtract with Carry from A

SBCA

Operation: $(A) - (M) - C \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + X7 \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| SBCA #opr8i | IMM | 82 ii | 1 | P |
| SBCA opr8a | DIR | 92 dd | 3 | rfp |
| SBCA opr16a | EXT | B2 hh 11 | 3 | rOP |
| SBCA oprx0_xySp | IDX | A2 xb | 3 | rfP |
| SBCA oprx9_xySp | IDX1 | A2 xb ff | 3 | rPO |
| SBCA oprx16_xySp | IDX2 | A2 xb ee ff | 4 | frPP |
| SBCA [D,xySp] | [D,IDX] | A2 xb | 6 | fIfrfP |
| SBCA [opr16,xySp] | [IDX2] | A2 xb ee ff | 6 | fIPrfP |

SBCB

Subtract with Carry from B

SBCB

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

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + X7 \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| SBCB #opr8 <i>i</i> | IMM | C2 ii | 1 | P |
| SBCB opr8 <i>a</i> | DIR | D2 dd | 3 | rfp |
| SBCB opr16 <i>a</i> | EXT | F2 hh 11 | 3 | rOP |
| SBCB oprx0_xy <i>sp</i> | IDX | E2 xb | 3 | rfP |
| SBCB oprx9_xy <i>sp</i> | IDX1 | E2 xb ff | 3 | rPO |
| SBCB oprx16_xy <i>sp</i> | IDX2 | E2 xb ee ff | 4 | frPP |
| SBCB [D,xy <i>sp</i>] | [D,IDX] | E2 xb | 6 | fIfrfP |
| SBCB [opr16,xy <i>sp</i>] | [IDX2] | E2 xb ee ff | 6 | fIPrfP |

SEC

Set Carry

SEC

Operation: $1 \Rightarrow C$ 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 1 |

C: 1; set

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------------------------|--------------|-------------|--------|---------------|
| SEC translates to... ORCC #\$01 | IMM | 14 01 | 1 | P |

SEI

Set Interrupt Mask

SEI

Operation: 1 \Rightarrow I bit

Description: Sets the I mask bit. This instruction is assembled as ORCC #\$10. The ORCC instruction can be used to set any combination of bits in the CCR in one operation. When the I bit is set, all maskable interrupts are inhibited, and the CPU will recognize only non-maskable interrupt sources or an SWI.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | - | - | - | - |

I: 1; set

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------------------------|--------------|-------------|--------|---------------|
| SEI translates to... ORCC #\$10 | IMM | 14 10 | 1 | P |

SEV

Set Two's Complement Overflow Bit

SEV

Operation: $1 \Rightarrow V$ 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | 1 | - |

V: 1; set

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------------------------|--------------|-------------|--------|---------------|
| SEV translates to... ORCC #\$02 | IMM | 14 02 | 1 | P |

SEX

Sign Extend into 16-Bit Register

SEX

Operation: If r1 bit 7 = 0, then \$00 : (r1) \Rightarrow r2
 If r1 bit 7 = 1, then \$FF : (r1) \Rightarrow r2

Description: This instruction is an alternate mnemonic for the TFR r1,r2 instruction, where r1 is an 8-bit register and r2 is a 16-bit register. The result in r2 is the 16-bit sign extended representation of the original two's complement number in r1. The content of r1 is unchanged in all cases except that of SEX A,D (D is A : B).

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|--------------|--------------|----------------------------|--------|---------------|
| SEX abc,dxys | INH | B7 eb | 1 | P |

1. Legal coding for eb is summarized in the following table. Columns represent the high-order digit, and rows represent the low-order digit in hexadecimal (MSB is a don't care).

| | 0 | 1 | 2 |
|---|------------------------------------|------------------------------------|--|
| 3 | sex:A \Rightarrow TMP2 | sex:B \Rightarrow TMP2 | sex:CCR \Rightarrow TMP2 |
| 4 | sex:A \Rightarrow D SEX A,D | sex:B \Rightarrow D SEX B,D | sex:CCR \Rightarrow D SEX CCR,D |
| 5 | sex:A \Rightarrow X SEX A,X | sex:B \Rightarrow X SEX B,X | sex:CCR \Rightarrow X SEX CCR,X |
| 6 | sex:A \Rightarrow Y SEX A,Y | sex:B \Rightarrow Y SEX B,Y | sex:CCR \Rightarrow Y SEX CCR,Y |
| 7 | sex:A \Rightarrow SP SEX A,SP | sex:B \Rightarrow SP SEX B,SP | sex:CCR \Rightarrow SP SEX CCR,SP |

STAA

Store Accumulator A

STAA**Operation:** $(A) \Rightarrow M$ **Description:** Stores the content of accumulator A in memory location M. The content of A is unchanged.**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| STAA <i>opr8a</i> | DIR | 5A dd | 2 | Pw |
| STAA <i>opr16a</i> | EXT | 7A hh 11 | 3 | wOP |
| STAA <i>opr0_xysp</i> | IDX | 6A xb | 2 | Pw |
| STAA <i>opr9_xysp</i> | IDX1 | 6A xb ff | 3 | PwO |
| STAA <i>opr16,xysp</i> | IDX2 | 6A xb ee ff | 3 | PwP |
| STAA [D, <i>xysp</i>] | [D,IDX] | 6A xb | 5 | PIfPw |
| STAA [<i>opr16,xysp</i>] | [IDX2] | 6A xb ee ff | 5 | PIPPw |

STAB

Store Accumulator B

STAB

Operation: $(B) \Rightarrow M$

Description: Stores the content of accumulator B in memory location M. The content of B is unchanged.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| STAB <i>opr8a</i> | DIR | 5B dd | 2 | Pw |
| STAB <i>opr16a</i> | EXT | 7B hh 11 | 3 | wOP |
| STAB <i>opr0_xysp</i> | IDX | 6B xb | 2 | Pw |
| STAB <i>opr9_xysp</i> | IDX1 | 6B xb ff | 3 | PwO |
| STAB <i>opr16,xysp</i> | IDX2 | 6B xb ee ff | 3 | PwP |
| STAB [D, <i>xysp</i>] | [D,IDX] | 6B xb | 5 | PIfPw |
| STAB [<i>opr16,xysp</i>] | [IDX2] | 6B xb ee ff | 5 | PIPPw |

STD

Store Double Accumulator

STD

Operation: $(A : B) \Rightarrow M : M + 1$

Description: Stores the content of double accumulator D in memory location M : M + 1. The content of D is unchanged.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| STD opr8a | DIR | 5C dd | 2 | PW |
| STD opr16a | EXT | 7C hh ll | 3 | WOP |
| STD oprx0_xysp | IDX | 6C xb | 2 | PW |
| STD oprx9,xysp | IDX1 | 6C xb ff | 3 | PWO |
| STD oprx16,xysp | IDX2 | 6C xb ee ff | 3 | PWP |
| STD [D,xysp] | [D,IDX] | 6C xb | 5 | PIfPW |
| STD [opr16,xysp] | [IDX2] | 6C xb ee ff | 5 | PIPPW |

STOP

Stop Processing

STOP

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) – \$0001 ⇒ SP; CCR ⇒ (M_(SP))
 Stop All Clocks

Description: When the S control bit is set, STOP is disabled and operates like a 2-cycle NOP instruction. When the S bit is cleared, STOP stacks CPU context, stops all system clocks, and puts the device in standby mode.

Standby operation minimizes system power consumption. The contents of registers and the states of I/O pins remain unchanged.

Asserting the RESET, XIRQ, or IRQ signals ends standby mode. Stacking on entry to STOP allows the CPU to recover quickly when an interrupt is used, provided a stable clock is applied to the device. If the system uses a clock reference crystal that also stops during low-power mode, crystal startup delay lengthens recovery time.

If XIRQ is asserted while the X mask bit = 0 (XIRQ interrupts enabled), execution resumes with a vector fetch for the XIRQ interrupt. If the X mask bit = 1 (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.

Condition Codes

and Boolean

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

Addressing

Modes, Machine

Code, and

Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------|--------------|-------------|--------|---------------|
| STOP (entering STOP) | INH | 18 3E | 9 | 00SSSfSS |
| (exiting STOP) | | | 5 | fVfPPP |
| (continue) | | | 2 | fO |
| (if STOP disabled) | | | 2 | OO |

STS

Store Stack Pointer

STS

Operation: $(SP_H : SP_L) \Rightarrow M : M + 1$

Description: Stores the content of the stack pointer in memory. The most significant byte of the SP is stored at the specified address, and the least significant byte of the SP is stored at the next higher byte address (the specified address plus one).

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------|--------------|-------------|--------|---------------|
| STS opr8a | DIR | 5F dd | 2 | PW |
| STS opr16a | EXT | 7F hh 11 | 3 | WOP |
| STS oprx0_xy | IDX | 6F xb | 2 | PW |
| STS oprx9_xy | IDX1 | 6F xb ff | 3 | PWO |
| STS oprx16_xy | IDX2 | 6F xb ee ff | 3 | PWP |
| STS [D,xy] | [D,IDX] | 6F xb | 5 | PIFPW |
| STS [opr16_xy] | [IDX2] | 6F xb ee ff | 5 | PIPPW |

STX**Store Index Register X****STX**

Operation: $(X_H : X_L) \Rightarrow M : M + 1$

Description: Stores the content of index register X in memory. The most significant byte of X is stored at the specified address, and the least significant byte of X is stored at the next higher byte address (the specified address plus one).

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------------|--------------|-------------|--------|---------------|
| STX <i>opr8a</i> | DIR | 5E dd | 2 | PW |
| STX <i>opr16a</i> | EXT | 7E hh 11 | 3 | WOP |
| STX <i>opr16,xysp</i> | IDX | 6E xb | 2 | PW |
| STX <i>opr16,xysp</i> | IDX1 | 6E xb ff | 3 | PWO |
| STX <i>opr16,xysp</i> | IDX2 | 6E xb ee ff | 3 | PWP |
| STX [D, <i>xysp</i>] | [D,IDX] | 6E xb | 5 | PIfPW |
| STX [<i>opr16,xysp</i>] | [IDX2] | 6E xb ee ff | 5 | PIPPW |

STY

Store Index Register Y

STY

Operation: $(Y_H : Y_L) \Rightarrow M : M + 1$

Description: Stores the content of index register Y in memory. The most significant byte of Y is stored at the specified address, and the least significant byte of Y is stored at the next higher byte address (the specified address plus one).

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|------------------|--------------|-------------|--------|---------------|
| STY opr8a | DIR | 5D dd | 2 | PW |
| STY opr16a | EXT | 7D hh 11 | 3 | WOP |
| STY oprx0_xysp | IDX | 6D xb | 2 | PW |
| STY oprx9,xysp | IDX1 | 6D xb ff | 3 | PWO |
| STY oprx16,xysp | IDX2 | 6D xb ee ff | 3 | PWP |
| STY [D,xysp] | [D,IDX] | 6D xb | 5 | PIFPW |
| STY [opr16,xysp] | [IDX2] | 6D xb ee ff | 5 | PIPPW |

SUBA

Subtract A

SUBA

Operation: $(A) - (M) \Rightarrow A$

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.

**Condition Codes
and Boolean
Formulas:**

| 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: $X7 \bullet \overline{M7} \bullet \overline{R7} + X7 \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| SUBA #opr8i | IMM | 80 ii | 1 | P |
| SUBA opr8a | DIR | 90 dd | 3 | rP |
| SUBA opr16a | EXT | B0 hh ll | 3 | rOP |
| SUBA oprx0_xySp | IDX | A0 xb | 3 | rP |
| SUBA oprx9_xySp | IDX1 | A0 xb ff | 3 | rPO |
| SUBA oprx16_xySp | IDX2 | A0 xb ee ff | 4 | fPP |
| SUBA [D,xySp] | [D,IDX] | A0 xb | 6 | fIfP |
| SUBA [opr16,xySp] | [IDX2] | A0 xb ee ff | 6 | fIPrfP |

SUBB

Subtract B

SUBB

Operation: $(B) - (M) \Rightarrow B$

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.

**Condition Codes
and Boolean
Formulas:**

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

C: $\overline{X7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{X7}$
Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------------|--------------|-------------|--------|---------------|
| SUBB #opr8i | IMM | C0 ii | 1 | P |
| SUBB opr8a | DIR | D0 dd | 3 | rP |
| SUBB opr16a | EXT | F0 hh ll | 3 | rOP |
| SUBB oprx0_xySp | IDX | E0 xb | 3 | rP |
| SUBB oprx9_xySp | IDX1 | E0 xb ff | 3 | rPO |
| SUBB oprx16_xySp | IDX2 | E0 xb ee ff | 4 | fP |
| SUBB [D,xySp] | [D,IDX] | E0 xb | 6 | fIfP |
| SUBB [opr16,xySp] | [IDX2] | E0 xb ee ff | 6 | fIPrfP |

SUBD

Subtract Double Accumulator

SUBD

Operation: $(A : B) - (M : M + 1) \Rightarrow A : B$

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.

Condition Codes

and Boolean

Formulas:

| 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: $D15 \bullet \overline{M15} \bullet R15 + \overline{D15} \bullet M15 \bullet R15$
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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------------|--------------|-------------|--------|---------------|
| SUBD #opr16i | IMM | 83 jj kk | 2 | OP |
| SUBD opr8a | DIR | 93 dd | 3 | RfP |
| SUBD opr16a | EXT | B3 hh ll | 3 | ROP |
| SUBD oprx0_xyssp | IDX | A3 xb | 3 | RfP |
| SUBD oprx9_xyssp | IDX1 | A3 xb ff | 3 | RPO |
| SUBD oprx16_xyssp | IDX2 | A3 xb ee ff | 4 | fRPP |
| SUBD [D,xyssp] | [D,IDX] | A3 xb | 6 | fIFRfP |
| SUBD [opr16,xyssp] | [IDX2] | A3 xb ee ff | 6 | fIPRfP |

SWI**Software Interrupt****SWI**

Operation: $(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
 $1 \Rightarrow I$
 $(SWI \text{ Vector}) \Rightarrow PC$

Description: Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to [Section 7. Exception Processing](#) for more information.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | - | - | - | - |

I: 1; set

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|--------------------------|
| SWI | INH | 3F | 9 | VSPSSPSSP ⁽¹⁾ |

1. The CPU also uses the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation of the sequence (VfPPP) is used for resets.

TAB

Transfer from Accumulator A to Accumulator B

TAB

Operation: (A) \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| TAB | INH | 18 0E | 2 | 00 |

TAP

**Transfer from Accumulator A
to Condition Code Register**

TAP

Operation: (A) \Rightarrow CCR

Description: Transfers the logic states of bits [7:0] of accumulator A to the corresponding bit positions of the CCR. The content of A remains unchanged. The X mask bit can be cleared as a result of a TAP, but cannot be set if it was cleared prior to execution of the TAP. If the I bit is cleared, there is a 1-cycle delay before the system allows interrupt requests. This prevents interrupts from occurring between instructions in the sequences CLI, WAI and CLI, SEI.

This instruction is accomplished with the TFR A,CCR instruction. For compatibility with the M68HC11, the mnemonic TAP is translated by the assembler.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|----------|--------------|----------|----------|----------|----------|----------|----------|
| Δ | \Downarrow | 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 XIRQ interrupt.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------------|--------------|-------------|--------|---------------|
| TAP translates to... TFR A,CCR | INH | B7 02 | 1 | P |

TBA**Transfer from Accumulator B
to Accumulator A****TBA****Operation:** $(B) \Rightarrow A$ **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.**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| TBA | INH | 18 0F | 2 | 00 |

TBEQ

Test and Branch if Equal to Zero

TBEQ

Operation: If (Counter) = 0, then (PC) + \$0003 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|------------------|--------------|----------------------------|--------|---------------|
| TBEQ abdxys,rel9 | REL | 04 1b rr | 3/3 | PPP |

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

| Count Register | Bits 2:0 | Source Form | Object Code (If Offset is Positive) | Object Code (If Offset is Negative) |
|----------------|----------|---------------|-------------------------------------|-------------------------------------|
| A | 000 | TBEQ A, rel9 | 04 40 rr | 04 50 rr |
| B | 001 | TBEQ B, rel9 | 04 41 rr | 04 51 rr |
| D | 100 | TBEQ D, rel9 | 04 44 rr | 04 54 rr |
| X | 101 | TBEQ X, rel9 | 04 45 rr | 04 55 rr |
| Y | 110 | TBEQ Y, rel9 | 04 46 rr | 04 56 rr |
| SP | 111 | TBEQ SP, rel9 | 04 47 rr | 04 57 rr |

TBL

Table Lookup and Interpolate

TBL

Operation: $(M) + [(B) \times ((M+1) - (M))] \Rightarrow A$

Description: Linearly interpolates one of 256 result values that fall between each pair of data entries in a lookup table stored in memory. Data points in the table represent the 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, set up an index register so that it will point to the starting point (X1) of a line segment when the instruction is executed. X1 is the table entry closest to, but less than or equal to, the desired lookup value. The next table entry after X1 is X2. XL is the X position of the desired lookup point. Load accumulator B with a binary fraction (radix point to left of MSB), representing the ratio $(XL - X1) \div (X2 - X1)$.

The 8-bit unrounded result is calculated using the following expression:

$$A = Y1 + [(B) \times (Y2 - Y1)]$$

Where

$$(B) = (XL - X1) \div (X2 - X1)$$

Y1 = 8-bit data entry pointed to by <effective address>

Y2 = 8-bit data entry pointed to by <effective address> + 1

The intermediate value $[(B) \times (Y2 - Y1)]$ produces a 16-bit result with the radix point between bits 7 and 8. The result in A is the upper 8 bits (integer part) of this intermediate 16-bit value, plus the 8-bit value Y1.

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | Δ | Δ | - | ? |

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

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

C: Undefined

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------|--------------|-------------|--------|---------------|
| TBL oprx0_xysp | IDX | 18 3D xb | 8 | OrfffffP |

TBNE

Test and Branch if Not Equal to Zero

TBNE

Operation: If (Counter) $\neq 0$, then (PC) + \$0003 + Rel \Rightarrow 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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|------------------|--------------|----------------------------|--------|---------------|
| TBNE abdxys,rel9 | REL | 04 1b rr | 3/3 | PPP |

1. Encoding for 1b 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.

**Condition Codes
and Boolean**

Formulas:

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

or

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 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 XIRQ interrupt.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code ⁽¹⁾ | Cycles | Access Detail |
|---------------------|--------------|----------------------------|--------|---------------|
| TFR abcdxys,abcdxys | INH | B7 eb | 1 | P |

1. Legal coding for eb is summarized in the following table. Columns represent the high-order digit, and rows represent the low-order digit in hexadecimal (MSB is a don't-care).

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|------------------------|------------------------|----------------------------|-------------------------|----------|----------------------|----------------------|-----------------------|
| A ⇒ A | B ⇒ A | CCR ⇒ A | TMP3 _L ⇒ A | B ⇒ A | X _L ⇒ A | Y _L ⇒ A | SP _L ⇒ A |
| A ⇒ B | B ⇒ B | CCR ⇒ B | TMP3 _L ⇒ B | B ⇒ B | X _L ⇒ B | Y _L ⇒ B | SP _L ⇒ B |
| A ⇒ CCR | B ⇒ CCR | CCR ⇒ CCR | TMP3 _L ⇒ CCR | B ⇒ CCR | X _L ⇒ CCR | Y _L ⇒ CCR | SP _L ⇒ CCR |
| sex:A ⇒ TMP2 | sex:B ⇒ TMP2 | sex:CCR ⇒ TMP2 | TMP3 ⇒ TMP2 | D ⇒ TMP2 | X ⇒ TMP2 | Y ⇒ TMP2 | SP ⇒ TMP2 |
| sex:A ⇒ D SEX A,D | sex:B ⇒ D SEX B,D | sex:CCR ⇒ D SEX CCR,D | TMP3 ⇒ D | D ⇒ D | X ⇒ D | Y ⇒ D | SP ⇒ D |
| sex:A ⇒ X SEX A,X | sex:B ⇒ X SEX B,X | sex:CCR ⇒ X SEX CCR,X | TMP3 ⇒ X | D ⇒ X | X ⇒ X | Y ⇒ X | SP ⇒ X |
| sex:A ⇒ Y SEX A,Y | sex:B ⇒ Y SEX B,Y | sex:CCR ⇒ Y SEX CCR,Y | TMP3 ⇒ Y | D ⇒ Y | X ⇒ Y | Y ⇒ Y | SP ⇒ Y |
| sex:A ⇒ SP SEX A,SP | sex:B ⇒ SP SEX B,SP | sex:CCR ⇒ SP SEX CCR,SP | TMP3 ⇒ SP | D ⇒ SP | X ⇒ SP | Y ⇒ SP | SP ⇒ SP |

TPA

Transfer from Condition Code Register to Accumulator A

TPA

Operation: (CCR) \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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------------|--------------|-------------|--------|---------------|
| TPA translates to... TFR CCR,A | INH | B7 20 | 1 | P |

TRAP

Unimplemented Opcode Trap

TRAP

Operation: $(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
 $1 \Rightarrow I$
 $(\text{Trap Vector}) \Rightarrow PC$

Description: Traps unimplemented opcodes. There are opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the unimplemented opcodes on page 2, an opcode trap interrupt occurs. Unimplemented opcode traps are essentially interrupts that share the \$FFF8:\$FFF9 interrupt vector.

TRAP uses the next address after the unimplemented opcode as a return address. It stacks the return address, index registers Y and X, accumulators B and A, and the CCR, automatically decrementing the SP before each item is stacked. The I mask bit is then set, the PC is loaded with the trap vector, and instruction execution resumes at that location. This instruction is not maskable by the I bit. Refer to [Section 7](#).

[Exception Processing](#) for more information.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | - | - | - | - |

I: 1; set

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|--------------|--------------|------------------------|--------|---------------|
| TRAP trapnum | INH | \$18 tn ⁽¹⁾ | 11 | OFVSPSSPSSP |

1. The value tn represents an unimplemented page 2 opcode in either of the two ranges \$30 to \$39 or \$40 to \$FF.

TST

Test Memory

TST

Operation: (M) – \$00

Description: Subtracts \$00 from the content of memory location M and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying M.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

Condition Codes

and Boolean
Formulas:

| 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

Addressing Modes, Machine Code, and Execution Times:

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|---------------------|--------------|-------------|--------|---------------|
| TST <i>opr16a</i> | EXT | F7 hh 11 | 3 | rOP |
| TST <i>opr16,xy</i> | IDX | E7 xb | 3 | rfP |
| TST <i>opr16,xy</i> | IDX1 | E7 xb ff | 3 | rPO |
| TST <i>opr16,xy</i> | IDX2 | E7 xb ee ff | 4 | frPP |
| TST [D,xy] | [D,IDX] | E7 xb | 6 | fIfrfP |
| TST [opr16,xy] | [IDX2] | E7 xb ee ff | 6 | fIPrfP |

TSTA

Test A

TSTA

Operation: (A) – \$00

Description: Subtracts \$00 from the content of accumulator A and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying A.

The TSTA instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TSTA. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| TSTA | INH | 97 | 1 | 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.

**Condition Codes
and Boolean
Formulas:**

| 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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-------------|--------------|-------------|--------|---------------|
| TSTB | INH | D7 | 1 | o |

TSX

**Transfer from Stack Pointer
to Index Register X**

TSX

Operation: $(SP) \Rightarrow X$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| TSX translates to... TFR SP,X | INH | B7 75 | 1 | P |

TSY

Transfer from Stack Pointer
to Index Register Y

TSY

Operation: $(SP) \Rightarrow Y$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| TSY translates to... TFR SP,Y | INH | B7 76 | 1 | P |

TXS

**Transfer from Index Register X
to Stack Pointer**

TXS

Operation: $(X) \Rightarrow SP$

Description: This is an alternate mnemonic to transfer index register X value to the stack pointer. The content of X is unchanged.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| TXS translates to... TFR X,SP | INH | B7 57 | 1 | P |

TYS

Transfer from Index Register Y
to Stack Pointer

TYS

Operation: $(Y) \Rightarrow SP$

Description: This is an alternate mnemonic to transfer index register Y value to the stack pointer. The content of Y is unchanged.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| TYS translates to... TFR Y,SP | INH | B7 67 | 1 | P |

WAI

Wait for Interrupt

WAI

Operation: $(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$
Stop CPU Clocks

Description: Puts the CPU into a wait state. Uses the address of the instruction following WAI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked.

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

Upon leaving the wait state, the CPU sets the appropriate interrupt mask bit(s), fetches the vector corresponding to the interrupt sensed, and instruction execution continues at the location the vector points to.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

Although the WAI instruction itself does not alter the condition codes, the interrupt that causes the CPU to resume processing also causes the I mask bit (and the X mask bit, if the interrupt was XIRQ) to be set as the interrupt vector is fetched.

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------|--------------|-------------|--------|---------------|
| WAI (before interrupt) | INH | 3E | 8 | OSSSfSsf |
| WAI (when interrupt comes) | | | 5 | VfPPP |

WAV**Weighted Average****WAV**

Operation: Do until B = 0, leave SOP in Y : D, SOW in X

Partial Product = (M pointed to by X) \times (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 \Rightarrow X; (Y) + \$0001 \Rightarrow Y

(B) - \$01 \Rightarrow 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.

Condition Codes**and Boolean****Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | ? | - | ? | 1 | ? | ? |

Z: 1; set

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

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|-----------------------------|--------------|-------------|-------------------------|----------------------------|
| WAV (add if interrupted) | Special | 18 3C | See note ⁽¹⁾ | Off(frrfffff)O SSSUUUrr |

1. The 8-cycle sequence in parentheses represents the loop for one iteration of SOP and SOW accumulation.

XGDX

**Exchange Double Accumulator
and Index Register X**

XGDX

Operation: $(D) \Leftrightarrow (X)$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| XGDX translates to... EXG D,X | INH | B7 C5 | 1 | P |

XGDY

Exchange Double Accumulator
and Index Register Y

XGDY

Operation: $(D) \Leftrightarrow (Y)$

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.

**Condition Codes
and Boolean
Formulas:**

| S | X | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - |

None affected

**Addressing
Modes, Machine
Code, and
Execution Times:**

| Source Form | Address Mode | Object Code | Cycles | Access Detail |
|----------------------------------|--------------|-------------|--------|---------------|
| XGDY translates to... EXG D,Y | INH | B7 C6 | 1 | P |

Section 7. Exception Processing

7.1 Contents

| | | |
|-------|---|-----|
| 7.2 | Introduction | 323 |
| 7.3 | Types of Exceptions | 324 |
| 7.4 | Exception Priority | 325 |
| 7.5 | Resets. | 326 |
| 7.5.1 | Power-On Reset. | 327 |
| 7.5.2 | External Reset | 327 |
| 7.5.3 | COP Reset | 327 |
| 7.5.4 | Clock Monitor Reset. | 327 |
| 7.6 | Interrupts. | 328 |
| 7.6.1 | Non-Maskable Interrupt Request (XIRQ) | 328 |
| 7.6.2 | Maskable Interrupts | 329 |
| 7.6.3 | Interrupt Recognition | 329 |
| 7.6.4 | External Interrupts | 330 |
| 7.6.5 | Return-from-Interrupt Instruction (RTI) | 330 |
| 7.7 | Unimplemented Opcode Trap. | 331 |
| 7.8 | Software Interrupt Instruction (SWI) | 331 |
| 7.9 | Exception Processing Flow. | 331 |
| 7.9.1 | Vector Fetch. | 333 |
| 7.9.2 | Reset Exception Processing | 333 |
| 7.9.3 | Interrupt and Unimplemented Opcode Trap Exception Processing | 333 |

7.2 Introduction

Exceptions are events that require processing outside the normal flow of instruction execution. This section describes exceptions and the way each is handled.

7.3 Types of Exceptions

Central processor unit (CPU12) exceptions include:

- Resets
- An unimplemented opcode trap
- A software interrupt instruction (SWI)
- X-bit interrupts
- 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 128 bytes of the standard 64-Kbyte address map.

Table 7-1. CPU12 Exception Vector Map

| Vector Address | Source |
|----------------|--------------------------------------|
| \$FFFE–\$FFFF | System Reset |
| \$FFFC–\$FFFD | Clock Monitor Reset |
| \$FFFA–\$FFFB | COP Reset |
| \$FFF8–\$FFF9 | Unimplemented Opcode Trap |
| \$FFF6–\$FFF7 | Software Interrupt Instruction (SWI) |
| \$FFF4–\$FFF5 | $\overline{\text{XIRQ}}$ Signal |
| \$FFF2–\$FFF3 | $\overline{\text{IRQ}}$ Signal |
| \$FFC0–\$FFF1 | Device-Specific Interrupt Sources |

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.

The CPU12 can handle up to 64 exception vectors, but the number actually used varies from device to device, and some vectors are reserved for Motorola use. Refer to device documentation for more information.

Exceptions can be classified by the effect of the X and I interrupt mask bits on recognition of a pending request.

- Resets, the unimplemented opcode trap, and the SWI instruction are not affected by the X and I mask bits.
- Interrupt service requests from the XIRQ pin are inhibited when $X = 1$, but are not affected by the I bit.
- All other interrupts are inhibited when $I = 1$.

7.4 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 [7.6 Interrupts](#) for more detail concerning interrupt priority and servicing.

The priorities of the unmaskable sources are:

1. RESET pin
2. Clock monitor reset
3. Computer operating properly (COP) watchdog reset
4. Non-maskable interrupt request (XIRQ) signal
5. Unimplemented opcode trap
6. Software interrupt instruction (SWI)

An external reset has the highest exception-processing priority, followed by clock monitor reset, and then the on-chip watchdog reset.

The XIRQ interrupt is pseudo-non-maskable. After reset, the X bit in the CCR is set, which inhibits all interrupt service requests from the XIRQ pin until the X bit is cleared. The X bit can be cleared by a program instruction, but program instructions cannot reset X from 0 to 1. Once the X bit is cleared, interrupt service requests made via the XIRQ pin become non-maskable.

The unimplemented page 2 opcode trap (TRAP) and the SWI are special cases. In one sense, these two exceptions have very low priority, because any enabled interrupt source that is pending prior to the time exception processing begins will take precedence. However, once the CPU begins processing a TRAP or SWI, neither can be interrupted. Also, since these are mutually exclusive instructions, they have no relative priority.

All remaining interrupts are subject to masking via the I bit in the CCR. Most M68HC12 microcontroller units (MCU) have an external 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 M68HC12 derivative for more information.

7.5 Resets

M68HC12 devices perform resets with a combination of hardware and software. Integration module circuitry determines the type of reset that has occurred, performs basic system configuration, then passes control to the CPU12. The CPU fetches a vector determined by the type of reset that has occurred, jumps to the address pointed to by the vector, and begins to execute code at that address.

The are four possible sources of reset are:

- Power-on reset (POR)
- External reset
- COP reset
- Clock monitor reset

Power-on reset (POR) and external reset share the same reset vector. The computer operating properly (COP) reset and the clock monitor reset each have a vector.

7.5.1 Power-On Reset

The M68HC12 device integration module incorporates 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.

7.5.2 External Reset

The MCU distinguishes between internal and external resets by sensing how quickly the signal on the 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 RESET signal low for 16 clock cycles, then releases. Eight clock cycles later, the MCU samples the state of the signal applied to the RESET pin. If the signal is still low, an external reset has occurred. If the signal is high, reset has been initiated internally by either the COP system or the clock monitor.

7.5.3 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.

7.5.4 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.

7.6 Interrupts

Each M68HC12 device can recognize a number of interrupt sources. Each source has a vector in the vector table. The 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 M68HC12 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 one maskable source priority over other maskable sources.

7.6.1 Non-Maskable Interrupt Request (XIRQ)

The XIRQ input is an updated version of the non-maskable interrupt (NMI) input of earlier MCUs. The XIRQ function is disabled during system reset and upon entering the interrupt service routine for an 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 XIRQ signal. After minimum system initialization, software can clear the X bit using an instruction such as ANDCC #\$BF. Software cannot reset the X bit from 0 to 1 once it has been cleared, and interrupt requests made via the 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.

7.6.2 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 integration 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.

7.6.3 Interrupt Recognition

Once enabled, an interrupt request can be recognized at any time after the I mask bit is cleared. When an interrupt service request is recognized, the CPU responds at the completion of the instruction being executed. Interrupt latency varies according to the number of cycles required to complete the current instruction. Because the fuzzy logic rule evaluation (REV), fuzzy logic rule evaluation weighted (REVW), and weighted average (WAV) instructions can take many cycles to complete, they are designed so that they can be interrupted. Instruction execution resumes when interrupt execution is complete. When the CPU begins to service an interrupt, the instruction queue is refilled, a return address is calculated, and then the return address and the contents of the CPU registers are stacked as shown in [Table 7-2](#).

Table 7-2. Stacking Order on Entry to Interrupts

| Memory Location | CPU Registers |
|-----------------|-------------------------------------|
| SP + 7 | RTN _H : RTN _L |
| SP + 5 | Y _H : Y _L |
| SP + 3 | X _H : X _L |
| SP + 1 | B : A |
| SP | CCR |

After the CCR is stacked, the I bit (and the X bit, if an 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.

7.6.4 External Interrupts

External interrupt service requests are made by asserting an active-low signal connected to the IRQ pin. Typically, control bits in the device integration module affect how the signal is detected and recognized.

The I bit serves as the IRQ interrupt enable flag. When an 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.

7.6.5 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 a 10-cycle instruction when another interrupt is pending. In either case, the first five cycles are used to restore (pull) the CCR, B:A, X, Y, and the return address from the stack. If no other interrupt is pending at this point, three program words are fetched to refill the instruction queue from the area of the return address and processing proceeds from there.

If another interrupt is pending after registers are restored, a new vector is fetched, and the stack pointer is adjusted to point at the CCR value that was just recovered ($SP = SP - 9$). This makes it appear that the registers have been stacked again. After the SP is adjusted, three program words are fetched to refill the instruction queue, starting at the address the vector points to. Processing then continues with execution of the instruction that is now at the head of the queue.

7.7 Unimplemented Opcode Trap

The CPU12 has opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the 202 unused opcodes on page 2, an unimplemented opcode trap occurs. The 202 unimplemented opcodes are essentially interrupts that share a common interrupt vector, \$FFF8:\$FFF9.

The CPU12 uses the next address after an unimplemented page 2 opcode as a return address. This differs from the M68HC11 illegal opcode interrupt, which uses the address of an illegal opcode as the return address. In the CPU12, the stacked return address can be used to calculate the address of the unimplemented opcode for software-controlled traps.

7.8 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.

7.9 Exception Processing Flow

The first cycle in the exception processing flow for all CPU12 exceptions is the same, regardless of the source of the exception. Between the first and second cycles of execution, the CPU chooses one of three alternative paths. The first path is for resets, the second path is for pending X or I interrupts, and the third path is used for software interrupts (SWI) and trapping unimplemented opcodes. The last two paths are virtually identical, differing only in the details of calculating the return address. Refer to [Figure 7-1](#) for the following discussion.

Exception Processing

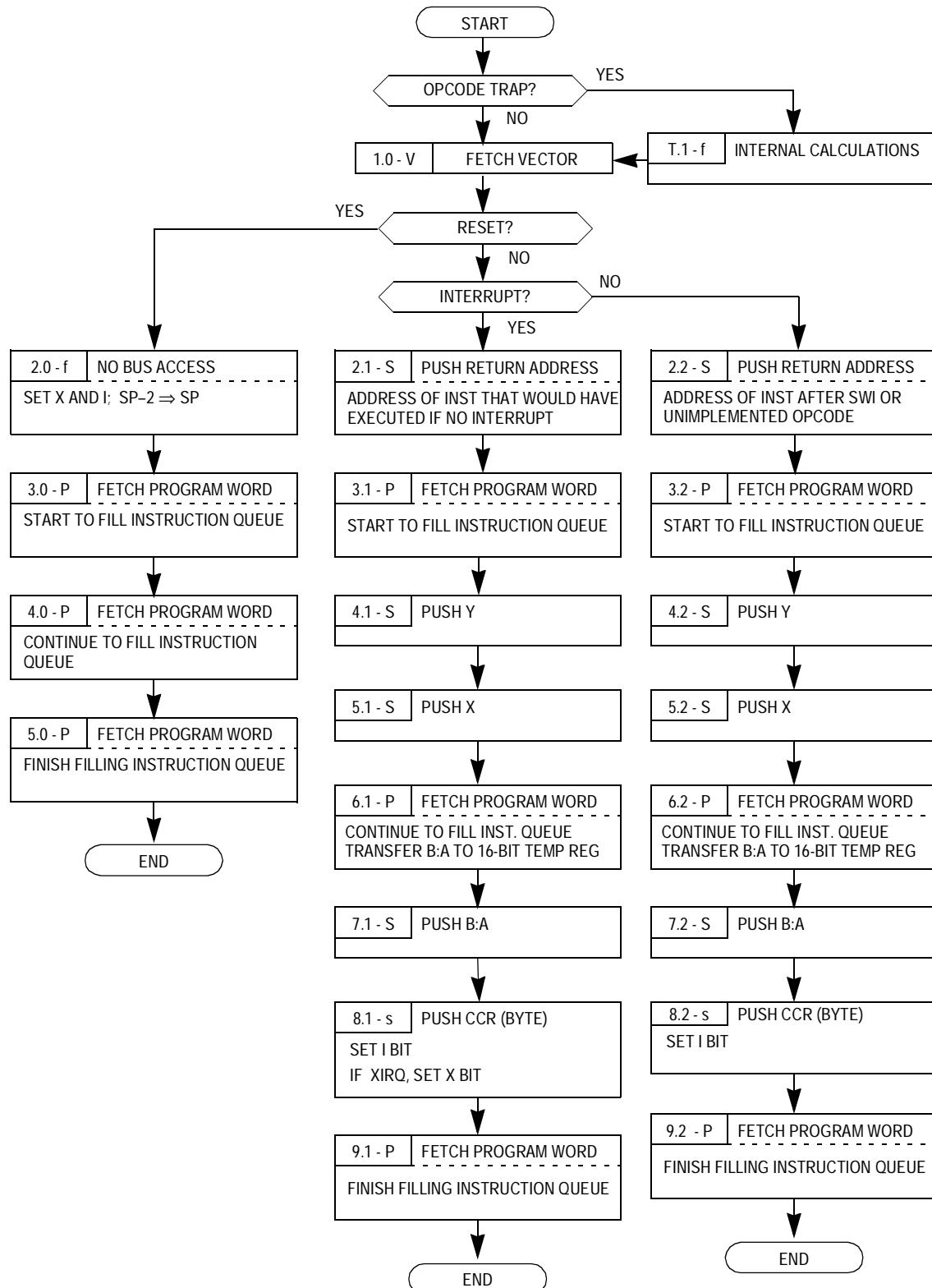


Figure 7-1. Exception Processing Flow Diagram

7.9.1 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 (\$FFC0–\$FFFF). The CPU cannot use the fetched vector until the third cycle of the exception processing sequence.

During the vector fetch cycle, the CPU issues a signal that tells the integration module to drive the vector address of the highest priority, pending exception onto the system address bus (the CPU does not provide this address).

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

7.9.2 Reset Exception Processing

If reset caused the exception, processing continues to cycle 2.0. This cycle sets the X and I bits in the CCR. The stack pointer is also decremented by two, but this is an artifact of shared code used for interrupt processing; the SP is not intended to have any specific value after a reset. Cycles 3.0 through 5.0 are program word fetches that refill the instruction queue. Fetches start at the address pointed to by the reset vector. When the fetches are completed, exception processing ends, and the CPU starts executing the instruction at the head of the instruction queue.

7.9.3 Interrupt and Unimplemented Opcode Trap Exception Processing

If an exception was not caused by a reset, a return address is calculated.

- Cycles 2.1 and 2.2 are both S cycles (a 16-bit word), but the cycles are not identical because 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 [7.7 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 X mask bit manipulation performed in cycle 8.1.
- Cycle 3.1/3.2 is the first of three program word fetches that refill the instruction queue.
- Cycle 4.1/4.2 pushes Y onto the stack.
- Cycle 5.1/5.2 pushes X onto the stack.
- Cycle 6.1/6.2 is the second of three program word fetches that refill the instruction queue. During this cycle, the contents of the A and B accumulators are concatenated into a 16-bit word in the order B:A. This makes register order in the stack frame the same as that of the M68HC11, M6801, and the M6800.
- Cycle 7.1/7.2 pushes the 16-bit word containing B:A onto the stack.
- Cycle 8.1/8.2 pushes the 8-bit CCR onto the stack, then updates the mask bits.
 - When an XIRQ interrupt causes an exception, both X and I are set, which inhibits further interrupts during exception processing.
 - When any other interrupt causes an exception, the I bit is set, but the X bit is not changed.
- Cycle 9.1/9.2 is the third of three program word fetches that refill the instruction queue. It is the last cycle of exception processing. After this cycle the CPU starts executing the first cycle of the instruction at the head of the instruction queue.

Section 8. Development and Debug Support

8.1 Contents

| | | |
|---------|---|-----|
| 8.2 | Introduction | 336 |
| 8.3 | External Reconstruction of the Queue | 336 |
| 8.4 | Instruction Queue Status Signals | 337 |
| 8.4.1 | Zero Encoding (0:0) | 338 |
| 8.4.2 | LAT — Latch Data from Bus Encoding (0:1) | 339 |
| 8.4.3 | ALD — Advance and Load from Data Bus Encoding (1:0) | 339 |
| 8.4.4 | ALL — Advance and Load from Latch Encoding (1:1) | 339 |
| 8.4.5 | INT — Interrupt Sequence Encoding (0:1) | 339 |
| 8.4.6 | SEV — Start Instruction on Even Address Encoding (1:0) | 340 |
| 8.4.7 | SOD — Start Instruction on Odd Address Encoding (1:1) | 340 |
| 8.5 | Implementing Queue Reconstruction | 340 |
| 8.5.1 | Queue Status Registers | 341 |
| 8.5.1.1 | in_add, in_dat Registers | 341 |
| 8.5.1.2 | fetch_add, fetch_dat Registers | 341 |
| 8.5.1.3 | st1_add, st1_dat Registers | 342 |
| 8.5.1.4 | st2_add, st2_dat Registers | 342 |
| 8.5.2 | Reconstruction Algorithm | 342 |
| 8.5.2.1 | LAT Decoding | 343 |
| 8.5.2.2 | ALD Decoding | 343 |
| 8.5.2.3 | ALL Decoding | 343 |
| 8.6 | Background Debug Mode | 344 |
| 8.6.1 | Enabling BDM | 344 |
| 8.6.2 | BDM Serial Interface | 345 |
| 8.6.3 | BDM Commands | 347 |
| 8.6.4 | BDM Registers | 350 |

| | | |
|-------|--------------------------------|-----|
| 8.7 | Instruction Tagging | 352 |
| 8.8 | Breakpoints | 353 |
| 8.8.1 | Breakpoint Type | 353 |
| 8.8.2 | Breakpoint Operation | 354 |

8.2 Introduction

This section is an explanation of central processor unit (CPU12) related aspects of the background debugging system. Topics include:

- Instruction queue status signals
- Instruction tagging
- Single-wire background debug interface

8.3 External Reconstruction of the Queue

The CPU12 uses an instruction queue to buffer program information and increase instruction throughput. The queue consists of two 16-bit stages, plus a 16-bit holding latch. Program information is always fetched in aligned 16-bit words. At least three bytes of program information are available to the CPU when instruction execution begins. The holding latch is used when a word of program information arrives before the queue can advance.

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

Two external pins, IPIPE1 and IPIPE0, provide time-multiplexed information about data movement in the queue and instruction execution. To complete the picture for system debugging, it is also necessary to include program information and associated addresses in the reconstructed queue.

The instruction queue and cycle-by-cycle activity can be reconstructed in real time or from trace history captured by a logic analyzer. However, neither scheme can be used to stop the CPU12 at a specific instruction.

By the time an operation is visible outside the MCU, the instruction has already begun execution. A separate instruction tagging mechanism is provided for this purpose. A tag follows the information in the queue as the queue is advanced. During debugging, the CPU enters active background debug mode when a tagged instruction reaches the head of the queue, rather than executing the tagged instruction. For more information about tagging, refer to [8.7 Instruction Tagging](#).

8.4 Instruction Queue Status Signals

The IPIPE1 and IPIPE0 signals carry time-multiplexed information about data movement and instruction execution during normal CPU operation. The signals are available on two multifunctional device pins. During reset, the pins are used as mode-select input signals MODA and MODB. After reset, information on the pins does not become valid until an instruction reaches queue stage 2.

To reconstruct the queue, the information carried by the status signals must be captured externally. In general, data movement and execution start information are considered to be distinct 2-bit values, with the low-order bit on IPIPE0 and the high-order bit on IPIPE1. Data movement information is available on rising edges of the E clock; execution start information is available on falling edges of the E clock, as shown in [Figure 8-1](#). Data movement information refers to data on the bus at the previous falling edge of E. Execution information refers to the bus cycle from the current falling edge to the next falling edge of E. [Table 8-1](#) summarizes the information encoded on the IPIPE1 and IPIPE0 pins.

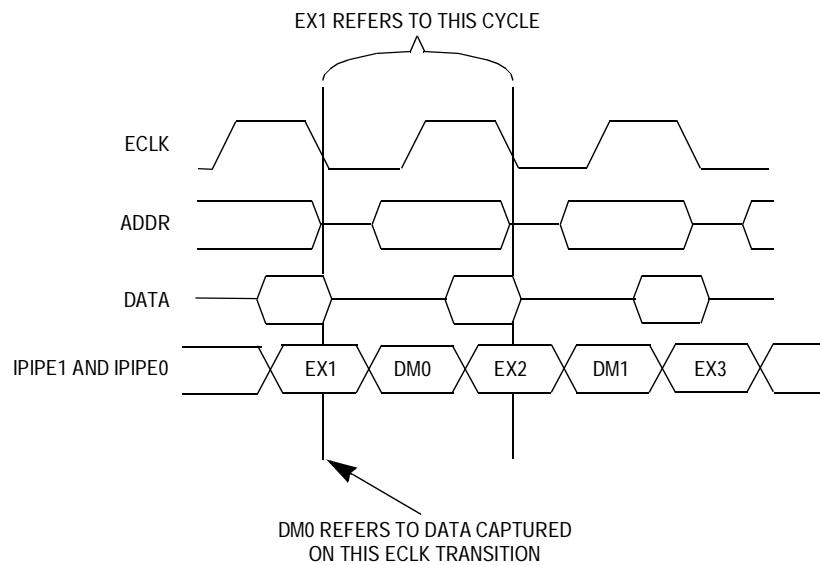


Figure 8-1. Queue Status Signal Timing

Table 8-1. IPIPE1 and IPIPE0 Decoding

| Data Movement | Mnemonic | Meaning |
|--------------------------|----------|-----------------------------------|
| Capture at E Rise | | |
| 0:0 | — | No movement |
| 0:1 | LAT | Latch data from bus |
| 1:0 | ALD | Advance queue and load from bus |
| 1:1 | ALL | Advance queue and load from latch |
| Capture at E Fall | | |
| 0:0 | — | No start |
| 0:1 | INT | Start interrupt sequence |
| 1:0 | SEV | Start even instruction |
| 1:1 | SOD | Start odd instruction |

8.4.1 Zero Encoding (0:0)

The 0:0 state at the rising edge of E indicates that there was no data movement in the instruction queue during the previous cycle; the 0:0 state at the falling edge of E indicates continuation of an instruction or interrupt sequence.

8.4.2 LAT — Latch Data from Bus Encoding (0:1)

Fetched program information has arrived, but the queue is not ready to advance. The information is latched into the buffer. Later, when the queue does advance, stage 1 is refilled from the buffer or from the data bus if the buffer is empty. In some instruction sequences, there can be several latch cycles before the queue advances. In these cases, the buffer is filled on the first latch event and additional latch requests are ignored.

8.4.3 ALD — Advance and Load from Data Bus Encoding (1:0)

The 2-stage instruction queue is advanced by one word and stage 1 is refilled with a word of program information from the data bus. The CPU requested the information two bus cycles earlier but, due to access delays, the information was not available until the E cycle immediately prior to the ALD.

8.4.4 ALL — Advance and Load from Latch Encoding (1:1)

The 2-stage instruction queue is advanced by one word and stage 1 is refilled with a word of program information from the buffer. The information was latched from the data bus at the falling edge of a previous E cycle because the instruction queue was not ready to advance when it arrived.

8.4.5 INT — Interrupt Sequence Encoding (0:1)

The E cycle starting at this E fall 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 only the upper byte of the 16-bit interrupt vector.

8.4.6 SEV — Start Instruction on Even Address Encoding (1:0)

The E cycle starting at this E fall 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.

8.4.7 SOD — Start Instruction on Odd Address Encoding (1:1)

The E cycle starting at this E fall 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.

8.5 Implementing Queue Reconstruction

The raw signals required for queue reconstruction are the address bus (ADDR), the data bus (DATA), the read/write strobe (R/W), the system clock (E), and the queue status signals (IPIPE1 and IPIPE0). An E-clock cycle begins after an E fall. Addresses, R/W state, and data movement status must be captured at the E rise in the middle of the cycle. Data and execution start status must be captured at the E fall at the end of the cycle. These captures can then be organized into records with one record per E clock cycle.

Implementation details depend upon the type of device and the mode of operation. For instance, the data bus can be eight bits or 16 bits wide, and non-multiplexed or multiplexed. In all cases, the externally reconstructed queue must use 16-bit words. Demultiplexing and assembly of 8-bit data into 16-bit words is done before program information enters the real queue, so it must also be done for the external reconstruction.

An example:

Systems with an 8-bit data bus and a program stored in external memory require two cycles for each program word fetch. MCU bus control logic freezes the CPU clocks long enough to do two 8-bit accesses rather than a single 16-bit access, so the CPU sees only 16-bit words of program information. To recover the 16-bit program words externally, latch the data bus state at the falling edge of E when ADDR0 = 0, and gate the outputs of the latch onto DATA15–DATA8 when a LAT or ALD cycle occurs. Since the 8-bit data bus is connected to DATA7–DATA0, the 16-bit word on the data lines corresponds to the ALD or LAT status indication at the E rise after the second 8-bit fetch, which is always to an odd address. IPIPE1 and IPIPE0 status signals indicate 0:0 at the beginning (E fall) and middle (E rise) of the first 8-bit fetch.

Some M68HC12 devices have address lines to support memory expansion beyond the standard 64-Kbyte address space. When memory expansion is used, expanded addresses must also be captured and maintained.

8.5.1 Queue Status Registers

Queue reconstruction requires these registers, which can be implemented as software variables when previously captured trace data is used or as hardware latches in real time.

8.5.1.1 *in_add, in_dat* Registers

These registers contain the address and data from the previous external bus cycle. Depending on how records are read and processed from the raw capture information, it may be possible to simply read this information from the raw capture data file when needed.

8.5.1.2 *fetch_add, fetch_dat* Registers

These registers buffer address and data for information that was fetched before the queue was ready to advance.

8.5.1.3 *st1_add, st1_dat Registers*

These registers contain address and data for the first stage of the reconstructed instruction queue.

8.5.1.4 *st2_add, st2_dat Registers*

These registers contain address and data for the final stage of the reconstructed instruction queue. When the IPIPE1 and IPIPE0 signals indicate that an instruction is starting to execute, the address and opcode can be found in these registers.

8.5.2 Reconstruction Algorithm

This subsection describes in detail how to use IPIPE1 and IPIPE0 signals and status storage registers to perform queue reconstruction. An “is_full” flag is used to indicate when the fetch_add and fetch_dat buffer registers contain information. The use of the flag is explained more fully in subsequent paragraphs.

Typically, the first few cycles of raw capture data are not useful because it takes several cycles before an instruction propagates to the head of the queue. During these first raw cycles, the only meaningful information available are data movement signals. Information on the external address and data buses during this setup time reflects the actions of instructions that were fetched before data collection started.

In the special case of a reset, there is a 5-cycle sequence (VfPPP) during which the reset vector is fetched and the instruction queue is filled, before execution of the first instruction begins. Due to the timing of the switchover of the IPIPE1 and IPIPE0 pins from their alternate function as mode select inputs, the status information on these two pins may be erroneous during the first cycle or two after the release of reset. This is not a problem because the status is correct in time for queue reconstruction logic to correctly replicate the queue.

Before starting to reconstruct the queue, clear the is_full flag to indicate that there is no meaningful information in the fetch_add and fetch_dat buffers. Further movement of information in the instruction queue is based on the decoded status on the IPIPE1 and IPIPE0 signals at the rising edges of E.

8.5.2.1 LAT Decoding

On a latch cycle (LAT), check the `is_full` flag. If and only if `is_full` = 0, transfer the address and data from the previous bus cycle (`in_add` and `in_dat`) into the `fetch_add` and `fetch_dat` registers, respectively. Then, set the `is_full` flag. The usual reason for a latch request instead of an advance request is that the previous instruction ended with a single aligned byte of program information in the last stage of the instruction queue. Since the odd half of this word still holds the opcode for the next instruction, the queue cannot advance on this cycle. However, the cycle to fetch the next word of program information has already started and the data is on its way.

8.5.2.2 ALD Decoding

On an advance-and-load-from-data-bus (ALD) cycle, the information in the instruction queue must advance by one stage. Whatever was in stage 2 of the queue is simply thrown away. The previous contents of stage 1 are moved to stage 2, and the address and data from the previous cycle (`in_add` and `in_dat`) are transferred into stage 1 of the instruction queue. Finally, clear the `is_full` flag to indicate the buffer latch is ready for new data. Usually, there would be no useful information in the fetch buffer when an ALD cycle was encountered, but in the case of a change-of-flow, any data that was there needs to be flushed out (by clearing the `is_full` flag).

8.5.2.3 ALL Decoding

On an advance-and-load-from-latch (ALL) cycle, the information in the instruction queue must advance by one stage. Whatever was in stage 2 of the queue is simply thrown away. The previous contents of stage 1 are moved to stage 2, and the contents of the fetch buffer latch are transferred into stage 1 of the instruction queue. One or more cycles preceding the ALL cycle will have been a LAT cycle. After updating the instruction queue, clear the `is_full` flag to indicate the fetch buffer is ready for new information.

8.6 Background Debug Mode

M68HC12 MCUs include a resident debugging system. This system is implemented with on-chip hardware rather than external software and provides a full set of debugging options. The debugging system is less intrusive than systems used on other microcontrollers, because the control logic resides in the on-chip integration module, rather than in the CPU. Some activities, such as reading and writing memory locations, can be performed while the CPU is executing normal code with no effect on real-time system activity.

The integration module generally uses CPU dead cycles to execute debugging commands while the CPU is operating normally, but can steal cycles from the CPU when necessary. Other commands are firmware based, and require that the CPU be in active background debug mode (BDM) for execution. While BDM is active, the CPU executes a monitor program located in a small on-chip ROM.

Debugging control logic communicates with external devices serially, via the BKGD pin. This single-wire approach helps to minimize the number of pins needed for development support.

Background debug does not operate in stop mode.

8.6.1 Enabling BDM

The debugger must be enabled before it can be activated. Enabling has two phases. First, the BDM read-only memory (ROM) must be enabled by writing the ENBDM bit in the BDM status register, using a debugging command sent via the single wire interface. Once the ROM is enabled, it remains available until the next system reset or until ENBDM is cleared by another debugging command. Second, BDM must be activated to map the ROM and BDM control registers to addresses \$FF00 to \$FFFF and put the MCU in background mode.

After the firmware is enabled, BDM can be activated by the hardware BACKGROUND command, by breakpoints tagged via the LIM breakpoint logic or the BDM tagging mechanism, and by the enter background debug mode (BGND) instruction. An attempt to activate BDM before firmware has been enabled causes the MCU to resume normal instruction execution after a brief delay.

BDM becomes active at the next instruction boundary following execution of the BDM BACKGROUND command. Breakpoints can be configured to activate BDM before a tagged instruction is executed.

While BDM is active, BDM control registers are mapped to addresses \$FF00 to \$FF06. These registers are only accessible through BDM firmware or BDM hardware commands. Registers are described in [8.6.4 BDM Registers](#).

Some M68HC12 on-chip peripherals have a BDM control bit, which determines whether the peripheral function is available during BDM. If no bit is shown, the peripheral is active in BDM.

8.6.2 BDM Serial Interface

The BDM serial interface uses a clocking scheme in which the external host generates a falling edge on the BKGD pin to indicate the start of each bit time. This falling edge must be sent for every bit, whether data is transmitted or received.

BKGD is an open-drain pin that can be driven either by the MCU or by an external host. Data is transferred most significant bit (MSB) first, at 16 E-clock cycles per bit. The interface times out if 512 E-clock cycles occur between falling edges from the host. The hardware clears the command register when a time-out occurs.

The BKGD pin is used to send and receive data. [Figure 8-2](#), [Figure 8-3](#), and [Figure 8-4](#) show timing for each of these cases. Interface timing is synchronous to MCU clocks, but the external host is asynchronous to the target MCU. The internal clock signal is shown for reference in counting cycles.

[Figure 8-2](#) shows an external host transmitting a data bit to the BKGD pin of a target M68HC12 MCU. The host is asynchronous to the target, so there is a 0- to 1-cycle delay from the host-generated falling edge to the time when the target perceives the bit. Ten target E-cycles later, the target senses the bit level on the BKGD pin. The host can drive high during host-to-target transmission to speed up rising edges, because the target does not drive the pin during this time.

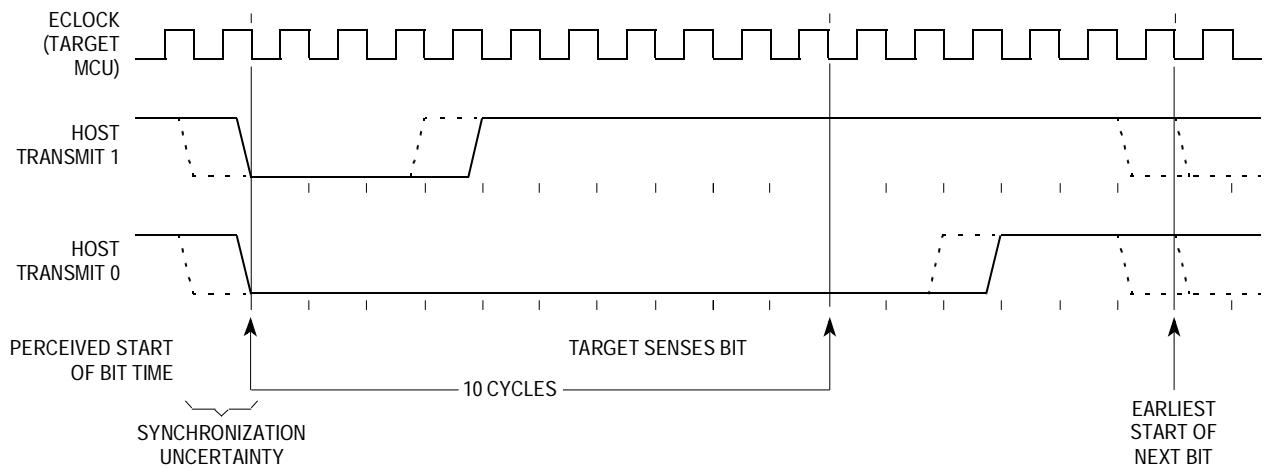


Figure 8-2. BDM Host to Target Serial Bit Timing

Figure 8-3 shows an external host receiving a logic 1 from the target MCU. Since the host is asynchronous to the target, there is a 0- or 1-cycle delay from the host-generated falling edge on BKGD until the target perceives the bit. The host holds the signal low long enough for the target to recognize it (a minimum of two target E-clock cycles), but must release the low drive before the target begins to drive the active-high speed-up pulse seven cycles after the start of the bit time. The host should sample the bit level about 10 cycles after the start of bit time.

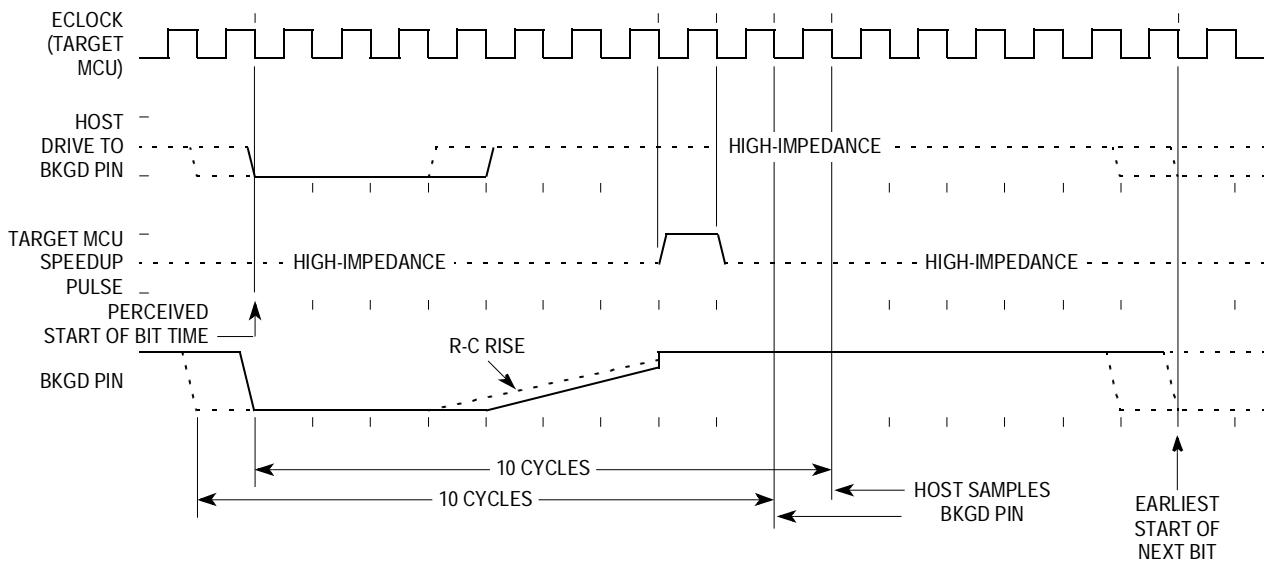


Figure 8-3. BDM Target to Host Serial Bit Timing (Logic 1)

Figure 8-4 shows the host receiving a logic 0 from the target. Since the host is asynchronous to the target, there is a 0- or 1-cycle delay from the host-generated falling edge on BKGD until the target perceives the bit. The host initiates the bit time, but the target finishes it. To make certain the host receives a logic 0, the target drives the BKGD pin low for 13 E-clock cycles, then briefly drives the signal high to speed up the rising edge. The host samples the bit level about 10 cycles after starting the bit time.

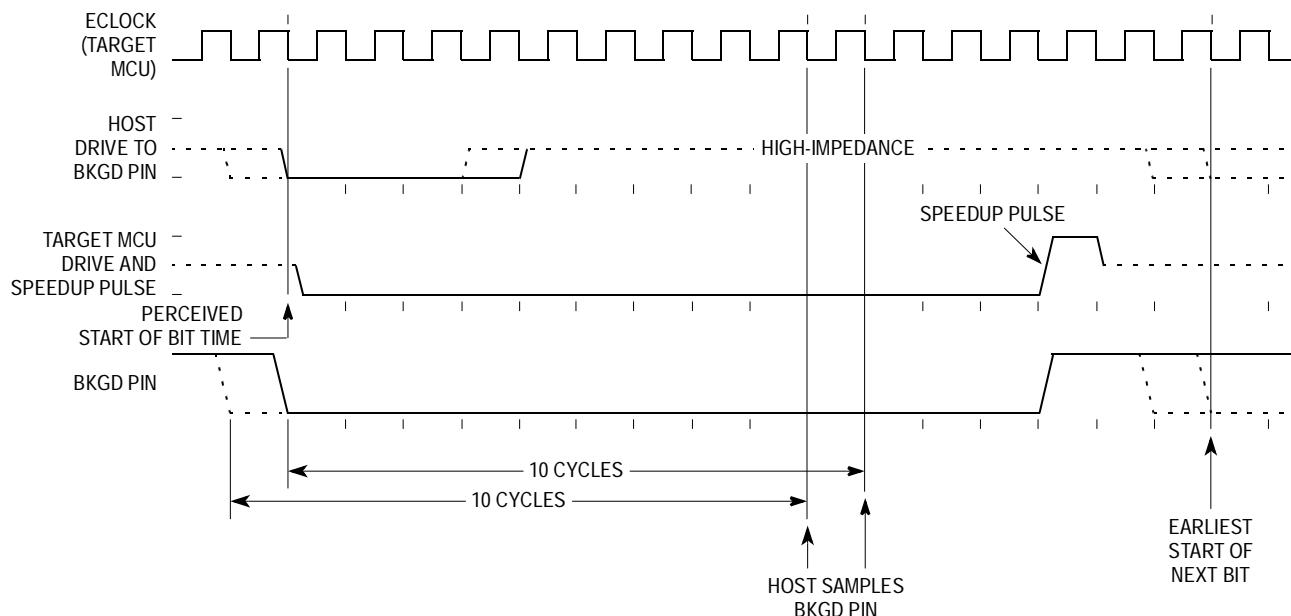


Figure 8-4. BDM Target to Host Serial Bit Timing (Logic 0)

8.6.3 BDM Commands

All BDM opcodes are eight bits long and can be followed by an address or data, as indicated by the instruction.

Commands implemented in BDM control hardware are listed in **Table 8-2**. These commands, except for BACKGROUND, do not require the CPU to be in BDM mode for execution. The control logic uses CPU dead cycles to execute these instructions. If a dead cycle cannot be found within 128 cycles, the control logic steals cycles from the CPU.

Table 8-2. BDM Commands Implemented in Hardware

| Command | Opcode (Hex) | Data | Description |
|--------------------------------|--------------|-----------------------------------|--|
| BACKGROUND | 90 | None | Enter background mode (if firmware enabled). |
| READ_BD_BYTE | E4 | 16-bit address 16-bit data out | Read from memory with BDM in map (may steal cycles if external access) data for odd address on low byte, data for even address on high byte. |
| STATUS ⁽¹⁾ | E4 | FF01, 0000 0000 (out) | READ_BD_BYTE \$FF01. Running user code (BGND instruction is not allowed). |
| | | FF01, 1000 0000 (out) | READ_BD_BYTE \$FF01. BGND instruction is allowed. |
| | | FF01, 1100 0000 (out) | READ_BD_BYTE \$FF01. Background mode active (waiting for single wire serial command). |
| READ_BD_WORD | EC | 16-bit address 16-bit data out | Read from memory with BDM in map (may steal cycles if external access) must be aligned access. |
| READ_BYTE | E0 | 16-bit address 16-bit data out | Read from memory with BDM out of map (may steal cycles if external access) data for odd address on low byte, data for even address on high byte. |
| READ_WORD | E8 | 16-bit address 16-bit data out | Read from memory with BDM out of map (may steal cycles if external access) must be aligned access. |
| WRITE_BD_BYTE | C4 | 16-bit address 16-bit data in | Write to memory with BDM in map (may steal cycles if external access) data for odd address on low byte, data for even address on high byte. |
| ENABLE_FIRMWARE ⁽²⁾ | C4 | FF01, 1xxx xxxx(in) | Write byte \$FF01, set the ENBDM bit. This allows execution of commands which are implemented in firmware. Typically, read STATUS, or in the MSB, write the result back to STATUS. |
| WRITE_BD_WORD | CC | 16-bit address 16-bit data in | Write to memory with BDM in map (may steal cycles if external access) must be aligned access. |
| WRITE_BYTE | C0 | 16-bit address 16-bit data in | Write to memory with BDM out of map (may steal cycles if external access) data for odd address on low byte, data for even address on high byte. |
| WRITE_WORD | C8 | 16-bit address 16-bit data in | Write to memory with BDM out of map (may steal cycles if external access) must be aligned access. |

1. STATUS command is a specific case of the READ_BD_BYTE command.

2. ENABLE_FIRMWARE is a specific case of the WRITE_BD_BYTE command.

The CPU must be in background mode to execute commands that are implemented in the BDM ROM. The CPU executes code from the ROM to perform the requested operation. These commands are shown in **Table 8-3**.

The host controller must wait 150 cycles for a non-intrusive BDM command to execute before another command can be sent. This delay includes 128 cycles for the maximum delay for a dead cycle.

BDM logic retains control of the internal buses until a read or write is completed. If an operation can be completed in a single cycle, it does not intrude on normal CPU operation. However, if an operation requires multiple cycles, CPU clocks are frozen until the operation is complete.

Table 8-3. BDM Firmware Commands

| Command | Opcode (Hex) | Data | Description |
|------------|--------------|-----------------|---|
| GO | 08 | none | Resume normal processing |
| TRACE1 | 10 | none | Execute one user instruction then return to BDM |
| TAGGO | 18 | none | Enable tagging then resume normal processing |
| WRITE_NEXT | 42 | 16-bit data in | X = X + 2; Write next word @ 0,X |
| WRITE_PC | 43 | 16-bit data in | Write program counter |
| WRITE_D | 44 | 16-bit data in | Write D accumulator |
| WRITE_X | 45 | 16-bit data in | Write X index register |
| WRITE_Y | 46 | 16-bit data in | Write Y index register |
| WRITE_SP | 47 | 16-bit data in | Write stack pointer |
| READ_NEXT | 62 | 16-bit data out | X = X + 2; Read next word @ 0,X |
| READ_PC | 63 | 16-bit data out | Read program counter |
| READ_D | 64 | 16-bit data out | Read D accumulator |
| READ_X | 65 | 16-bit data out | Read X index register |
| READ_Y | 66 | 16-bit data out | Read Y index register |
| READ_SP | 67 | 16-bit data out | Read stack pointer |

8.6.4 BDM Registers

Seven BDM registers are mapped into the standard 64-Kbyte address space when BDM is active. Mapping is shown in **Table 8-4**.

Table 8-4. BDM Register Mapping

| Address | Register |
|---------------|----------------------------|
| \$FF00 | BDM instruction register |
| \$FF01 | BDM status register |
| \$FF02-\$FF03 | BDM shift register |
| \$FF04-\$FF05 | BDM address register |
| \$FF06 | BDM condition coderegister |

The content of the instruction register is determined by the type of background instruction being executed. The status register indicates BDM operating conditions. The shift register contains data being received or transmitted via the serial interface. The address register is temporary storage for BDM commands. The CCR preserves the content of the CPU12 CCR while BDM is active.

The only register of interest to users is the status register. The other BDM registers are used only by the BDM firmware to execute commands. The registers can be accessed by means of the hardware READ_BD and WRITE_BD commands, but must not be written during BDM operation.

Address: \$FF01

| | Bit 7 | 6 | 5 | 4 | 3 | 2 | 1 | Bit 0 |
|--------|-------|--------|-------|-----|-------|---|---|-------|
| Read: | ENBDM | BDMACT | ENTAG | SDV | TRACE | 0 | 0 | 0 |
| Write: | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reset: | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Figure 8-5. BDM Status Register (STATUS)

ENBDM — Enable BDM ROM Bit

Shows whether the BDM ROM is enabled. Cleared by reset.

0 = BDM ROM not enabled

1 = BDM ROM enabled, but not in memory map unless BDM is active

BDMACT — BDM Active Flag

Shows whether the BDM ROM is in the memory map. Cleared by reset.

0 = ROM not in map

1 = ROM in map (MCU is in active BDM)

ENTAG — Instruction Tagging Enable Bit

Shows whether instruction tagging is enabled. Set by the TAGGO instruction and cleared when BDM is entered. Cleared by reset.

0 = Tagging not enabled, or BDM active

1 = Tagging active

NOTE: *Execute a TAGGO command to enable instruction tagging. Do not write ENTAG directly.*

SDV — Shifter Data Valid Bit

Shows that valid data is in the serial interface shift register.

0 = No valid data

1 = Valid Data

NOTE: *SDV is used by firmware-based instructions. Do not attempt to write SDV directly.*

TRACE — Trace Flag

Shows when tracing is enabled.

0 = Tracing not enabled

1 = Tracing active

NOTE: *Execute a TRACE1 command to enable instruction tagging. Do not attempt to write TRACE directly.*

8.7 Instruction Tagging

The instruction queue and cycle-by-cycle CPU activity can be reconstructed in real time or from trace history that was captured by a logic analyzer. However, the reconstructed queue cannot be used to stop the CPU at a specific instruction, because execution has already begun by the time an operation is visible outside the MCU. A separate instruction tagging mechanism is provided for this purpose.

Executing the BDM TAGGO command configures two MCU pins for tagging. The TAGLO signal shares a pin with the LSTRB signal, and the TAGHI signal shares a pin with the BKGD pin. Tagging information is latched on the falling edge of ECLK, as shown in [Figure 8-6](#).

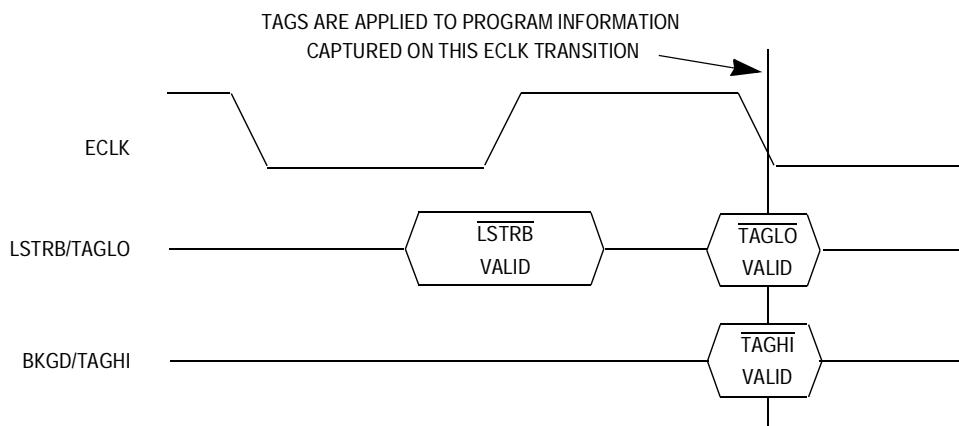


Figure 8-6. Tag Input Timing

Table 8-5 shows the functions of the two tagging pins. The pins operate independently; the state of one pin does not affect the function of the other. The presence of logic level 0 on either pin at the fall of ECLK performs the indicated function. Tagging is allowed in all modes. Tagging is disabled when BDM becomes active.

Table 8-5. Tag Pin Function

| <u>TAGHI</u> | <u>TAGLO</u> | Tag |
|--------------|--------------|------------|
| 1 | 1 | No tag |
| 1 | 0 | Low byte |
| 0 | 1 | High byte |
| 0 | 0 | Both bytes |

In 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 CPU enters active background debug mode rather than executing the instruction.

8.8 Breakpoints

Breakpoints halt instruction execution at particular places in a program. To assure transparent operation, breakpoint control logic is implemented outside the CPU, and particular models of MCU can have different breakpoint capabilities. Refer to the appropriate device manual for detailed information. Generally, breakpoint logic can be configured to halt execution before an instruction executes or to halt execution on the next instruction boundary following the breakpoint.

8.8.1 Breakpoint Type

The three basic types of breakpoints are:

1. Address-only breakpoints that cause the CPU to execute an SWI. These breakpoints can be set only on addresses. When the breakpoint logic encounters the breakpoint tag, the CPU12 executes an SWI instruction.
2. Address-only breakpoints that cause the MCU to enter BDM. These breakpoints can be set only on addresses. When the breakpoint logic encounters the breakpoint tag, BDM is activated.
3. Address/data breakpoints that cause the MCU to enter BDM. These breakpoints can be set on an address, or on an address and data. When the breakpoint logic encounters the breakpoint tag, BDM is activated.

8.8.2 Breakpoint Operation

Breakpoints use these two mechanisms to halt execution:

1. The tag mechanism marks a particular program fetch with a high (even) or low (odd) byte indicator. The tagged byte moves through the instruction queue until a start cycle occurs, then the breakpoint is taken. Breakpoint logic can be configured to force BDM, or to initiate an SWI when the tag is encountered.
2. The force BDM mechanism causes the MCU to enter active BDM at the next instruction start cycle.

CPU12 instructions are used to implement both breakpoint mechanisms. When an SWI tag is encountered, the CPU performs the same sequence of operations as for an SWI. When BDM is forced, the CPU executes a BGND instruction. However, because these operations are not part of the normal flow of instruction execution, the control program must keep track of the actual breakpoint address.

Both SWI and BGND store a return PC value (SWI on the stack and BGND in the CPU12 TMP2 register), but this value is automatically incremented to point to the next instruction after SWI or BGND. To resume execution where a breakpoint occurred, the control program must preserve the breakpoint address rather than use the incremented PC value.

The breakpoint logic generally uses match registers to determine when a break is taken. Registers can be used to match the high and low bytes of addresses for single and dual breakpoints, to match data for single breakpoints, or to do both functions. Use of the registers is generally determined by control bit settings.

Section 9. Fuzzy Logic Support

9.1 Contents

| | | |
|---------|--|-----|
| 9.2 | Introduction | 356 |
| 9.3 | Fuzzy Logic Basics | 357 |
| 9.3.1 | Fuzzification (MEM) | 359 |
| 9.3.2 | Rule Evaluation (REV and REVW) | 361 |
| 9.3.3 | Defuzzification (WAV) | 363 |
| 9.4 | Example Inference Kernel | 364 |
| 9.5 | MEM Instruction Details | 366 |
| 9.5.1 | Membership Function Definitions | 366 |
| 9.5.2 | Abnormal Membership Function Definitions | 368 |
| 9.5.2.1 | Abnormal Membership Function Case 1 | 370 |
| 9.5.2.2 | Abnormal Membership Function Case 2 | 371 |
| 9.5.2.3 | Abnormal Membership Function Case 3 | 371 |
| 9.6 | REV and REVW Instruction Details | 372 |
| 9.6.1 | Unweighted Rule Evaluation (REV) | 372 |
| 9.6.1.1 | Set Up Prior to Executing REV | 372 |
| 9.6.1.2 | Interrupt Details | 374 |
| 9.6.1.3 | Cycle-by-Cycle Details for REV | 374 |
| 9.6.2 | Weighted Rule Evaluation (REVW) | 378 |
| 9.6.2.1 | Set Up Prior to Executing REVW | 378 |
| 9.6.2.2 | Interrupt Details | 380 |
| 9.6.2.3 | Cycle-by-Cycle Details for REVW | 380 |
| 9.7 | WAV Instruction Details | 383 |
| 9.7.1 | Set Up Prior to Executing WAV | 384 |
| 9.7.2 | WAV Interrupt Details | 384 |
| 9.7.3 | Cycle-by-Cycle Details for WAV and wavr | 385 |
| 9.8 | Custom Fuzzy Logic Programming | 388 |
| 9.8.1 | Fuzzification Variations | 388 |
| 9.8.2 | Rule Evaluation Variations | 391 |
| 9.8.3 | Defuzzification Variations | 392 |

9.2 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 15 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.

9.3 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. The first part is a fuzzy inference kernel which is executed periodically to determine system outputs based on current system inputs. The second part of the system is a knowledge base which contains membership functions and rules. **Figure 9-1** is a block diagram of this kind of fuzzy logic system.

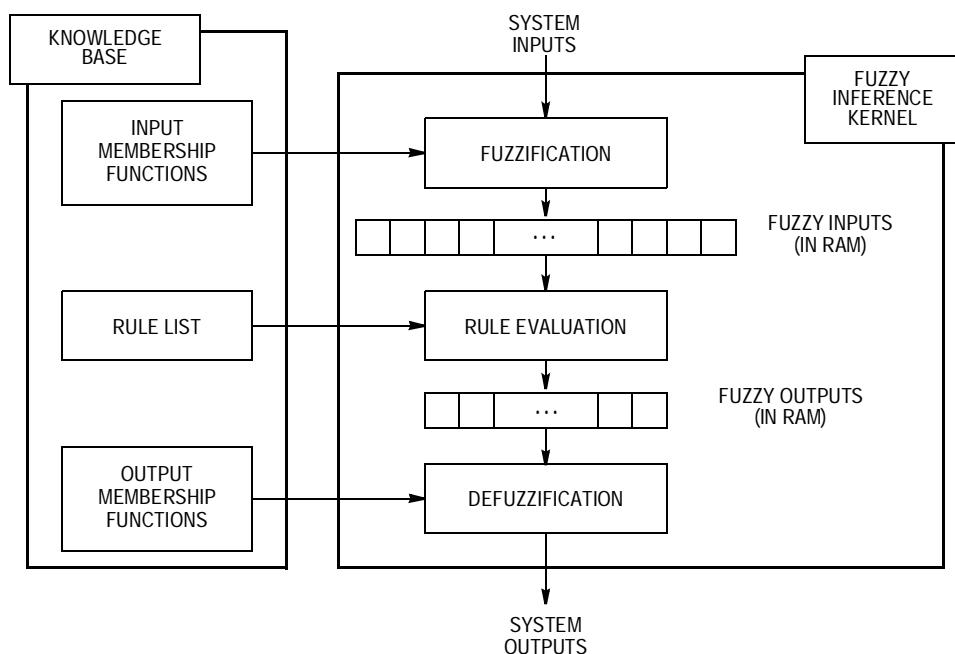


Figure 9-1. Block Diagram of a Fuzzy Logic System

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.

9.3.1 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).

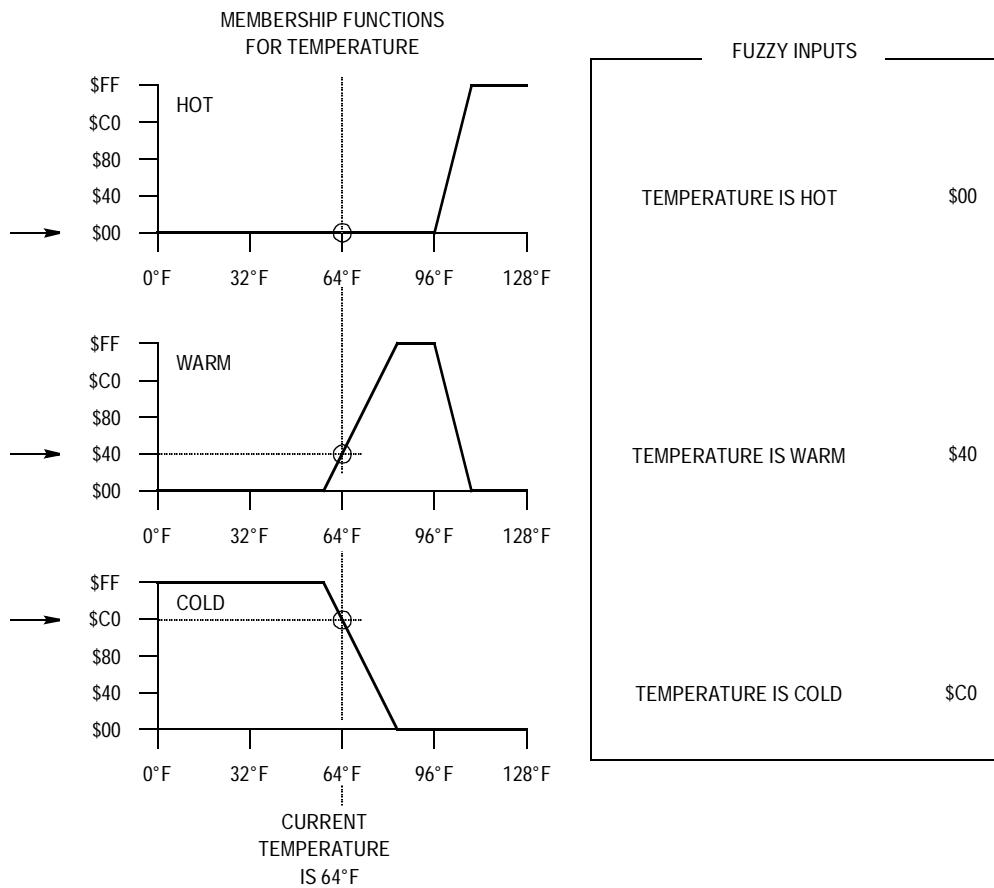


Figure 9-2. Fuzzification Using Membership Functions

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.

9.3.2 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 left portion of the rule is a statement of input conditions and the right portion of the rule is a statement of output actions.

The left 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 right 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.

9.3.3 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.

9.4 Example Inference Kernel

Figure 9-3 is a complete fuzzy inference kernel written in CPU12 assembly language. Numbers in square brackets are cycle counts. 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 54 µs at an 8-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 [1]          LDX    #SGLTN_POS    ;Point at singleton positions
20 [1]          LDAB    #7           ;7 fuzzy outs per COG output
21 [8b+9]        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.

9.5 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.

9.5.1 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.

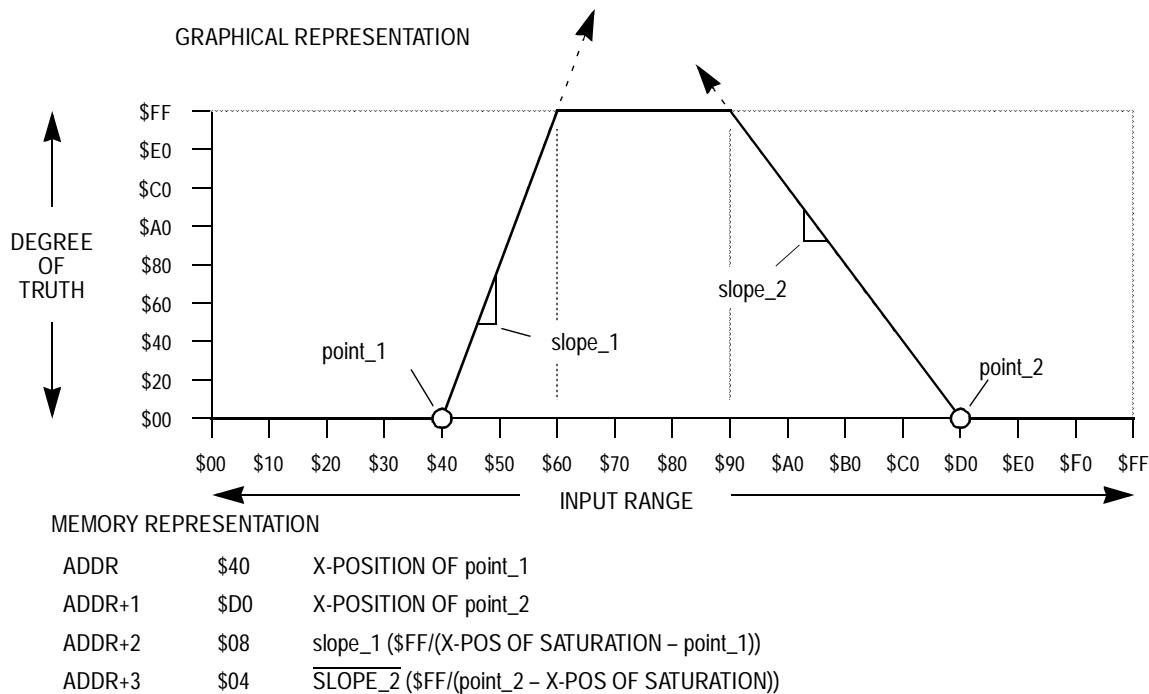


Figure 9-4. Defining a Normal Membership Function

An internal CPU algorithm calculates the y-value where the current input intersects a membership function. This algorithm assumes the membership function obeys some common-sense rules. If the membership function definition is improper, the results may be unusual. See [9.5.2 Abnormal Membership Function Definitions](#) for a discussion of these cases.

These rules apply to normal membership functions.

- $\$00 \leq \text{point1} < \FF
- $\$00 < \text{point2} \leq \FF
- $\text{point1} < \text{point2}$
- 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.

9.5.2 Abnormal Membership Function Definitions

In the CPU12, it is possible (and proper) to define “crisp” membership functions. A crisp membership function has one or both sides vertical (infinite slope). Since the slope value \$00 is not used otherwise, it is assigned to mean infinite slope to the MEM instruction in the CPU12.

Although a good fuzzy development tool will not allow the user to specify an improper membership function, it is possible to have program errors or memory errors which result in erroneous abnormal membership functions. Although these abnormal shapes do not correspond to any working systems, understanding how the CPU12 treats these cases can be helpful for debugging.

A close examination of the MEM instruction algorithm will show how such membership functions are evaluated. [Figure 9-5](#) is a complete flow diagram for the execution of a MEM instruction. Each rectangular box represents one CPU bus cycle. The number in the upper left corner corresponds to the cycle number and the letter corresponds to the cycle type (refer to [Section 6. Instruction Glossary](#) for details). The upper portion of the box includes information about bus activity during this cycle (if any). The lower portion of the box, which is separated by a dashed line, includes information about internal CPU processes. It is common for several internal functions to take place during a single CPU cycle (for example, in cycle 2, two 8-bit subtractions take place and a flag is set based on the results).

Consider 4a: If (((Slope_2 = 0) or (Grade_2 > \$FF)) and (flag_d12n = 0)).

The flag_d12n is zero as long as the input value (in accumulator A) is within the trapezoid. Everywhere outside the trapezoid, one or the other delta term will be negative, and the flag will equal one. Slope_2 equals zero indicates the right side of the trapezoid has infinite slope, so the resulting grade should be \$FF everywhere in the trapezoid, including at point_2, as far as this side is concerned. The term grade_2 greater than \$FF means the value is far enough into the trapezoid that the right sloping side of the trapezoid has crossed above the \$FF cutoff level and the resulting grade should be \$FF as far as the right sloping side is concerned. 4a decides if the value is left of the right sloping side (Grade = \$FF), or on the sloping portion of the right side of the trapezoid (Grade = Grade_2). 4b could still override this tentative value in grade.

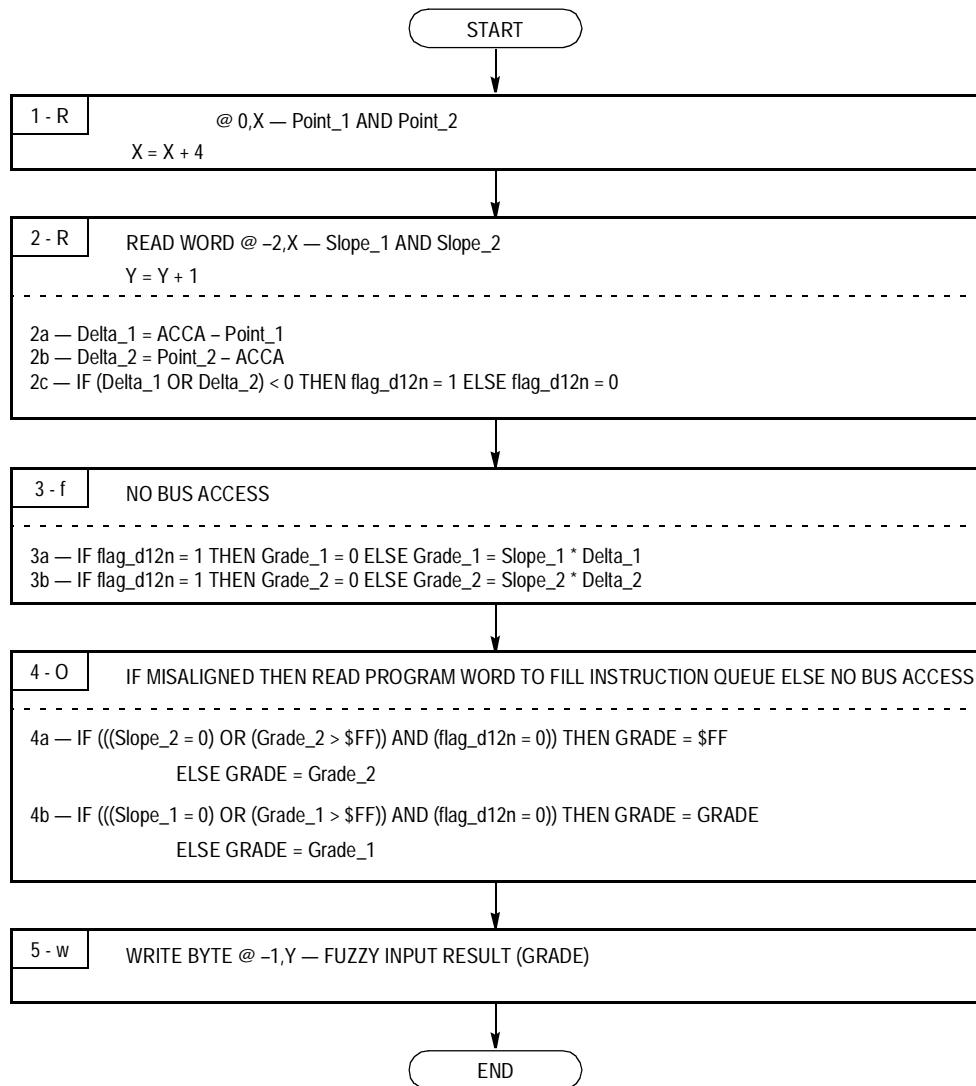


Figure 9-5. MEM Instruction Flow Diagram

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.

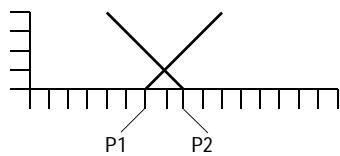
The special cases shown here represent abnormal membership function definitions. The explanations describe how the specific algorithm in the CPU12 resolves these unusual cases. The results are not all intuitively obvious, but rather fall out from the specific algorithm. Remember, these cases should not occur in a normal system.

9.5.2.1 Abnormal Membership Function Case 1

This membership function is abnormal because the sloping sides cross below the \$FF cutoff level. The flag_d12n signal forces the membership function to evaluate to \$00 everywhere except from point_1 to point_2. Within this interval, the tentative values for grade_1 and grade_2 calculated in cycle 3 fall on the crossed sloping sides. In step 4a, grade gets set to the grade_2 value, but in 4b this is overridden by the grade_1 value, which ends up as the result of the MEM instruction. One way to say this is that the result follows the left sloping side until the input passes point_2, where the result goes to \$00.

MEMORY DEFINITION: \$60, \$80, \$04, \$04; point_1, point_2, slope_1, slope_2

GRAPHICAL REPRESENTATION



HOW INTERPRETED

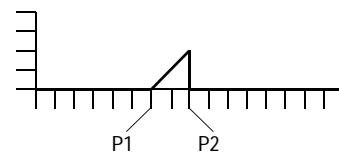


Figure 9-6. Abnormal Membership Function Case 1

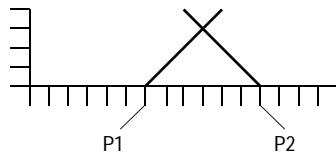
If point_1 was to the right of point_2, flag_d12n would force the result to be \$00 for all input values. In fact, flag_d12n always limits the region of interest to the space greater than or equal to point_1 and less than or equal to point_2.

9.5.2.2 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 ($\text{grade_1} > \$FF$) 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 ($\text{grade} = \text{grade_2}$), and the result follows the right sloping side.

MEMORY DEFINITION: \$60, \$C0, \$04, \$04; point_1, point_2, slope_1, slope_2

GRAPHICAL REPRESENTATION



HOW INTERPRETED

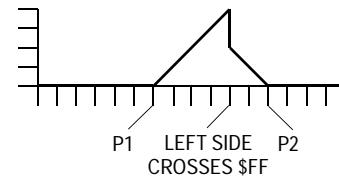


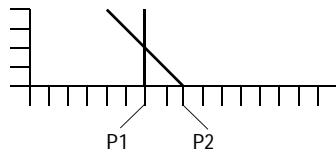
Figure 9-7. Abnormal Membership Function Case 2

9.5.2.3 Abnormal Membership Function Case 3

The membership function in case 3 is abnormal because the sloping sides cross below the \$FF cutoff level, and the left sloping side has infinite slope. In this case, 4a is not true, so grade equals grade_2. 4b is true because slope_1 is zero, so 4b does not overwrite grade.

MEMORY DEFINITION: \$60, \$80, \$00, \$04; point_1, point_2, slope_1, slope_2

GRAPHICAL REPRESENTATION



HOW INTERPRETED

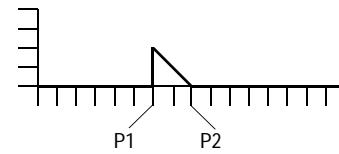


Figure 9-8. Abnormal Membership Function Case 3

9.6 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.

9.6.1 Unweighted Rule Evaluation (REV)

This instruction implements basic min-max rule evaluation. CPU registers are used for pointers and intermediate calculation results.

Since the REV instruction is essentially a list-processing instruction, execution time is dependent on the number of elements in the rule list. The REV instruction is interruptible (typically within three bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and REVW instructions.

9.6.1.1 Set Up Prior to Executing REV

Some CPU registers and memory locations need to be set up prior to executing the REV instruction. X and Y index registers are used as index pointers to the rule list and the fuzzy inputs and outputs. The A accumulator is used for intermediate calculation results and needs to be set to \$FF initially. The V condition code bit is used as an instruction status indicator to show whether antecedents or consequents are being processed. Initially, the V bit is cleared to zero to indicate antecedents are being processed. The fuzzy outputs (working RAM locations) need to be cleared to \$00. If these values are not initialized before executing the REV instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The REV instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the REV instruction finishes, X will point at the next address past the \$FF separator character that marks the end of the rule list.

The Y index register is set to the base address for the fuzzy inputs and outputs (in working RAM). Each rule antecedent is an unsigned 8-bit offset from this base address to the referenced fuzzy input. Each rule consequent is an unsigned 8-bit offset from this base address to the referenced fuzzy output. The Y index register remains constant throughout execution of the REV instruction.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REV instruction. During antecedent processing, A starts out at \$FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent (MIN). During consequent processing, A holds the truth value for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REV, A must be set to \$FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to \$FF when the instruction detects the \$FE marker character between the last consequent of the previous rule and the first antecedent of a new rule.

The instruction LDAA #\$FF clears the V bit at the same time it initializes A to \$FF. This satisfies the REV setup requirement to clear the V bit as well as the requirement to initialize A to \$FF. Once the REV instruction starts, the value in the V bit is automatically maintained as \$FE separator characters are detected.

The final requirement to clear all fuzzy outputs to \$00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value for the current rule. If the current truth value is larger, it is written over the previous value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REV finishes, A will hold the truth value for the last rule in the rule list. The V condition code bit should be one because the last element before the \$FF end marker should have been a rule consequent. If V is zero after executing REV, it indicates the rule list was structured incorrectly.

9.6.1.2 Interrupt Details

The REV instruction includes a 3-cycle processing loop for each byte in the rule list (including antecedents, consequents, and special separator characters). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REV instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REV instruction, points to the REV instruction rather than the instruction that follows. This causes the CPU to try to execute a new REV instruction upon return from the interrupt. Since the CPU registers (including the V bit in the condition codes register) indicate the current status of the interrupted REV instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.6.1.3 Cycle-by-Cycle Details for REV

The central element of the REV instruction is a 3-cycle loop that is executed once for each byte in the rule list. There is a small amount of housekeeping activity to get this loop started as REV begins and a small sequence to end the instruction. If an interrupt comes, there is a special small sequence to save CPU status on the stack before honoring the requested interrupt.

Figure 9-9 is a REV instruction flow diagram. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to execution cycle codes (refer to [Section 6. Instruction Glossary](#) for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit or no data is transferred.

When a value is read from memory, it cannot be used by the CPU until the second cycle after the read takes place. This is due to access and propagation delays.

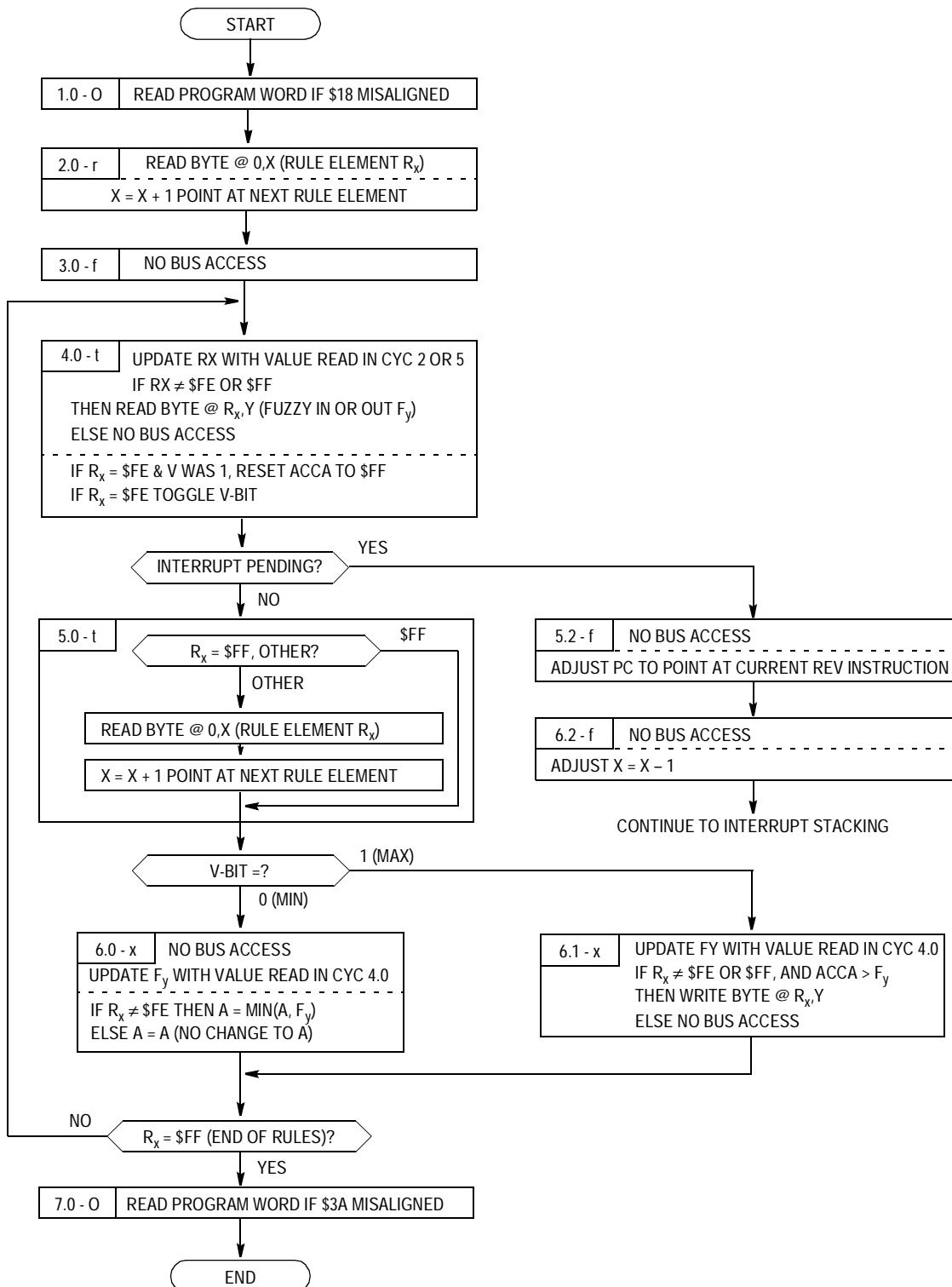


Figure 9-9. REV Instruction Flow Diagram

Since there is more than one flow path through the REV instruction, cycle numbers have a decimal place. This decimal place indicates which of several possible paths is being used. The CPU normally moves forward by one digit at a time within the same flow (flow number is indicated after the decimal point in the cycle number). There are two exceptions possible to this orderly sequence through an instruction. The first is a branch back to an earlier cycle number to form a loop as in 6.0 to 4.0. The second type of sequence change is from one flow to a parallel flow within the same instruction such as 4.0 to 5.2, which occurs if the REV instruction senses an interrupt. In this second type of sequence branch, the whole number advances by one and the flow number changes to a new value (the digit after the decimal point).

In cycle 1.0, the CPU12 does an optional program word access to replace the \$18 prebyte of the REV instruction. Notice that cycle 7.0 is also an O type cycle. One or the other of these will be a program word fetch, while the other will be a free cycle where the CPU does not access the bus. Although the \$18 page prebyte is a required part of the REV instruction, it is treated by the CPU12 as a somewhat separate single cycle instruction.

Rule evaluation begins at cycle 2.0 with a byte read of the first element in the rule list. Usually this would be the first antecedent of the first rule, but the REV instruction can be interrupted, so this could be a read of any byte in the rule list. The X index register is incremented so it points to the next element in the rule list. Cycle 3.0 is needed to satisfy the required delay between a read and when data is valid to the CPU. Some internal CPU housekeeping activity takes place during this cycle, but there is no bus activity. By cycle 4.0, the rule element that was read in cycle 2.0 is available to the CPU.

Cycle 4.0 is the first cycle of the main three cycle rule evaluation loop. Depending upon whether rule antecedents or consequents are being processed, the loop will consist of cycles 4.0, 5.0, 6.0, or the sequence 4.0, 5.0, 6.1. This loop is executed once for every byte in the rule list, including the \$FE separators and the \$FF end-of-rules marker.

At each cycle 4.0, a fuzzy input or fuzzy output is read, except during the loop passes associated with the \$FE and \$FF marker bytes, where no bus access takes place during cycle 4.0. The read access uses the Y index register as the base address and the previously read rule byte (R_x) as an unsigned offset from Y. The fuzzy input or output value read here

will be used during the next cycle 6.0 or 6.1. Besides being used as the offset from Y for this read, the previously read R_x is checked to see if it is a separator character (\$FE). If R_x was \$FE and the V bit was one, this indicates a switch from processing consequents of one rule to starting to process antecedents of the next rule. At this transition, the A accumulator is initialized to \$FF to prepare for the min operation to find the smallest fuzzy input. Also, if Rx is \$FE, the V bit is toggled to indicate the change from antecedents to consequents, or consequents to antecedents.

During cycle 5.0, a new rule byte is read unless this is the last loop pass, and R_x is \$FF (marking the end of the rule list). This new rule byte will not be used until cycle 4.0 of the next pass through the loop.

Between cycle 5.0 and 6.x, the V-bit is used to decide which of two paths to take. If V is zero, antecedents are being processed and the CPU progresses to cycle 6.0. If V is one, consequents are being processed and the CPU goes to cycle 6.1.

During cycle 6.0, the current value in the A accumulator is compared to the fuzzy input that was read in the previous cycle 4.0, and the lower value is placed in the A accumulator (min operation). If Rx is \$FE, this is the transition between rule antecedents and rule consequents, and this min operation is skipped (although the cycle is still used). No bus access takes place during cycle 6.0 but cycle 6.x is considered an x type cycle because it could be a byte write (cycle 6.1) or a free cycle (cycle 6.0 or 6.1 with Rx = \$FE or \$FF).

If an interrupt arrives while the REV instruction is executing, REV can break between cycles 4.0 and 5.0 in an orderly fashion so that the rule evaluation operation can resume after the interrupt has been serviced. Cycles 5.2 and 6.2 are needed to adjust the PC and X index register so the REV operation can recover after the interrupt. PC is adjusted backward in cycle 5.2 so it points to the currently running REV instruction. After the interrupt, rule evaluation will resume, but the values that were stored on the stack for index registers, accumulator A, and CCR will cause the operation to pick up where it left off. In cycle 6.2, the X index register is adjusted backward by one because the last rule byte needs to be re-fetched when the REV instruction resumes.

After cycle 6.2, the REV instruction is finished, and execution would continue to the normal interrupt processing flow.

9.6.2 Weighted Rule Evaluation (REVW)

This instruction implements a weighted variation of min-max rule evaluation. The weighting factors are stored in a table with one 8-bit entry per rule. The weight is used to multiply the truth value of the rule (minimum of all antecedents) by a value from zero to one to get the weighted result. This weighted result is then applied to the consequents, just as it would be for unweighted rule evaluation.

Since the REVW instruction is essentially a list-processing instruction, execution time is dependent on the number of rules and the number of elements in the rule list. The REVW instruction is interruptible (typically within three to five bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and REVW instructions.

The rule structure is different for REVW than for REV. For REVW, the rule list is made up of 16-bit elements rather than 8-bit elements. Each antecedent is represented by the full 16-bit address of the corresponding fuzzy input. Each rule consequent is represented by the full address of the corresponding fuzzy output.

The markers separating antecedents from consequents are the reserved 16-bit value \$FFFE, and the end of the last rule is marked by the reserved 16-bit value \$FFFF. Since \$FFFE and \$FFFF correspond to the addresses of the reset vector, there would never be a fuzzy input or output at either of these locations.

9.6.2.1 Set Up Prior to Executing REVW

Some CPU registers and memory locations need to be set up prior to executing the REVW instruction. X and Y index registers are used as index pointers to the rule list and the list of rule weights. The A accumulator is used for intermediate calculation results and needs to be set to \$FF initially. The V condition code bit is used as an instruction status indicator that shows whether antecedents or consequents are being processed. Initially the V bit is cleared to zero to indicate antecedents are being processed. The C condition code bit is used to indicate whether rule weights are to be used (1) or not (0). The fuzzy outputs (working RAM locations) need to be cleared to \$00. If these

values are not initialized before executing the REVW instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The REVW instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the REVW instruction finishes, X will point at the next address past the \$FFFF separator word that marks the end of the rule list.

The Y index register is set to the starting address of the list of rule weights. Each rule weight is an 8-bit value. The weighted result is the truncated upper eight bits of the 16-bit result, which is derived by multiplying the minimum rule antecedent value (\$00–\$FF) by the weight plus one (\$001–\$100). This method of weighting rules allows an 8-bit weighting factor to represent a value between zero and one inclusive.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REVW instruction. During antecedent processing, A starts out at \$FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent. If rule weights are enabled by the C condition code bit equal one, the rule truth value is multiplied by the rule weight just before consequent processing starts. During consequent processing, A holds the truth value (possibly weighted) for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REVW, A must be set to \$FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to \$FF when the instruction detects the \$FFFE marker word between the last consequent of the previous rule, and the first antecedent of a new rule.

Both the C and V condition code bits must be set up prior to starting a REVW instruction. Once the REVW instruction starts, the C bit remains constant and the value in the V bit is automatically maintained as \$FFFE separator words are detected.

The final requirement to clear all fuzzy outputs to \$00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value (weighted) for the current rule. If the current truth value is larger, it is written over the previous

value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REVW finishes, A will hold the truth value (weighted) for the last rule in the rule list. The V condition code bit should be one because the last element before the \$FFFF end marker should have been a rule consequent. If V is zero after executing REVW, it indicates the rule list was structured incorrectly.

9.6.2.2 Interrupt Details

The REVW instruction includes a 3-cycle processing loop for each word in the rule list (this loop expands to five cycles between antecedents and consequents to allow time for the multiplication with the rule weight). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REVW instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REVW instruction, points to the REVW instruction rather than the instruction that follows. This causes the CPU to try to execute a new REVW instruction upon return from the interrupt. Since the CPU registers (including the C bit and V bit in the condition codes register) indicate the current status of the interrupted REVW instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.6.2.3 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 enabled by the C bit equal one), this loop takes five cycles. There is a small amount of housekeeping activity to get this loop started as REVW begins and a small sequence to end the instruction. If an interrupt comes, there is a special small

sequence to save CPU status on the stack before the interrupt is serviced.

Figure 9-10 is a detailed flow diagram for the REVW instruction. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to the execution cycle codes (refer to [Section 6. Instruction Glossary](#) for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

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.

Fuzzy Logic Support

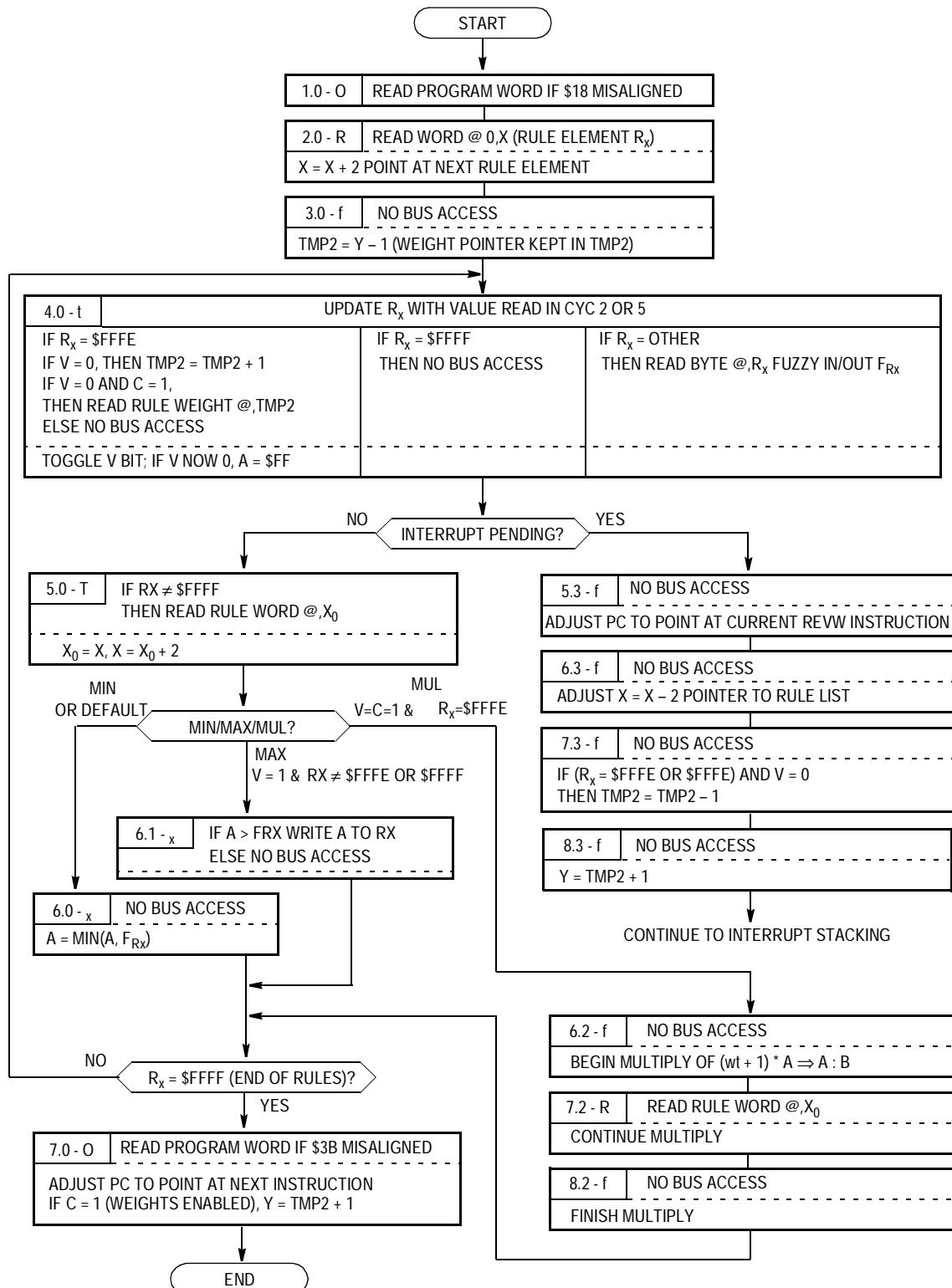


Figure 9-10. REVW Instruction Flow Diagram

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.

9.7 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 [9.7.3 Cycle-by-Cycle Details for WAV and wavr](#) for more detail.

9.7.1 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.

9.7.2 WAV Interrupt Details

The WAV instruction includes an 8-cycle processing loop for each label of the system output. Within this loop, the CPU checks whether a qualified interrupt request is pending. If an interrupt is detected, the current values of the internal temporary registers for the 24-bit and 16-bit sums are stacked, the CPU registers are stacked, and the interrupt is serviced.

A special processing sequence is executed when an interrupt is detected during a weighted average calculation. This exit sequence adjusts the PC so that it points to the second byte of the WAV object code (\$3C), before the PC is stacked. Upon return from the interrupt, the \$3C value is interpreted as a wavr pseudo-instruction. The wavr pseudo-instruction causes the CPU to execute a special WAV resumption sequence. The wavr recovery sequence adjusts the PC so that it looks like it did during execution of the original WAV instruction, then jumps back into the WAV processing loop. If another interrupt occurs before the weighted average calculation finishes, the PC is adjusted again as it was for the first interrupt. WAV can be interrupted any number of times, and additional WAV instructions can be executed while a WAV instruction is interrupted.

9.7.3 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 9-11 is a flow diagram of the WAV instruction including the wavr pseudo-instruction. Each rectangular box in this figure represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of the boxes correspond to execution cycle codes (refer to [Section 6. Instruction Glossary](#) for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

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.

There is no bus access in cycles 2.0 or 3.0. In cycle 3.0, three internal 16-bit temporary registers are cleared in preparation for summation operations. The WAV instruction maintains a 32-bit sum-of-products in TMP3 : TMP2 and a 16-bit sum-of-weights in TMP1. By keeping these sums inside the CPU, bus accesses are reduced and the WAV operation is optimized for high speed.

Cycles 4.0 through 11.0 form the 8-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 4.0, and the test for zero is located at the bottom of the loop after cycle 11.0. Cycle 5.0 and 6.0 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 7.0 is used to accumulate the current fuzzy output into TMP1. Cycles 8.0 through 10.0 are used to perform the eight by eight multiply of F_i times S_i . The multiply result is accumulated into TMP3 : TMP2 during cycles 10.0 and 11.0. Even though the sum-of-products will not exceed 24 bits,

Fuzzy Logic Support

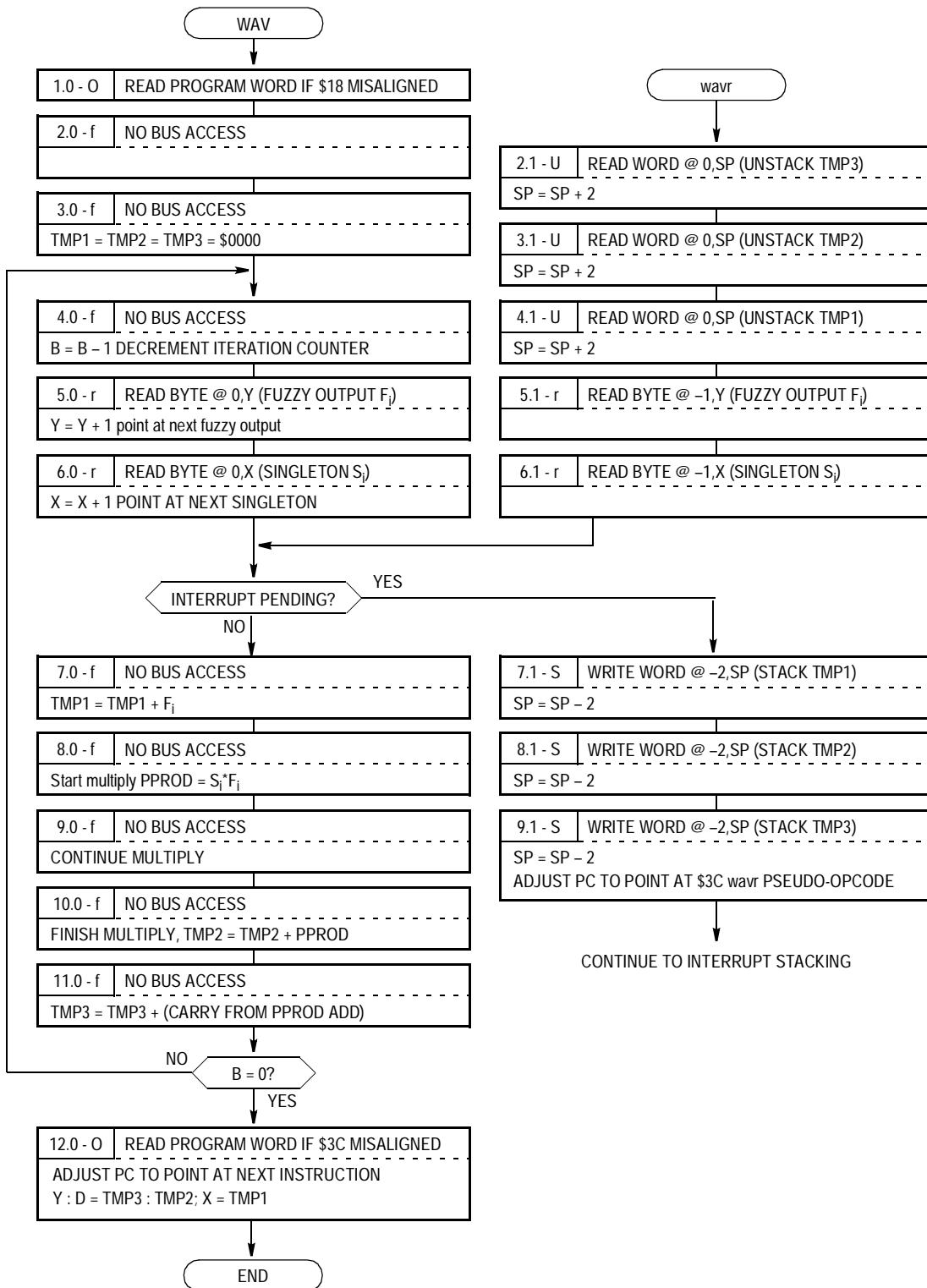


Figure 9-11. WAV and wavr Instruction Flow Diagram

the sum is maintained in the 32-bit combined TMP3 : 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 7.0, through cycle 11.0, to the top of the loop to cycle 4.0, through cycle 6.0 to the interrupt check. There is also a three cycle (7.1 through 9.1) exit sequence making this latency path a total of 12 cycles. There is an even longer path, but it is much less likely to occur. If an interrupt comes near the beginning of cycle 2.1, when a weighted average operation is being resumed after a previous interrupt, the latency path is 2.1 through 6.1 plus 7.0 through 11.0 plus 4.0 through 6.0 plus the exit 7.1 through 9.1. This is a worst-case total of 17 cycles.

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 (2.1 through 4.1), and reads to get the operands for the current iteration. The normal WAV flow is then rejoined at cycle 7.0.

Upon normal completion of the instruction (cycle 12.0), the PC is adjusted so it points to the next instruction. The results are transferred from the TMP registers into CPU registers in such a way that the EDIV instruction can be used to divide the sum-of-products by the sum-of-weights. TMP3 : TMP2 is transferred into Y : D and TMP1 is transferred into X.

9.8 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.

9.8.1 Fuzzification Variations

The MEM instruction supports trapezoidal membership functions and several other varieties, including membership functions with vertical sides (infinite slope sides). Triangular membership functions are a subset of trapezoidal functions. Some practitioners refer to s-, z-, and π -shaped membership functions. These refer to a trapezoid butted against the right end of the x-axis, a trapezoid butted against the left end of the x-axis, and a trapezoidal membership function that isn't butted against either end of the x-axis, respectively. Many other membership function shapes are possible, if memory space and processing bandwidth are sufficient.

Tabular membership functions offer complete flexibility in shape and very fast evaluation time. However, tables take a very large amount of memory space (as many as 256 bytes per label of one system input). The excessive size to specify tabular membership functions makes them impractical for most microcontroller-based fuzzy systems. The CPU12 instruction set includes two instructions (TBL and ETBL) for lookup and interpolation of compressed tables.

The TBL instruction uses 8-bit table entries (y-values) and returns an 8-bit result. The ETBL instruction uses 16-bit table entries (y-values) and returns a 16-bit result. A flexible indexed addressing mode is used to identify the effective address of the data point at the beginning of the line segment, and the data value for the end point of the line segment is the next consecutive memory location (byte for TBL and word for ETBL). In both cases, the B accumulator represents the ratio of (the x-distance from the beginning of the line segment to the lookup point) to (the

x-distance from the beginning of the line segment to the end of the line segment). B is treated as an 8-bit binary fraction with radix point left of the MSB, so each line segment can effectively be divided into 256 pieces. During execution of the TBL or ETBL instruction, the difference between the end point y-value and the beginning point y-value (a signed byte-TBL or word-ETBL) is multiplied by the B accumulator to get an intermediate delta-y term. The result is the y-value of the beginning point, plus this signed intermediate delta-y value.

Because indexed addressing mode is used to identify the starting point of the line segment of interest, there is a great deal of flexibility in constructing tables. A common method is to break the x-axis range into 256 equal width segments and store the y value for each of the resulting 257 endpoints. The 16-bit D accumulator is then used as the x input to the table. The upper eight bits (A) is used as a coarse lookup to find the line segment of interest, and the lower eight bits (B) is used to interpolate within this line segment.

In the program sequence

```
LDX      #TBL_START
LDD      DATA_IN
TBL      A,X
```

The notation A,X causes the TBL instruction to use the Ath line segment in the table. The low-order half of D (B) is used by TBL to calculate the exact data value from this line segment. This type of table uses only 257 entries to approximate a table with 16 bits of resolution. This type of table has the disadvantage of equal width line segments, which means just as many points are needed to describe a flat portion of the desired function as are needed for the most active portions.

Another type of table stores x:y coordinate pairs for the endpoints of each linear segment. This type of table may reduce the table storage space compared to the previous fixed-width segments because flat areas of the functions can be specified with a single pair of endpoints. This type of table is a little harder to use with the CPU12 TBL and ETBL instructions because the table instructions expect y-values for segment endpoints to be in consecutive memory locations.

Consider a table made up of an arbitrary number of x:y coordinate pairs, where all values are eight bits. The table is entered with the x-coordinate of the desired point to lookup in the A accumulator. When the table is exited, the corresponding y-value is in the A accumulator. [Figure 9-12](#) 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)
                      ;loop if XL .le. Xn
BLS       FIND_LOOP
* 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
LDAB    0,Y           ;zero upper half of D
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
EXG     D,X           ;X reg = (XL-XB)/(XE-XB)
EXG     A,B           ;move fractional result to A:B
TSTA
BPL     NO_ROUND
NO_ROUND INCB
LDAA    1,Y           ;round B up by 1
PSHA
LDAA    -1,Y          ;YE
PSHA
TBL     2,SP+          ;put on stack for TBL later
                      ;now YB@0,SP and YE@1,SP
                      ;interpolate and deallocate
                      ;stack temps

```

Figure 9-12. 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 like “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 teen-ager 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.

9.8.2 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

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.

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.

9.8.3 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 8-MHz cycles for unsigned or signed variations. A 16-bit by 16-bit multiply with a 32-bit result takes only three 8-MHz cycles. The EMACS instruction uses 16-bit operands and accumulates the result in a 32-bit memory location, taking only 12 8-MHz cycles per iteration, including accessing all operands from memory and storing the result to memory.

Fuzzy Logic Support

Section 10. Memory Expansion

10.1 Contents

| | | |
|----------|---|-----|
| 10.2 | Introduction | 396 |
| 10.3 | Expansion System Description | 396 |
| 10.4 | CALL and Return from Call Instructions | 398 |
| 10.5 | Address Lines for Expansion Memory | 401 |
| 10.6 | Overlay Window Controls | 401 |
| 10.7 | Using Chip-Select Circuits | 402 |
| 10.7.1 | Program Memory Expansion Chip-Select Controls | 403 |
| 10.7.1.1 | CSP1E Control Bit | 403 |
| 10.7.1.2 | CSP0E Control Bit | 403 |
| 10.7.1.3 | CSP1FL Control Bit | 403 |
| 10.7.1.4 | CSPA21 Control Bit | 403 |
| 10.7.1.5 | STRP0A:STRP0B Control Field | 403 |
| 10.7.1.6 | STRP1A:STRP1B Control Field | 404 |
| 10.7.2 | Data Expansion Chip Select Controls | 404 |
| 10.7.2.1 | CSDE Control Bit | 404 |
| 10.7.2.2 | CSDHF Control Bit | 404 |
| 10.7.2.3 | STRDA:STRDB Control Field | 405 |
| 10.7.3 | Extra Expansion Chip Select Controls | 405 |
| 10.7.3.1 | CSEE Control Bit | 405 |
| 10.7.3.2 | CSEEP Control Bit | 405 |
| 10.7.3.3 | STREA:STREB Control Field | 405 |
| 10.8 | System Notes | 405 |

10.2 Introduction

This section discusses expansion memory principles that apply to the entire M68HC12 Family. Some family devices do not have memory expansion capabilities, and the size of the expanded memory can also vary. Refer to the documentation for a derivative to determine details of implementation.

10.3 Expansion System Description

Certain members of the M68HC12 Family 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 paged memory or bank-switching scheme.

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

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

The M68HC12 system requires no external glue logic. Bank switching overhead is reduced by implementing control logic in the microcontroller unit (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. Operation of the bank-switching logic is transparent to the CPU.

The CPU12 has a linear 64-Kbyte address space. All MCU system resources, including control registers for on-chip peripherals and on-chip memory arrays, are mapped into this space. In a typical M68HC12 derivative, the resources have default addresses out of reset, but can be re-mapped to other addresses by means of control registers in the on-chip integration module.

Memory expansion control logic is outside the CPU. A block of circuitry in the MCU integration module manages overlays that occupy pre-defined locations in the 64-Kbyte space addressed by the CPU. These overlays can be thought of as windows through which the CPU accesses information in the expanded memory space.

There are three overlay windows. The program window expands program memory, the data window is used for independent data expansion, and the extra window expands access to special types of memory such as electrically-erasable, programmable read-only memory (EEPROM). The program window always occupies the 16-Kbyte space from \$8000 to \$BFFF. Data and extra windows can vary in size and location.

Each window has an associated page select register that selects external memory pages to be accessed via the window. Only one page at a time can occupy a window; the value in the register must be changed to access a different page of memory. With 8-bit registers, there can be up to 256 expansion pages per window, each page the same size as the window.

For data and extra windows, page switching is accomplished by means of normal read and write instructions. This is the traditional method of managing a bank-switching system. The CPU12 call subroutine in expanded memory (CALL) and return-from-call (RTC) instructions automatically manipulate the program page select (PPAGE) register for the program window.

In M68HC12 expanded memory systems, control registers, vector spaces, and a portion of on-chip memory are located in unpaged portions of the 64-Kbyte address space. The stack and I/O addresses should also be placed in unpaged memory to make them accessible from any overlay page.

The initial portions of exception handlers must be located in unpaged memory because the 16-bit exception vectors cannot point to addresses in paged memory. However, service routines can call other routines in paged memory. The upper 16-Kbyte block of memory space (\$C000–\$FFFF) is unpaged. It is recommended that all reset and interrupt vectors point to locations in this area.

Although internal MCU resources, such as control registers and on-chip memory, have default addresses out of reset, each can typically be relocated by changing the default values in control registers. Normally, input/output (I/O) addresses, control registers, vector spaces, overlay windows, and on-chip memory are not mapped so that their respective address ranges overlap. However, there is an access priority order that prevents access conflicts should such overlaps occur. **Table 10-1** shows the mapping precedence. Resources with higher precedence block access to those with a lower precedence. The windows have lowest priority — registers, exception vectors, and on-chip memory are always visible to a program regardless of the values in the page select registers.

Table 10-1. Mapping Precedence

| Precedence | Resource |
|------------|---------------------------|
| 1 | Registers |
| 2 | Exception Vectors/BDM ROM |
| 3 | RAM |
| 4 | EEPROM |
| 5 | FLASH |
| 6 | Expansion Windows |

When background debugging is enabled and active, the CPU executes code located in a small on-chip ROM mapped to addresses \$FF20 to \$FFFF, and BDM control registers are accessible at addresses \$FF00 to \$FF06. The BDM ROM replaces the regular system vectors while BDM is active, but BDM resources are not in the memory map during normal execution of application programs.

10.4 CALL and Return from Call Instructions

The CALL is similar to a jump-to-subroutine (JSR) instruction, but the subroutine that is called can be located anywhere in the normal 64-Kbyte address space or on any page of program expansion memory. When CALL is executed, a return address is calculated, then it and the current program page register value are stacked, and a new instruction-supplied value is written to PPAGE. The PPAGE value controls which of the 256 possible pages is visible through the 16-Kbyte window in the 64-Kbyte

memory map. Execution continues at the address of the called subroutine.

The actual sequence of operations that occur during execution of CALL is:

- The CPU reads the old PPAGE value into an internal temporary register and writes the new instruction-supplied PPAGE value to PPAGE. This switches the destination page into the program overlay window.
- The CPU calculates the address of the next instruction after the CALL instruction (the return address) and pushes this 16-bit value onto the stack.
- The old 8-bit PPAGE value is pushed onto the stack.
- The effective address of the subroutine is calculated, the queue is refilled, and execution begins at the new address.

This sequence of operations is an uninterruptable CPU instruction. There is no need to inhibit interrupts during CALL execution. In addition, a CALL can be performed from any address in memory to any other address. This is a big improvement over other bank-switching schemes, where the page switch operation can be performed only by a program outside the overlay window.

For all practical purposes, the PPAGE value supplied by the instruction can be considered to be 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 new page value and the address within the page allows use of run-time calculated values rather than immediate values that must be known at the time of assembly.

The RTC instruction is used to terminate subroutines invoked by a CALL instruction. RTC unstacks the PPAGE value and the return address, the queue is refilled, and execution resumes with the next instruction after the corresponding CALL.

The actual sequence of operations that occur during execution of RTC is:

- The return value of the 8-bit PPAGE register is pulled from the stack.
- The 16-bit return address is pulled from the stack and loaded into the PC.
- The return PPAGE value is written to the PPAGE register.
- The queue is refilled and execution begins at the new address.

Since the return operation is implemented as a single uninterruptable CPU instruction, the RTC can be executed from anywhere in memory, including from a different page of extended memory in the overlay window.

In an MCU where there is no memory expansion, the CALL and RTC instructions still perform the same sequence of operations, but there is no PPAGE register or address translation logic. The value the CPU reads when the PPAGE register is accessed is indeterminate but doesn't matter, because the value is not involved in addressing memory in the unpaged 64-Kbyte memory map. When the CPU writes to the non-existent PPAGE register, nothing happens.

The CALL and RTC instructions behave like JSR and RTS, except they have slightly longer execution times. Since extra execution cycles are required, routinely substituting CALL/RTC for JSR/RTS is not recommended. JSR and RTS can be used to access subroutines that are located on the same memory page. However, if a subroutine can be called from other pages, it must be terminated with an RTC. In this case, since RTC unstacks the PPAGE value as well as the return address, all accesses to the subroutine, even those made from the same page, must use CALL instructions.

10.5 Address Lines for Expansion Memory

All M68HC12 Family members have at least 16 address lines, ADDR15–ADDR0. Devices with memory expansion capability can have as many as six additional high-order external address lines, ADDR21–ADDR16. Each of these additional address lines is typically associated with a control bit that allows address expansion to be selectively enabled. When expansion is enabled, internal address translation circuitry multiplexes data from the page select registers onto the high order address lines when there is an access to an address in a corresponding expansion window.

Assume that a device has six expansion address lines and an 8-bit PPAGE register. The lines and the program expansion window have been enabled. The address \$9000 is within the 16-Kbyte program overlay window. When there is an access to this address, the value in the PPAGE register is multiplexed onto external address lines ADDR21–ADDR14. The 14 low-order address lines select a location within the program overlay page. Up to 256 16-Kbyte pages (4 Mbytes) of memory can be accessed through the window. When there is an access to a location that is not within any enabled overlay window, ADDR21–ADDR16 are driven to logic level 1.

The address translation logic can produce the same address on the external address lines for two different internal addresses. For example, the 22-bit address \$3FFFF could result from an internal access to \$FFFF in the 64-Kbyte memory map or to the last location (\$BFFF) within page 255 (PPAGE = \$FF) of the program overlay window. Considering only the 22 external address lines, the last physical page of the program overlay appears to occupy the same address space as the unpaged 16-Kbyte block from \$C000 to \$FFFF of the 64-Kbyte memory map. Using MCU chip-select circuits to enable external memory can resolve these ambiguities.

10.6 Overlay Window Controls

A page select register is associated with each overlay window. PPAGE holds the page select for the program overlay, DPAGE holds the page select for the data overlay, and EPAGE holds the page select for the extra page. The CPU12 manipulates the PPAGE register directly, so it

will always be eight bits or less in devices that support program memory expansion. The DPAGE and EPAGE registers are not controlled by dedicated CPU12 instructions. These registers can be larger or smaller than eight bits in various M68HC12 derivatives.

Typically, each of the overlay windows also has an associated control bit to enable memory expansion through the appropriate window. Memory expansion is generally disabled out of reset, so control bits must be written to enable the address translation logic.

10.7 Using Chip-Select Circuits

M68HC12 chip-select circuits can be used to preclude ambiguities in memory-mapping due to the operation of internal address translation logic. If built-in chip selects are not used, take care to use only overlay pages which produce unique addresses on the external address lines.

M68HC12 derivatives typically have two or more chip-select circuits. Chip-select function is conceptually simple. Whenever an access to a pre-defined range of addresses is made, internal MCU circuitry detects an address match and asserts a control signal that can be used to enable external devices. Chip-select circuits typically incorporate a number of options that make it possible to use more than one range of addresses for matches as well as to enable various types and configurations of external devices.

Chip-select circuits used in conjunction with the memory-expansion scheme must be able to match all accesses made to addresses within the appropriate program overlay window. In the case of the program expansion window, the range of addresses occupies the 16-Kbyte space from \$8000 to \$BFFF. For data and extra expansion windows, the range of addresses varies from device to device. The following paragraphs discuss a typical implementation of memory expansion chip-select functions in the system integration module. Implementation will vary from device to device within the M68HC12 Family. Refer to the appropriate device manual for details.

10.7.1 Program Memory Expansion Chip-Select Controls

There are two program memory expansion chip-select circuits are CSP0 and CSP1. The associated control register contains eight control bits that provide for a number of system configurations.

10.7.1.1 *CSP1E Control Bit*

Enables (1) or disables (0) the CSP1 chip select. The default is disabled.

10.7.1.2 *CSP0E Control Bit*

Enables (1) or disables (0) the CSP0 chip select. The default is enabled. This allows CSP0 to be used to select an external memory that includes the reset vector and startup initialization programs.

10.7.1.3 *CSP1FL Control Bit*

Configures CSP1 to occupy all of the 64-Kbyte memory map that is not used by a higher-priority resource. If CSP1FL = 0, CSP1 is mapped to the area from \$8000 to \$FFFF. CSP1 has the lowest access priority except for external memory space that is not associated with any chip select.

10.7.1.4 *CSPA21 Control Bit*

Logic 1 causes CSP0 and CSP1 to be controlled by the ADDR21 signal. CSP1 is active when ADDR21 = 0, and CSP0 is active when ADDR21 = 1. When CSPA21 is 1, the CSP1FL bit is ignored and both CSP0 and CSP1 are active in the region \$8000–\$FFFF. When CSPA21 is 0, CSP0 and CSP1 operate independently from the value of the ADDR21 signal.

10.7.1.5 *STRP0A:STRP0B Control Field*

These two bits program an extra delay into accesses to the CSP0 area of memory. The choices are 0, 1, 2, or 3 E-cycles in addition to the normal one cycle for unstretched accesses. This allows use of slow external memory without slowing down the entire system.

10.7.1.6 STRP1A:STRP1B Control Field

These two bits program an extra delay into accesses to the CSP1 area of memory. The choices are 0, 1, 2, or 3 E-cycles in addition to the normal one cycle for unstretched accesses. This allows use of slow external memory without slowing down the entire system.

When enabled, CSP0 is active for the memory space from \$8000 through \$FFFF. This includes the program overlay space (\$8000–\$BFFF) and the unpaged 16-Kbyte block from \$C000 through \$FFFF. This configuration can be used if there is a single program memory device (up to four Mbytes) in the system.

If CSP1 is also enabled and the CSPA21 bit is set, CSP1 can be used to select the first 128 16-Kbyte pages (two Mbytes) in the program overlay expansion memory space while CSP0 selects the higher numbered program expansion pages and the unpaged block from \$C000 through \$FFFF. Recall that the external memory device cannot distinguish between an access to the \$C000 to \$FFFF space and an access to \$8000–\$BFFF in the 255th page (PPAGE = \$FF) of the program overlay window.

10.7.2 Data Expansion Chip Select Controls

The data chip select (CSD) has four associated control bits.

10.7.2.1 CSDE Control Bit

Enables (1) or disables (0) the CSD chip select. The default is disabled.

10.7.2.2 CSDHF Control Bit

Configures CSD to occupy the lower half of the 64-Kbyte memory map (for areas that are not used by a higher priority resource). If CSDHF is 0, CSD occupies the range of addresses used by the data expansion window.

10.7.2.3 STRDA:STRDB Control Field

These two bits program an extra delay into accesses to the CSD area of memory. The choices are 0, 1, 2, or 3 additional E-cycles in addition to the normal one cycle for unstretched accesses. This allows use of slow external memory without slowing down the entire system.

10.7.3 Extra Expansion Chip Select Controls

The extra chip select (CSE) has four associated control bits.

10.7.3.1 CSEE Control Bit

Enables (1) or disables (0) the CSE chip select. The default is disabled.

10.7.3.2 CSEEP Control Bit

Logic 1 configures CSE to be active for the EPAGE area. A logic 0 causes CSE to be active for the CS3 area of the internal register space, which can typically be remapped to any 2-Kbyte boundary.

10.7.3.3 STREA:STREB Control Field

These two bits program an extra delay into accesses to the CSE area of memory. The choices are 0, 1, 2, or 3 E-cycles in addition to the normal one cycle for unstretched accesses. This allows use of slow external memory without slowing down the entire system.

To use CSE with the extra overlay window, it must be enabled (CSEE = 1) and configured to follow the extra page (CSEEP = 1).

10.8 System Notes

The expansion overlay windows are specialized for specific application uses, but there are no restrictions on the use of these memory spaces. Motorola MCUs have a memory-mapped architecture in which all memory resources are treated equally. Although it is possible to execute programs in paged external memory in the data and extra overlay areas, it is less convenient than using the program overlay area.

Memory Expansion

The CALL and RTC instructions automate the program page switching functions in an uninterruptable instruction. For the data and extra overlay windows, the user must take care not to let interrupts corrupt the page switching sequence or change the active page while executing out of another page in the same overlay area.

Internal MCU chip-select circuits have access to all 16 internal CPU address lines and the overlay window select lines. This allows all 256 expansion pages in an overlay window to be distinguished from unpaged memory locations with 22-bit addresses that are the same as addresses in overlay pages.

Appendix A. Instruction Reference

A.1 Contents

| | | |
|---------|--|-----|
| A.2 | Introduction | 407 |
| A.3 | Instruction Set Summary | 408 |
| A.3.1 | Notation Used in Instruction Set Summary | 408 |
| A.3.1.1 | Explanation of Italic Expressions in Source Form Column | 408 |
| A.3.1.2 | Address Modes | 409 |
| A.3.1.3 | Machine Coding | 409 |
| A.3.1.4 | Access Detail | 410 |
| A.3.1.5 | Condition Codes Columns | 410 |
| A.4 | Indexed Addressing Postbyte Encoding | 411 |
| A.5 | Transfer and Exchange Postbyte Encoding | 411 |
| A.6 | Loop Primitive Postbyte Encoding | 411 |
| A.7 | Opcode Map | 412 |
| A.8 | Memory Expansion | 432 |
| A.9 | Hexadecimal to ASCII Conversion | 438 |
| A.10 | Hexadecimal to Decimal Conversion | 439 |
| A.11 | Decimal to Hexadecimal Conversion | 439 |

A.2 Introduction

This appendix provides quick references for the instruction set, opcode map, and encoding.

A.3 Instruction Set Summary

Table A-1 is a quick reference to the CPU12 instruction set. The table shows source form, describes the operation performed, lists the addressing modes used, gives machine encoding in hexadecimal form, and describes the effect of execution on the condition code bits.

A.3.1 Notation Used in Instruction Set Summary

A.3.1.1 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 value in the range -16 . . . +15
oprX9 — Any value in the range -256 . . . +255
oprX16 — Any value 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 -256 to +255 locations
rel9 — Label of branch destination within -512 to +511 locations
rel16 — Any label within 64-Kbyte memory space
trapnum — Any 8-bit value in the range \$30-\$39 or \$40-\$FF
xys — X or Y or SP
xysp — X or Y or SP or PC

A.3.1.2 Address Modes

| | |
|----------|---|
| IMM | — Immediate |
| IDX | — Indexed (no extension bytes) includes: 5-bit constant offset Pre/post increment/decrement by 1 . . . 8 Accumulator A, B, or D offset |
| IDX1 | — 9-bit signed offset (one extension byte) |
| IDX2 | — 16-bit signed offset (two extension bytes) |
| [D, IDX] | — Indexed indirect (accumulator D offset) |
| [IDX2] | — Indexed indirect (16-bit offset) |
| INH | — Inherent (no operands in object code) |
| REL | — Two's complement relative offset (branches) |

A.3.1.3 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-4 . |
| 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-5 . |
| 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-2 and Table A-3 . |

A.3.1.4 Access Detail

Each code letter equals one CPU cycle. Uppercase = 16-bit operation and lowercase = 8-bit operation.

- f — Free cycle, CPU doesn't use bus
- g — Read PPAGE internally
- I — Read indirect pointer (indexed indirect)
- i — Read indirect PPAGE value (call indirect)
- n — Write PPAGE internally
- o — Optional program word fetch (P) if instruction is misaligned and has an odd number of bytes of object code — otherwise, appears as a free cycle (f)
- 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
- t — 8-bit conditional read (or free cycle)
- T — 16-bit conditional read (or free cycle)
- x — 8-bit conditional write

Special Cases

PPP/P — Short branch, PPP if branch taken, P if not

OPPP/OPO — Long branch, OPPP if branch taken, OPO if not

A.3.1.5 Condition Codes Columns

- — Status bit is not affected by operation.
- 0 — Status bit is cleared by operation.
- 1 — Status bit is set by operation.
- Δ — Status bit is 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 is used for a special purpose.

A.4 Indexed Addressing Postbyte Encoding

Table A-2 shows postbyte encoding for indexed addressing modes. The mnemonic for the indexed addressing mode postbyte is xb. This is also the notation used in instruction glossary entries. **Table A-3** presents the same information in 2-digit hexadecimal format. The first digit of the postbyte is represented by the value of the columns in the table. The second digit of the postbyte is represented by the value of the row.

A.5 Transfer and Exchange Postbyte Encoding

Table A-4 shows postbyte encoding for transfer and exchange instructions. The mnemonic for the transfer and exchange postbyte is eb. This is also the notation used in instruction glossary entries. The first digit of the instruction postbyte is related to the columns of the table. The second digit of the postbyte is related to the rows. The body of the table shows actions caused by the postbyte.

A.6 Loop Primitive Postbyte Encoding

Table A-5 shows postbyte encoding for loop primitive instructions. The mnemonic for the loop primitive postbyte is lb. This is also the notation used in instruction glossary entries. The loop primitive instructions are:

- DBEQ (decrement and branch if equal to 0)
- DBNE (decrement and branch if not equal to 0)
- IBEQ (increment and branch if equal to 0)
- IBNE (increment and branch if not equal to 0)
- TBEQ (test and branch if equal to 0)
- TBNE (test and branch if not equal to 0)

The first digit of the instruction postbyte corresponds to the columns of the table. The second digit of the postbyte corresponds to the rows. The body of the table shows actions caused by the postbyte.

A.7 Opcode Map

Table A-7 displays the mnemonic, opcode, addressing mode, and cycle count for each instruction. The first table represents those opcodes with no prebyte. The second page of the table represents those opcodes with a prebyte value of \$18. Notice that the first hexadecimal digit of the opcode (shown in the upper left corner of each cell) corresponds to column location, while the second hexadecimal digit of the opcode corresponds to row location.

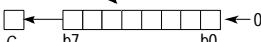
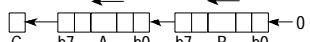
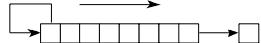
Table A-1. Instruction Set Summary (Sheet 1 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|---|---|--|---|---|----------|---|----------|----------|----------|----------|
| ABA | (A) + (B) \Rightarrow A Add Accumulators A and B | INH | 18 06 | OO | - | - | Δ | - | Δ | Δ | Δ | Δ |
| ABX | (B) + (X) \Rightarrow X <i>Translates to LEAX B,X</i> | IDX | 1A E5 | PP ¹ | - | - | - | - | - | - | - | - |
| ABY | (B) + (Y) \Rightarrow Y <i>Translates to LEAY B,Y</i> | IDX | 19 ED | PP ¹ | - | - | - | - | - | - | - | - |
| ADCA #opr8i ADCA opr8a ADCA opr16a ADCA oprx0_xySp ADCA oprx9_xySp ADCA oprx16_xySp ADCA [D,xySp] ADCA [oprX16,xySp] | (A) + (M) + C \Rightarrow A Add with Carry to A | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 89 ii 99 dd B9 hh ll A9 xb A9 xb ff A9 xb ee ff A9 xb A9 xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | Δ | - | Δ | Δ | Δ | Δ |
| ADCB #opr8i ADCB opr8a ADCB opr16a ADCB oprx0_xySp ADCB oprx9_xySp ADCB oprx16_xySp ADCB [D,xySp] ADCB [oprX16,xySp] | (B) + (M) + C \Rightarrow B Add with Carry to B | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C9 ii D9 dd F9 hh ll E9 xb E9 xb ff E9 xb ee ff E9 xb E9 xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | Δ | - | Δ | Δ | Δ | Δ |
| ADDA #opr8i ADDA opr8a ADDA opr16a ADDA oprx0_xySp ADDA oprx9_xySp ADDA oprx16_xySp ADDA [D,xySp] ADDA [oprX16,xySp] | (A) + (M) \Rightarrow A Add without Carry to A | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8B ii 9B dd BB hh ll AB xb AB xb ff AB xb ee ff AB xb AB xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | Δ | - | Δ | Δ | Δ | Δ |
| ADDB #opr8i ADDB opr8a ADDB opr16a ADDB oprx0_xySp ADDB oprx9_xySp ADDB oprx16_xySp ADDB [D,xySp] ADDB [oprX16,xySp] | (B) + (M) \Rightarrow B Add without Carry to B | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CB ii DB dd FB hh ll EB xb EB xb ff EB xb ee ff EB xb EB xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | Δ | - | Δ | Δ | Δ | Δ |
| ADDD #opr16i ADDD opr8a ADDD opr16a ADDD oprx0_xySp ADDD oprx9_xySp ADDD oprx16_xySp ADDD [D,xySp] ADDD [oprX16,xySp] | (A:B) + (M:M+1) \Rightarrow A:B Add 16-Bit to D (A:B) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C3 jj kk D3 dd F3 hh ll E3 xb E3 xb ff E3 xb ee ff E3 xb E3 xb ee ff | OP RfP ROP RfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | Δ | Δ |
| ANDA #opr8i ANDA opr8a ANDA opr16a ANDA oprx0_xySp ANDA oprx9_xySp ANDA oprx16_xySp ANDA [D,xySp] ANDA [oprX16,xySp] | (A) • (M) \Rightarrow A Logical And A with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 84 ii 94 dd B4 hh ll A4 xb A4 xb ff A4 xb ee ff A4 xb A4 xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| ANDB #opr8i ANDB opr8a ANDB opr16a ANDB oprx0_xySp ANDB oprx9_xySp ANDB oprx16_xySp ANDB [D,xySp] ANDB [oprX16,xySp] | (B) • (M) \Rightarrow B Logical And B with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C4 ii D4 dd F4 hh ll E4 xb E4 xb ff E4 xb ee ff E4 xb E4 xb ee ff | P rfP ROP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |

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

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 2 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|---|---|--|---|---|---|---|---|---|---|---|---|
| ANDCC #opr8i | (CCR) • (M) ⇒ CCR Logical And CCR with Memory | IMM | 10 ii | P | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |
| ASL opr16a ASL oprx0_xysp ASL oprx9_xysp ASL oprx16_xysp ASL [D,xysp] ASL [opr16,xysp] ASLA ASLB |  Arithmetic Shift Left Arithmetic Shift Left Accumulator A Arithmetic Shift Left Accumulator B | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 78 hh 11 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58 | rOPw rPw rPOw frPPw fIfPrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| ASLD |  Arithmetic Shift Left Double | INH | 59 | O | - | - | - | - | Δ | Δ | Δ | Δ |
| ASR opr16a ASR oprx0_xysp ASR oprx9_xysp ASR oprx16_xysp ASR [D,xysp] ASR [opr16,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 11 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57 | rOPw rPw rPOw frPPw fIfPrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| BCC rel8 | Branch if Carry Clear (if C = 0) | REL | 24 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BCLR opr8a, msk8 BCLR opr16a, msk8 BCLR oprx0_xysp, msk8 BCLR oprx9_xysp, msk8 BCLR oprx16_xysp, msk8 | (M) • (mm) ⇒ M Clear Bit(s) in Memory | DIR EXT IDX IDX1 IDX2 | 4D dd mm 1D hh 11 mm 0D xb mm 0D xb ff mm 0D xb ee ff mm | rPOw rPPw rPOw rPwP frPwOP | - | - | - | - | Δ | Δ | 0 | - |
| BCS rel8 | Branch if Carry Set (if C = 1) | REL | 25 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BEQ rel8 | Branch if Equal (if Z = 1) | REL | 27 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BGE rel8 | Branch if Greater Than or Equal (if N ⊕ V = 0) (signed) | REL | 2C rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BGND | Place CPU in Background Mode | INH | 00 | VfPPP | - | - | - | - | - | - | - | - |
| BGT rel8 | Branch if Greater Than (if Z : (N ⊕ V) = 0) (signed) | REL | 2E rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BHI rel8 | Branch if Higher (if C : Z = 0) (unsigned) | REL | 22 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BHS rel8 | Branch if Higher or Same (if C = 0) (unsigned) same function as BCC | REL | 24 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BITA #opr8i BITA opr8a BITA opr16a BITA oprx0_xysp BITA oprx9_xysp BITA oprx16_xysp BITA [D,xysp] BITA [opr16,xysp] | (A) • (M) Logical And A with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 85 ii 95 dd B5 hh 11 A5 xb A5 xb ff A5 xb ee ff A5 xb A5 xb ee ff | P rfP rOP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| BITB #opr8i BITB opr8a BITB opr16a BITB oprx0_xysp BITB oprx9_xysp BITB oprx16_xysp BITB [D,xysp] BITB [opr16,xysp] | (B) • (M) Logical And B with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C5 ii D5 dd F5 hh 11 E5 xb E5 xb ff E5 xb ee ff E5 xb E5 xb ee ff | P rfP rOP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | 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 3 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|---|---|---|---|---|---|---|---|---|---|---|
| BLE rel8 | Branch if Less Than or Equal (if Z : (N ⊕ V) = 1) (signed) | REL | 2F rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BLO rel8 | Branch if Lower (if C = 1) (unsigned) same function as BCS | REL | 25 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BLS rel8 | Branch if Lower or Same (if C : Z = 1) (unsigned) | REL | 23 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BLT rel8 | Branch if Less Than (if N ⊕ V = 1) (signed) | REL | 2D rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BMI rel8 | Branch if Minus (if N = 1) | REL | 2B rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BNE rel8 | Branch if Not Equal (if Z = 0) | REL | 26 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BPL rel8 | Branch if Plus (if N = 0) | REL | 2A rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BRA rel8 | Branch Always (if 1 = 1) | REL | 20 rr | PPP | - | - | - | - | - | - | - | - |
| BRCLR opr8a, msk8, rel8 BRCLR opr16a, msk8, rel8 BRCLR oprx0_xy, msk8, rel8 BRCLR oprx9_xy, msk8, rel8 BRCLR oprx16_xy, msk8, rel8 | Branch if (M) • (mm) = 0 (if All Selected Bit(s) Clear) | DIR EXT IDX IDX1 IDX2 | 4F dd mm rr 1F hh 11 mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr | rPPP rfPPP rPPP rfffPPP frPffffPP | - | - | - | - | - | - | - | - |
| BRN rel8 | Branch Never (if 1 = 0) | REL | 21 rr | P | - | - | - | - | - | - | - | - |
| BRSET opr8, msk8, rel8 BRSET opr16a, msk8, rel8 BRSET oprx0_xy, msk8, rel8 BRSET oprx9_xy, msk8, rel8 BRSET oprx16_xy, msk8, rel8 | Branch if (M) • (mm) = 0 (if All Selected Bit(s) Set) | DIR EXT IDX IDX1 IDX2 | 4E dd mm rr 1E hh 11 mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr | rPPP rfPPP rPPP rfffPPP frPffffPP | - | - | - | - | - | - | - | - |
| BSET opr8, msk8 BSET opr16a, msk8 BSET oprx0_xy, msk8 BSET oprx9_xy, msk8 BSET oprx16_xy, msk8 | (M) ; (mm) ⇒ M Set Bit(s) in Memory | DIR EXT IDX IDX1 IDX2 | 4C dd mm 1C hh 11 mm 0C xb mm 0C xb ff mm 0C xb ee ff mm | rPOw rPPw rPOw rPwP frPwOP | - | - | - | - | Δ | Δ | 0 | - |
| BSR rel8 | (SP) - 2 ⇒ SP; RTN _H :RTN _L ⇒ M _(SP) :M _(SP+1) Subroutine address ⇒ PC Branch to Subroutine | REL | 07 rr | PPPS | - | - | - | - | - | - | - | - |
| BVC rel8 | Branch if Overflow Bit Clear (if V = 0) | REL | 28 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| BVS rel8 | Branch if Overflow Bit Set (if V = 1) | REL | 29 rr | PPP/P ¹ | - | - | - | - | - | - | - | - |
| CALL opr16a, page CALL oprx0_xy, page CALL oprx9_xy, page CALL oprx16_xy, page CALL [D,xy] CALL [opr16_xy] | (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 11 pg 4B xb pg 4B xb ff pg 4B xb ee ff pg 4B xb 4B xb ee ff | gnfSsPPP gnfSsPPP gnfSsPPP fgnfSsPPP flignSsPPP flignSsPPP | - | - | - | - | - | - | - | - |
| CBA | (A) - (B) Compare 8-Bit Accumulators | INH | 18 17 | OO | - | - | - | - | Δ | Δ | Δ | Δ |
| CLC | 0 ⇒ C Translates to ANDCC #\$FE | IMM | 10 FE | P | - | - | - | - | - | - | - | 0 |
| CLI | 0 ⇒ I Translates to ANDCC #\$EF (enables I-bit interrupts) | IMM | 10 EF | P | - | - | - | 0 | - | - | - | - |

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

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 4 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|---|---|---|--|---|---|---|---|---|---|---|---|
| CLR opr16a CLR oprx0_xysp CLR oprx9_xysp CLR oprx16_xysp CLR [D,xysp] CLR [opr16,xysp] CLRA CLRB | 0 → M Clear Memory Location 0 → A Clear Accumulator A 0 → B Clear Accumulator B | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 79 hh 11 69 xb 69 xb ff 69 xb ee ff 69 xb 69 xb ee ff 87 C7 | WOP PW PwO PwP PIfPw PIPPw O O | - | - | - | - | 0 | 1 | 0 | 0 |
| CLV | 0 → V <i>Translates to ANDCC #\$FD</i> | IMM | 10 FD | P | - | - | - | - | - | - | 0 | - |
| CMPA #opr8i CMPA opr8a CMPA opr16a CMPA oprx0_xysp CMPA oprx9_xysp CMPA oprx16_xysp CMPA [D,xysp] CMPA [opr16,xysp] | (A) – (M) Compare Accumulator A with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 81 ii 91 dd B1 hh 11 A1 xb A1 xb ff A1 xb ee ff A1 xb A1 xb ee ff | P rfP rop rfP rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| CMPB #opr8i CMPB opr8a CMPB opr16a CMPB oprx0_xysp CMPB oprx9_xysp CMPB oprx16_xysp CMPB [D,xysp] CMPB [opr16,xysp] | (B) – (M) Compare Accumulator B with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C1 ii D1 dd F1 hh 11 E1 xb E1 xb ff E1 xb ee ff E1 xb E1 xb ee ff | P rfP rop rfP rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| COM opr16a COM oprx0_xysp COM oprx9_xysp COM oprx16_xysp COM [D,xysp] COM [opr16,xysp] COMA COMB | (M̄) → M equivalent to \$FF – (M) → M 1's Complement Memory Location (Ā) → A Complement Accumulator A (B̄) → B Complement Accumulator B | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 71 hh 11 61 xb 61 xb ff 61 xb ee ff 61 xb 61 xb ee ff 41 51 | rOPw rPw rPOw frPPw fifrfPw fiprfPw O O | - | - | - | - | Δ | Δ | 0 | 1 |
| CPD #opr16i CPD opr8a CPD opr16a CPD oprx0_xysp CPD oprx9_xysp CPD oprx16_xysp CPD [D,xysp] CPD [opr16,xysp] | (A:B) – (M:M+1) Compare D to Memory (16-Bit) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8C jj kk 9C dd BC hh 11 AC xb AC xb ff AC xb ee ff AC xb AC xb ee ff | OP Rfp ROP Rfp Rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| CPS #opr16i CPS opr8a CPS opr16a CPS oprx0_xysp CPS oprx9_xysp CPS oprx16_xysp CPS [D,xysp] CPS [opr16,xysp] | (SP) – (M:M+1) Compare SP to Memory (16-Bit) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8F jj kk 9F dd BF hh 11 AF xb AF xb ff AF xb ee ff AF xb AF xb ee ff | OP Rfp ROP Rfp Rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| CPX #opr16i CPX opr8a CPX opr16a CPX oprx0_xysp CPX oprx9_xysp CPX oprx16_xysp CPX [D,xysp] CPX [opr16,xysp] | (X) – (M:M+1) Compare X to Memory (16-Bit) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8E jj kk 9E dd BE hh 11 AE xb AE xb ff AE xb ee ff AE xb AE xb ee ff | OP Rfp ROP Rfp Rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| CPY #opr16i CPY opr8a CPY opr16a CPY oprx0_xysp CPY oprx9_xysp CPY oprx16_xysp CPY [D,xysp] CPY [opr16,xysp] | (Y) – (M:M+1) Compare Y to Memory (16-Bit) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8D jj kk 9D dd BD hh 11 AD xb AD xb ff AD xb ee ff AD xb AD xb ee ff | OP Rfp ROP Rfp Rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |

Table A-1. Instruction Set Summary (Sheet 5 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|---|--|--|---|---|---|---|---|---|---|---|
| DAA | Adjust Sum to BCD Decimal Adjust Accumulator A | INH | 18 07 | OFO | - | - | - | - | Δ | Δ | ? | Δ |
| DBEQ abdxys, rel9 | (cntr) - 1 ⇒ cntr if (cntr) = 0, then Branch else Continue to next instruction Decrement Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP) | REL (9-bit) | 04 1b rr | PPP | - | - | - | - | - | - | - | - |
| DBNE abdxys, rel9 | (cntr) - 1 ⇒ cntr If (cntr) not = 0, then Branch; else Continue to next instruction Decrement Counter and Branch if ≠ 0 (cntr = A, B, D, X, Y, or SP) | REL (9-bit) | 04 1b rr | PPP | - | - | - | - | - | - | - | - |
| DEC opr16a DEC oprx0_xySp DEC oprx9_xySp DEC oprx16_xySp DEC [D,xySp] DEC [opr16,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 11 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff 43 53 | rOPw rPw rPOw frPPw fIfrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | - |
| DES | (SP) - \$0001 ⇒ SP Translates to LEAS -1,SP | IDX | 1B 9F | PP ¹ | - | - | - | - | - | - | - | - |
| DEX | (X) - \$0001 ⇒ X Decrement Index Register X | INH | 09 | O | - | - | - | - | - | Δ | - | - |
| DEY | (Y) - \$0001 ⇒ Y Decrement Index Register Y | INH | 03 | O | - | - | - | - | - | Δ | - | - |
| EDIV | (Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 × 16 Bit ⇒ 16 Bit Divide (unsigned) | INH | 11 | ffffffffffffO | - | - | - | - | Δ | Δ | Δ | Δ |
| EDIVS | (Y:D) ÷ (X) ⇒ Y Remainder ⇒ D 32 × 16 Bit ⇒ 16 Bit Divide (signed) | INH | 18 14 | 0xffffffffffffO | - | - | - | - | Δ | Δ | Δ | Δ |
| EMACS opr16a ² | (M _(X) :M _(X+1)) × (M _(Y) :M _(Y+1)) + (M-M+3) ⇒ M-M+3 16 × 16 Bit ⇒ 32 Bit Multiply and Accumulate (signed) | Special | 18 12 hh 11 | ORROffFRRFWWP | - | - | - | - | Δ | Δ | Δ | Δ |
| EMAXD oprx0_xySp EMAXD oprx9_xySp EMAXD oprx16_xySp EMAXD [D,xySp] EMAXD [opr16,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 | ORfP ORPO OfRPP OfIFRfP OfIPRfP | - | - | - | - | Δ | Δ | Δ | Δ |
| EMAXM oprx0_xySp EMAXM oprx9_xySp EMAXM oprx16_xySp EMAXM [D,xySp] EMAXM [opr16,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 | - | - | - | - | Δ | Δ | Δ | Δ |
| EMIND oprx0_xySp EMIND oprx9_xySp EMIND oprx16_xySp EMIND [D,xySp] EMIND [opr16,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 | ORfP ORPO OfRPP OfIFRfP OfIPRfP | - | - | - | - | Δ | Δ | Δ | Δ |
| EMINM oprx0_xySp EMINM oprx9_xySp EMINM oprx16_xySp EMINM [D,xySp] EMINM [opr16,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 | - | - | - | - | Δ | Δ | Δ | Δ |
| EMUL | (D) × (Y) ⇒ Y:D 16 × 16 Bit Multiply (unsigned) | INH | 13 | ffO | - | - | - | - | Δ | Δ | - | Δ |

Notes:

1. Due to internal CPU requirements, the program word fetch is performed twice to the same address during this instruction.
2. *opr16a* is an extended address specification. Both X and Y point to source operands.

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 6 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|---|--|---|---|---|---|---|----------|----------|----------|---------------|
| EMULS | (D) \times (Y) \Rightarrow Y:D 16 \times 16 Bit Multiply (signed) | INH | 18 13 | Ofo | - | - | - | - | Δ | Δ | - | Δ |
| EORA #opr8i EORA opr8a EORA opr16a EORA oprx0_xysp EORA oprx9_xysp EORA oprx16_xysp EORA [D,xysp] EORA [opr16,xysp] | (A) \oplus (M) \Rightarrow A Exclusive-OR A with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 88 ii 98 dd B8 hh 11 A8 xb A8 xb ff A8 xb ee ff A8 xb A8 xb ee ff | P rfP rop rfP rpo frPP fifrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| EORB #opr8i EORB opr8a EORB opr16a EORB oprx0_xysp EORB oprx9_xysp EORB oprx16_xysp EORB [D,xysp] EORB [opr16,xysp] | (B) \oplus (M) \Rightarrow B Exclusive-OR B with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C8 ii D8 dd F8 hh 11 E8 xb E8 xb ff E8 xb ee ff E8 xb E8 xb ee ff | P rfP rop rfP rpo frPP fifrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| ETBL oprx0_xysp | (M:M+1) + [(B) \times ((M+2:M+3) - (M:M+1))] \Rightarrow 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 address modes or extensions allowed) | IDX | 18 3F xb | ORRfffffP | - | - | - | - | Δ | Δ | - | \Rightarrow |
| EXG abcdxys,abcdxys | (r1) \leftrightarrow (r2) (if r1 and r2 same size) or \$00:(r1) \Rightarrow r2 (if r1 = 8-bit; r2 = 16-bit) or (r1 _{low}) \leftrightarrow (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 | - | - | - | - | - | - | - | - |
| FDIV | (D) \div (X) \Rightarrow X; Remainder \Rightarrow D 16 \times 16 Bit Fractional Divide | INH | 18 11 | 0xffffffffFO | - | - | - | - | - | Δ | Δ | Δ |
| IBEQ abdxys, rel9 | (cntr) + 1 \Rightarrow cntr If (cntr) = 0, then Branch else Continue to next instruction Increment Counter and Branch if \neq 0 (cntr = A, B, D, X, Y, or SP) | REL (9-bit) | 04 lb rr | PPP | - | - | - | - | - | - | - | - |
| IBNE abdxys, rel9 | (cntr) + 1 \Rightarrow cntr if (cntr) not = 0, then Branch; else Continue to next instruction Increment Counter and Branch if \neq 0 (cntr = A, B, D, X, Y, or SP) | REL (9-bit) | 04 lb rr | PPP | - | - | - | - | - | - | - | - |
| IDIV | (D) \div (X) \Rightarrow X; Remainder \Rightarrow D 16 \times 16 Bit Integer Divide (unsigned) | INH | 18 10 | 0xffffffffFO | - | - | - | - | - | Δ | 0 | Δ |
| IDIVS | (D) \div (X) \Rightarrow X; Remainder \Rightarrow D 16 \times 16 Bit Integer Divide (signed) | INH | 18 15 | 0xffffffffFO | - | - | - | - | Δ | Δ | Δ | Δ |
| INC opr16a INC oprx0_xysp INC oprx9_xysp INC oprx16_xysp INC [D,xysp] INC [opr16,xysp] INCA INCB | (M) + \$01 \Rightarrow M Increment Memory Byte (A) + \$01 \Rightarrow A (B) + \$01 \Rightarrow B Increment Accumulator A Increment Accumulator B | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 72 hh 11 62 xb 62 xb ff 62 xb ee ff 62 xb 62 xb ee ff 42 52 | rOpw rPw rPw frPPw fifrpw fIPrw O O | - | - | - | - | Δ | Δ | Δ | - |
| INS | (SP) + \$0001 \Rightarrow SP Translates to LEAS 1,SP | IDX | 1B 81 | pp ¹ | - | - | - | - | - | - | - | - |
| INX | (X) + \$0001 \Rightarrow X Increment Index Register X | INH | 08 | O | - | - | - | - | Δ | - | - | - |

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

Table A-1. Instruction Set Summary (Sheet 7 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|---|--|---|---|---|---|---|----------|----------|---|---|
| INY | (Y) + \$0001 \Rightarrow Y Increment Index Register Y | INH | 02 | O | - | - | - | - | - | Δ | - | - |
| JMP opr16a JMP oprx0_xysp JMP oprx9_xysp JMP oprx16_xysp JMP [D,xysp] JMP [opr16,xysp] | Subroutine address \Rightarrow PC Jump | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 06 hh 11 05 xb 05 xb ff 05 xb ee ff 05 xb 05 xb ee ff | PPP PPP PPP fPPP fifPPP fifPPP | - | - | - | - | - | - | - | - |
| JSR opr8a JSR opr16a JSR oprx0_xysp JSR oprx9_xysp JSR oprx16_xysp JSR [D,xysp] JSR [opr16,xysp] | (SP) - 2 \Rightarrow SP; RTN _H ;RTN _L \Rightarrow M _(SP) ;M _(SP+1) ; Subroutine address \Rightarrow PC Jump to Subroutine | DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 17 dd 16 hh 11 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff | PPPS PPPS PPPS PPPS fPPPS fifPPPS fifPPPS | - | - | - | - | - | - | - | - |
| LBCC rel16 | Long Branch if Carry Clear (if C = 0) | REL | 18 24 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBCS rel16 | Long Branch if Carry Set (if C = 1) | REL | 18 25 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBEQ rel16 | Long Branch if Equal (if Z = 1) | REL | 18 27 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBGE rel16 | Long Branch Greater Than or Equal (if N \oplus V = 0) (signed) | REL | 18 2C qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBGT rel16 | Long Branch if Greater Than (if Z : (N \oplus V) = 0) (signed) | REL | 18 2E qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBHI rel16 | Long Branch if Higher (if C ; Z = 0) (unsigned) | REL | 18 22 qq rr | 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 | - | - | - | - | - | - | - | - |
| LBLE rel16 | Long Branch if Less Than or Equal (if Z : (N \oplus V) = 1) (signed) | REL | 18 2F qq rr | 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 | - | - | - | - | - | - | - | - |
| LBLS rel16 | Long Branch if Lower or Same (if C ; Z = 1) (unsigned) | REL | 18 23 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBLT rel16 | Long Branch if Less Than (if N \oplus V = 1) (signed) | REL | 18 2D qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBMI rel16 | Long Branch if Minus (if N = 1) | REL | 18 2B qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBNE rel16 | Long Branch if Not Equal (if Z = 0) | REL | 18 26 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBPL rel16 | Long Branch if Plus (if N = 0) | REL | 18 2A qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBRA rel16 | Long Branch Always (if 1 = 1) | REL | 18 20 qq rr | OPPP | - | - | - | - | - | - | - | - |
| LBRN rel16 | Long Branch Never (if 1 = 0) | REL | 18 21 qq rr | OPO | - | - | - | - | - | - | - | - |
| LBVC rel16 | Long Branch if Overflow Bit Clear (if V = 0) | REL | 18 28 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LBVS rel16 | Long Branch if Overflow Bit Set (if V = 1) | REL | 18 29 qq rr | OPPP/OPO ¹ | - | - | - | - | - | - | - | - |
| LDAA #opr8i LDAA opr8a LDAA opr16a LDAA oprx0_xysp LDAA oprx9_xysp LDAA oprx16_xysp LDAA [D,xysp] LDAA [opr16,xysp] | (M) \Rightarrow A Load Accumulator A | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 86 ii 96 dd B6 hh 11 A6 xb A6 xb ff A6 xb ee ff A6 xb A6 xb ee ff | P rfP rop rfP rpo frPP fifrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |

Note 1. OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

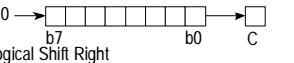
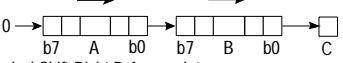
Instruction Reference

Table A-1. Instruction Set Summary (Sheet 8 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|---|---|--|---|---|---|---|----------|----------|----------|----------|
| LDAB #opr8i LDAB opr8a LDAB opr16a LDAB oprx0_xysp LDAB oprx9_xysp LDAB oprx16,xysp LDAB [D,xysp] LDAB [opr16,xysp] | (M) \Rightarrow B Load Accumulator B | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C6 ii D6 dd F6 hh 11 E6 xb E6 xb ff E6 xb ee ff E6 xb E6 xb ee ff | P rfP rop rfP rpo frpp fifrfp fiprfp | - | - | - | - | Δ | Δ | 0 | - |
| LDD #opr16i LDD opr8a LDD opr16a LDD oprx0_xysp LDD oprx9_xysp LDD oprx16,xysp LDD [D,xysp] LDD [opr16,xysp] | (M:M+1) \Rightarrow A:B Load Double Accumulator D (A:B) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CC jj kk DC dd FC hh 11 EC xb EC xb ff EC xb ee ff EC xb EC xb ee ff | OP rfP rop rfP rpo frpp fifrfp fiprfp | - | - | - | - | Δ | Δ | 0 | - |
| LDS #opr16i LDS opr8a LDS opr16a LDS oprx0_xysp LDS oprx9_xysp LDS oprx16,xysp LDS [D,xysp] LDS [opr16,xysp] | (M:M+1) \Rightarrow SP Load Stack Pointer | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CF jj kk DF dd FF hh 11 EF xb EF xb ff EF xb ee ff EF xb EF xb ee ff | OP rfP rop rfP rpo frpp fifrfp fiprfp | - | - | - | - | Δ | Δ | 0 | - |
| LDX #opr16i LDX opr8a LDX opr16a LDX oprx0_xysp LDX oprx9_xysp LDX oprx16,xysp LDX [D,xysp] LDX [opr16,xysp] | (M:M+1) \Rightarrow X Load Index Register X | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CE jj kk DE dd FE hh 11 EE xb EE xb ff EE xb ee ff EE xb EE xb ee ff | OP rfP rop rfP rpo frpp fifrfp fiprfp | - | - | - | - | Δ | Δ | 0 | - |
| LDY #opr16i LDY opr8a LDY opr16a LDY oprx0_xysp LDY oprx9_xysp LDY oprx16,xysp LDY [D,xysp] LDY [opr16,xysp] | (M:M+1) \Rightarrow Y Load Index Register Y | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CD jj kk DD dd FD hh 11 ED xb ED xb ff ED xb ee ff ED xb ED xb ee ff | OP rfP rop rfP rpo frpp fifrfp fiprfp | - | - | - | - | Δ | Δ | 0 | - |
| LEAS oprx0_xysp LEAS oprx9_xysp LEAS oprx16,xysp | Effective Address \Rightarrow SP Load Effective Address into SP | IDX IDX1 IDX2 | 1B xb 1B xb ff 1B xb ee ff | PP ¹ PO PP | - | - | - | - | - | - | - | - |
| LEAX oprx0_xysp LEAX oprx9_xysp LEAX oprx16,xysp | Effective Address \Rightarrow X Load Effective Address into X | IDX IDX1 IDX2 | 1A xb 1A xb ff 1A xb ee ff | PP ¹ PO PP | - | - | - | - | - | - | - | - |
| LEAY oprx0_xysp LEAY oprx9_xysp LEAY oprx16,xysp | Effective Address \Rightarrow Y Load Effective Address into Y | IDX IDX1 IDX2 | 19 xb 19 xb ff 19 xb ee ff | PP ¹ PO PP | - | - | - | - | - | - | - | - |
| LSL opr16a LSL oprx0_xysp LSL oprx9_xysp LSL oprx16,xysp LSL [D,xysp] LSL [opr16,xysp] LSLA LSLB | Logical Shift Left same function as ASL Logical Shift Accumulator A to Left Logical Shift Accumulator B to Left | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 78 hh 11 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58 | rOPw rpw rPOw frPPw fifrpw fiprw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| LSLD | Logical Shift Left D Accumulator same function as ASLD | INH | 59 | O | - | - | - | - | Δ | Δ | Δ | Δ |

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

Table A-1. Instruction Set Summary (Sheet 9 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|--|---|--|---|---|---|---|---|---|---|---|
| LSR opr16a LSR oprx0_xysp LSR oprx9_xysp LSR oprx16_xysp LSR [D,xysp] LSR [opr16,xysp] LSRA LSRB |  Logical Shift Right | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 74 hb 11 64 xb 64 xb ff 64 xb ee ff 64 xb 64 xb ee ff 44 54 | rOPw rPw rPO frPPw fIfPrPw fIPrPw O O | - | - | - | - | 0 | Δ | Δ | Δ |
| LSRD |  Logical Shift Right D Accumulator | INH | 49 | O | - | - | - | - | 0 | Δ | Δ | Δ |
| MAXA oprx0_xysp MAXA oprx9_xysp MAXA oprx16_xysp MAXA [D,xysp] MAXA [opr16,xysp] | MAX((A), (M)) → A 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 18 xb 18 18 xb ff 18 18 xb ee ff 18 18 xb 18 18 xb ee ff | OrfP OrPO OfrrP OfIfrfP OfIPrfP | - | - | - | - | Δ | Δ | Δ | Δ |
| MAXM oprx0_xysp MAXM oprx9_xysp MAXM oprx16_xysp MAXM [D,xysp] MAXM [opr16,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 OfrrPwP OfIfrPw OfIPrfPw | - | - | - | - | Δ | Δ | Δ | Δ |
| MEM | μ (grade) → 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 to fuzzy input (RAM location). | Special | 01 | RRfOw | - | - | ? | - | ? | ? | ? | ? |
| MINA oprx0_xysp MINA oprx9_xysp MINA oprx16_xysp MINA [D,xysp] MINA [opr16,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 | OrfP OrPO OfrrP OfIfrfP OfIPrfP | - | - | - | - | Δ | Δ | Δ | Δ |
| MINM oprx0_xysp MINM oprx9_xysp MINM oprx16_xysp MINM [D,xysp] MINM [opr16,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 OfrrPwP OfIfrPw OfIPrfPw | - | - | - | - | Δ | Δ | Δ | Δ |
| MOVB #opr8, opr16a ¹ MOVB #opr8i, oprx0_xysp ¹ MOVB opr16a, opr16a ¹ MOVB opr16a, oprx0_xysp ¹ MOVB oprx0_xysp, opr16a ¹ MOVB oprx0_xysp, oprx0_xysp ¹ | (M ₁) ⇒ M ₂ Memory to Memory Byte-Move (8-Bit) | IMM-EXT IMM-IDX EXT-EXT EXT-IDX IDX-EXT IDX-IDX | 18 0B ii hh ll 18 08 xb ii 18 0C hh ll hh ll 18 09 xb hh ll 18 0D xb hh ll 18 0A xb xb | OPwP OPwO OpPwPO OPrPw OrPwP OrPwO | - | - | - | - | - | - | - | - |
| MOVW #opr16, opr16a ¹ MOVW #opr16i, oprx0_xysp ¹ MOVW opr16a, opr16a ¹ MOVW opr16a, oprx0_xysp ¹ MOVW oprx0_xysp, opr16a ¹ MOVW oprx0_xysp, oprx0_xysp ¹ | (M:M+1 ₁) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit) | IMM-EXT IMM-IDX EXT-EXT EXT-IDX IDX-EXT IDX-IDX | 18 03 jj kk hh ll 18 00 xb jj kk 18 04 hh ll hh ll 18 01 xb hh ll 18 05 xb hh ll 18 02 xb xb | OPwPO OPPW OrPwPO OPrPw OrPwP OrPwO | - | - | - | - | - | - | - | - |
| MUL | (A) × (B) ⇒ A:B 8 × 8 Unsigned Multiply | INH | 12 | fFO | - | - | - | - | - | - | - | Δ |

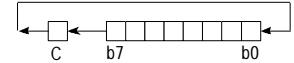
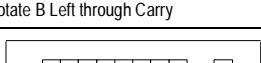
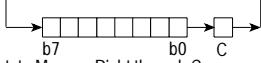
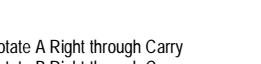
Note 1. The first operand in the source code statement specifies the source for the move.

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 10 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|---|--|---|------------|--------------|------------|------------|------------|------------|------------|------------|
| NEG opr16a NEG oprx0_xySp NEG oprx9_xySp NEG oprx16_xySp NEG [D_xySp] NEG [oprX16_xySp] NEGA NEGB | 0 - (M) \Rightarrow M or $\bar{(M)} + 1 \Rightarrow M$ Two's Complement Negate 0 - (A) \Rightarrow A equivalent to $\bar{(A)} + 1 \Rightarrow A$ Negate Accumulator A 0 - (B) \Rightarrow B equivalent to $\bar{(B)} + 1 \Rightarrow B$ Negate Accumulator B | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | 70 hh 11 60 xb 60 xb ff 60 xb ee ff 60 xb 60 xb ee ff 40 50 | rOPw rPw rPOw frPPw fIfPrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| NOP | No Operation | INH | A7 | O | - | - | - | - | - | - | - | - |
| ORAA #opr8i ORAA opr8a ORAA opr16a ORAA oprx0_xySp ORAA oprx9_xySp ORAA oprx16_xySp ORAA [D_xySp] ORAA [oprX16_xySp] | (A) ; (M) \Rightarrow A Logical OR A with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 8A ii 9A dd BA hh 11 AA xb AA xb ff AA xb ee ff AA xb AA xb ee ff | P rfP rOP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| ORAB #opr8i ORAB opr8a ORAB opr16a ORAB oprx0_xySp ORAB oprx9_xySp ORAB oprx16_xySp ORAB [D_xySp] ORAB [oprX16_xySp] | (B) ; (M) \Rightarrow B Logical OR B with Memory | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | CA ii DA dd FA hh 11 EA xb EA xb ff EA xb ee ff EA xb EA xb ee ff | P rfP rOP rfP rPO frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | 0 | - |
| ORCC #opr8i | (CCR) ; M \Rightarrow CCR Logical OR CCR with Memory | IMM | 14 ii | P | \uparrow | - | \uparrow | \uparrow | \uparrow | \uparrow | \uparrow | \uparrow |
| PSHA | (SP) - 1 \Rightarrow SP; (A) \Rightarrow M _(SP) Push Accumulator A onto Stack | INH | 36 | Os | - | - | - | - | - | - | - | - |
| PSHB | (SP) - 1 \Rightarrow SP; (B) \Rightarrow M _(SP) Push Accumulator B onto Stack | INH | 37 | Os | - | - | - | - | - | - | - | - |
| PSHC | (SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M _(SP) Push CCR onto Stack | INH | 39 | Os | - | - | - | - | - | - | - | - |
| PSHD | (SP) - 2 \Rightarrow SP; (A:B) \Rightarrow M _(SP) :M _(SP+1) Push D Accumulator onto Stack | INH | 3B | OS | - | - | - | - | - | - | - | - |
| PSHX | (SP) - 2 \Rightarrow SP; (X _H ;X _L) \Rightarrow M _(SP) :M _(SP+1) Push Index Register X onto Stack | INH | 34 | OS | - | - | - | - | - | - | - | - |
| PSHY | (SP) - 2 \Rightarrow SP; (Y _H ;Y _L) \Rightarrow M _(SP) :M _(SP+1) Push Index Register Y onto Stack | INH | 35 | OS | - | - | - | - | - | - | - | - |
| PULA | (M _(SP)) \Rightarrow A; (SP) + 1 \Rightarrow SP Pull Accumulator A from Stack | INH | 32 | ufO | - | - | - | - | - | - | - | - |
| PULB | (M _(SP)) \Rightarrow B; (SP) + 1 \Rightarrow SP Pull Accumulator B from Stack | INH | 33 | ufO | - | - | - | - | - | - | - | - |
| PULC | (M _(SP)) \Rightarrow CCR; (SP) + 1 \Rightarrow SP Pull CCR from Stack | INH | 38 | ufO | Δ | \downarrow | 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 | - | - | - | - | - | - | - | - |

Table A-1. Instruction Set Summary (Sheet 11 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|---|--|--|---|---|---|---|---|---|---|---|
| PULX | $(M_{(SP)}:M_{(SP+1)}) \Rightarrow X_H:X_L$; $(SP) + 2 \Rightarrow SP$ Pull Index Register X from Stack | INH | 30 | 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 | - | - | - | - | - | - | - | - |
| REV (add if interrupted) | 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(tTx) ¹ ff + Orf | - | - | ? | - | ? | ? | Δ | ? |
| REVW (add 2 at end of ins if wts) (add if interrupted) | 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 \$FFE separates rule inputs from rule outputs. \$FFF terminates the rule list. REVW may be interrupted. | Special | 18 3B | ORf(tTx) ² (rfffRf) ² fff + ORft | - | - | ? | - | ? | ? | Δ | ! |
| ROL opr16a ROL oprx0_xySp ROL oprx9_xySp ROL oprx16_xySp ROL [D,xySp] ROL [opr16_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 11 65 xb 65 xb ff 65 xb ee ff 65 xb 65 xb ee ff 45 55 | rOPw rPw rPOw frPPw fIfrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| ROR opr16a ROR oprx0_xySp ROR oprx9_xySp ROR oprx16_xySp ROR [D,xySp] ROR [opr16_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 11 66 xb 66 xb ff 66 xb ee ff 66 xb 66 xb ee ff 46 56 | rOPw rPw rPOw frPPw fIfrPw fIPrPw O O | - | - | - | - | Δ | Δ | Δ | Δ |
| RTC | $(M_{(SP)}) \Rightarrow PPAGE$; $(SP) + 1 \Rightarrow SP$; $(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L$; $(SP) + 2 \Rightarrow SP$ Return from Call | INH | 0A | uUnPPP | - | - | - | - | - | - | - | - |

Notes:

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.
2. The 3-cycle loop in parentheses expands to 5 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.

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 12 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|---|--|---|---|---|---|---|---|---|---|---|
| RTI (if interrupt pending) | $(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)}) \Rightarrow B:A; (SP) + 2 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)}) \Rightarrow X_L:Y_L; (SP) + 4 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L; (SP) - 2 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)}) \Rightarrow Y_H:Y_L;$ $(SP) + 4 \Rightarrow SP$ Return from Interrupt | INH | 0B | uuuuuppp uuuuuvfppp | Δ | ↓ | Δ | Δ | Δ | Δ | Δ | Δ |
| RTS | $(M_{(SP)}:M_{(SP+1)}) \Rightarrow PC_H:PC_L;$ $(SP) + 2 \Rightarrow SP$ Return from Subroutine | INH | 3D | ufppp | - | - | - | - | - | - | - | - |
| SBA | $(A) - (B) \Rightarrow A$ Subtract B from A | INH | 18 16 | 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] | $(A) - (M) - C \Rightarrow A$ 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 rfP rop rfP rpo frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | Δ | Δ |
| SBCB #opr8i SBCB opr8a SBCB opr16a SBCB oprx0_xySp SBCB oprx9_xySp SBCB oprx16_xySp SBCB [D_xySp] SBCB [oprX16_xySp] | $(B) - (M) - C \Rightarrow B$ 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 rfP rop rfP rpo frPP fIfrfP fIPrfP | - | - | - | - | Δ | Δ | Δ | Δ |
| SEC | $1 \Rightarrow C$ <i>Translates to ORCC #01</i> | IMM | 14 01 | P | - | - | - | - | - | - | - | 1 |
| SEI | $1 \Rightarrow I$; (inhibit I interrupts) <i>Translates to ORCC #10</i> | IMM | 14 10 | P | - | - | - | 1 | - | - | - | - |
| SEV | $1 \Rightarrow V$ <i>Translates to ORCC #02</i> | IMM | 14 02 | P | - | - | - | - | - | - | - | 1 |
| SEX abc,dxys | \$00:(r1) $\Rightarrow r2$ if r1, bit 7 is 0 or \$FF:(r1) $\Rightarrow r2$ 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 <i>Alternate mnemonic for TFR r1, r2</i> | INH | B7 eb | P | - | - | - | - | - | - | - | - |
| STAA opr8a STAA opr16a STAA oprx0_xySp STAA oprx9_xySp STAA oprx16_xySp STAA [D_xySp] STAA [oprX16_xySp] | $(A) \Rightarrow M$ 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 wOP Pw PwO PwP PIfPw PIPw | - | - | - | - | Δ | Δ | 0 | - |
| STAB opr8a STAB opr16a STAB oprx0_xySp STAB oprx9_xySp STAB oprx16_xySp STAB [D_xySp] STAB [oprX16_xySp] | $(B) \Rightarrow M$ 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 wOP Pw PwO PwP PIfPw PIPw | - | - | - | - | Δ | Δ | 0 | - |

Table A-1. Instruction Set Summary (Sheet 13 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|---|---|---|--|---|---|---|---|----------|----------|----------|----------|
| STD opr8a STD opr16a STD oprx0_xySp STD oprx9_xySp STD oprx16_xySp STD [D,xySp] STD [opr16_xySp] | (A) \Rightarrow M, (B) \Rightarrow M+1 Store Double Accumulator | DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 5C dd 7C hh 11 6C xb 6C xb ff 6C xb ee ff 6C xb 6C xb ee ff | PW WOP PW PWO PWP PIfPW PIPPW | - | - | - | - | Δ | Δ | 0 | - |
| STOP (entering STOP) (exiting STOP) (continue) (if STOP disabled) | (SP) - 2 \Rightarrow SP; RTN _H ;RTN _L \Rightarrow M _(SP) ;M _(SP+1) ; (SP) - 2 \Rightarrow SP; (Y _H ;Y _L) \Rightarrow M _(SP) ;M _(SP+1) ; (SP) - 2 \Rightarrow SP; (X _H ;X _L) \Rightarrow M _(SP) ;M _(SP+1) ; (SP) - 2 \Rightarrow SP; (B:A) \Rightarrow M _(SP) ;M _(SP+1) ; (SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M _(SP) ; STOP All Clocks If S control bit = 1, the STOP instruction is disabled and acts like a two-cycle NOP. Registers stacked to allow quicker recovery by interrupt. | INH | 18 3E | OSSSfSS fVfPPP fO OO | - | - | - | - | - | - | - | - |
| STS opr8a STS opr16a STS oprx0_xySp STS oprx9_xySp STS oprx16_xySp STS [D,xySp] STS [opr16_xySp] | (SP _H ;SP _L) \Rightarrow M:M+1 Store Stack Pointer | DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 5F dd 7F hh 11 6F xb 6F xb ff 6F xb ee ff 6F xb 6F xb ee ff | PW WOP PW PWO PWP PIfPW PIPPW | - | - | - | - | Δ | Δ | 0 | - |
| STX opr8a STX opr16a STX oprx0_xySp STX oprx9_xySp STX oprx16_xySp STX [D,xySp] STX [opr16_xySp] | (X _H ;X _L) \Rightarrow M:M+1 Store Index Register X | DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 5E dd 7E hh 11 6E xb 6E xb ff 6E xb ee ff 6E xb 6E xb ee ff | PW WOP PW PWO PWP PIfPW PIPPW | - | - | - | - | Δ | Δ | 0 | - |
| STY opr8a STY opr16a STY oprx0_xySp STY oprx9_xySp STY oprx16_xySp STY [D,xySp] STY [opr16_xySp] | (Y _H ;Y _L) \Rightarrow M:M+1 Store Index Register Y | DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 5D dd 7D hh 11 6D xb 6D xb ff 6D xb ee ff 6D xb 6D xb ee ff | PW WOP PW PWO PWP PIfPW PIPPW | - | - | - | - | Δ | Δ | 0 | - |
| SUBA #opr8i SUBA opr8a SUBA opr16a SUBA oprx0_xySp SUBA oprx9_xySp SUBA oprx16_xySp SUBA [D,xySp] SUBA [opr16_xySp] | (A) - (M) \Rightarrow A Subtract Memory from Accumulator A | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 80 ii 90 dd B0 hh 11 A0 xb A0 xb ff A0 xb ee ff A0 xb A0 xb ee ff | P rfP rop rfP rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| SUBB #opr8i SUBB opr8a SUBB opr16a SUBB oprx0_xySp SUBB oprx9_xySp SUBB oprx16_xySp SUBB [D,xySp] SUBB [opr16_xySp] | (B) - (M) \Rightarrow B Subtract Memory from Accumulator B | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | C0 ii D0 dd F0 hh 11 E0 xb E0 xb ff E0 xb ee ff E0 xb E0 xb ee ff | P rfP rop rfP rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |
| SUBD #opr16i SUBD opr8a SUBD opr16a SUBD oprx0_xySp SUBD oprx9_xySp SUBD oprx16_xySp SUBD [D,xySp] SUBD [opr16_xySp] | (D) - (M:M+1) \Rightarrow D Subtract Memory from D (A:B) | IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2] | 83 jj kk 93 dd B3 hh 11 A3 xb A3 xb ff A3 xb ee ff A3 xb A3 xb ee ff | OP rfP rop rfP rpo frpp fifrfP fiprfP | - | - | - | - | Δ | Δ | Δ | Δ |

Instruction Reference

Table A-1. Instruction Set Summary (Sheet 14 of 15)

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|---|--|---|--|---|----------|--------------|----------|----------|----------|----------|----------|----------|
| SWI | (SP) - 2 \Rightarrow SP; RTN _H :RTN _L \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (Y _H ;Y _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (X _H ;X _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (B:A) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M _(SP) 1 \Rightarrow I; (SWI Vector) \Rightarrow PC Software Interrupt | INH | 3F | VSPSSPSSP ¹ | - | - | - | 1 | - | - | - | - |
| TAB | (A) \Rightarrow B Transfer A to B | INH | 18 0E | OO | - | - | - | - | Δ | Δ | 0 | - |
| TAP | (A) \Rightarrow CCR Translates to TFR A, CCR | INH | B7 02 | P | Δ | \Downarrow | Δ | Δ | Δ | Δ | Δ | Δ |
| TBA | (B) \Rightarrow A Transfer B to A | INH | 18 0F | 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 1b rr | PPP | - | - | - | - | - | - | - | - |
| TBL oprx0_xysp | (M) + [(B) \times ((M+1) - (M))] \Rightarrow 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 | 0rrffffP | - | - | - | - | 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 1b rr | PPP | - | - | - | - | - | - | - | - |
| TFR abcdxys,abcdxys | (r1) \Rightarrow r2 or \$00:(r1) \Rightarrow r2 or (r1[7:0]) \Rightarrow r2 Transfer Register to Register r1 and r2 may be A, B, CCR, D, X, Y, or SP | INH | B7 eb | P | - | - | - | - | - | - | - | - |
| TPA | (CCR) \Rightarrow A Translates to TFR CCR ,A | INH | B7 20 | P | - | - | - | - | - | - | - | - |
| TRAP trapnum | (SP) - 2 \Rightarrow SP; RTN _H :RTN _L \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (Y _H ;Y _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (X _H ;X _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (B:A) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M _(SP) 1 \Rightarrow I; (TRAP Vector) \Rightarrow PC Unimplemented opcode trap | INH | 18 tn tn = \$30-\$39 or \$40-\$FF | OfVSPSSPSSP | - | - | - | 1 | - | - | - | - |
| TST opr16a TST oprx0_xysp TST oprx9_xysp TST oprx16_xysp TST [D,xysp] TST [opr16,xysp] | (M) - 0 Test Memory for Zero or Minus | EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH | F7 hh 11 E7 xb E7 xb ff E7 xb ee ff E7 xb E7 xb ee ff 97 D7 | rOP rfP rPO frPP fIfrfP fIPrfP O O | - | - | - | - | Δ | Δ | 0 | 0 |
| TSTA TSB | (A) - 0 Test A for Zero or Minus (B) - 0 Test B for Zero or Minus | INH INH | | | | | | | | | | |
| TSX | (SP) \Rightarrow X Translates to TFR SP,X | INH | B7 75 | P | - | - | - | - | - | - | - | - |

Note 1. The CPU also uses the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation of the sequence (VfPPP) is used for resets.

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

| Source Form | Operation | Addr. Mode | Machine Coding (hex) | Access Detail | S | X | H | I | N | Z | V | C |
|--|--|------------|----------------------|------------------------------------|---|---|---|---|---|----------|---|---|
| TSY | (SP) \Rightarrow Y <i>Translates to TFR SP,Y</i> | INH | B7 76 | P | - | - | - | - | - | - | - | - |
| TXS | (X) \Rightarrow SP <i>Translates to TFR X,SP</i> | INH | B7 57 | P | - | - | - | - | - | - | - | - |
| TYS | (Y) \Rightarrow SP <i>Translates to TFR Y,SP</i> | INH | B7 67 | P | - | - | - | - | - | - | - | - |
| WAI (before interrupt) (when interrupt comes) | (SP) - 2 \Rightarrow SP; RTN _H :RTN _L \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (Y _H :Y _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (X _H :X _L) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 2 \Rightarrow SP; (B:A) \Rightarrow M _(SP) :M _(SP+1) ; (SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M _(SP) ; WAIT for interrupt | INH | 3E | OSSSfSsf VfPPP | - | - | - | - | - | - | - | - |
| or or - | | | | | | | | | | | | |
| WAV (add if interrupt) | B $\sum_{i=1} S_i F_i \Rightarrow Y:D$ B $\sum_{i=1} F_i \Rightarrow X$ Calculate Sum of Products and Sum of Weights for Weighted Average Calculation Initialize B, X, and Y before WAV. B specifies number of elements. X points at first element in S _i list. Y points at first element in F _i list. All S _i and F _i elements are 8-bits. If interrupted, six extra bytes of stack used for intermediate values | Special | 18 3C | Off(frr- fffff)O SSS + UUUrr | - | - | ? | - | ? | Δ | ? | ? |
| wavr | see WAV | Special | 3C | | - | - | ? | - | ? | Δ | ? | ? |
| pseudo-instruction | Resume executing an interrupted WAV instruction (recover intermediate results from stack rather than initializing them to zero) | | | | | | | | | | | |
| XGDX | (D) \leftrightarrow (X) <i>Translates to EXG D, X</i> | INH | B7 C5 | P | - | - | - | - | - | - | - | - |
| XGDY | (D) \leftrightarrow (Y) <i>Translates to EXG D, Y</i> | INH | B7 C6 | P | - | - | - | - | - | - | - | - |

Table A-2. 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 |

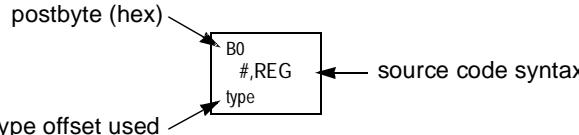
Key to Table A-2

Table A-3. 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) $-256 \leq n \leq 255$ 1 = 16-bit $-32,768 \leq n \leq 65,535$ if z = s = 1, 16-bit offset indexed-indirect (see entry) r can specify X, Y, SP, or PC |
| 111rr011 | [n,r] | 16-bit offset indexed-indirect rr can specify X, Y, SP, or PC $-32,768 \leq n \leq 65,535$ |
| rr1pnnnn | n,-r n,r- n,r+ | Auto pre-decrement/increment or auto post-decrement/increment; 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 |

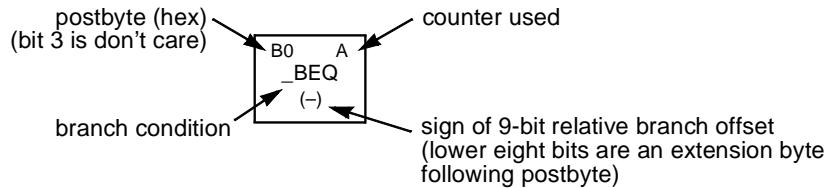
Table A-4. Transfer and Exchange Postbyte Encoding

| Transfers | | | | | | | | | |
|-----------------|------------------|---|---|---|---|---|---|---|---|
| $\downarrow LS$ | $MS \Rightarrow$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | | $A \Rightarrow A$ | $B \Rightarrow A$ | $CCR \Rightarrow A$ | $TMP3_L \Rightarrow A$ | $B \Rightarrow A$ | $X_L \Rightarrow A$ | $Y_L \Rightarrow A$ | $SP_L \Rightarrow A$ |
| 1 | | $A \Rightarrow B$ | $B \Rightarrow B$ | $CCR \Rightarrow B$ | $TMP3_L \Rightarrow B$ | $B \Rightarrow B$ | $X_L \Rightarrow B$ | $Y_L \Rightarrow B$ | $SP_L \Rightarrow B$ |
| 2 | | $A \Rightarrow CCR$ | $B \Rightarrow CCR$ | $CCR \Rightarrow CCR$ | $TMP3_L \Rightarrow CCR$ | $B \Rightarrow CCR$ | $X_L \Rightarrow CCR$ | $Y_L \Rightarrow CCR$ | $SP_L \Rightarrow CCR$ |
| 3 | | $sex:A \Rightarrow TMP2$ | $sex:B \Rightarrow TMP2$ | $sex:CCR \Rightarrow TMP2$ | $TMP3 \Rightarrow TMP2$ | $D \Rightarrow TMP2$ | $X \Rightarrow TMP2$ | $Y \Rightarrow TMP2$ | $SP \Rightarrow TMP2$ |
| 4 | | $sex:A \Rightarrow D$ SEX A,D | $sex:B \Rightarrow D$ SEX B,D | $sex:CCR \Rightarrow D$ SEX CCR,D | $TMP3 \Rightarrow D$ | $D \Rightarrow D$ | $X \Rightarrow D$ | $Y \Rightarrow D$ | $SP \Rightarrow D$ |
| 5 | | $sex:A \Rightarrow X$ SEX A,X | $sex:B \Rightarrow X$ SEX B,X | $sex:CCR \Rightarrow X$ SEX CCR,X | $TMP3 \Rightarrow X$ | $D \Rightarrow X$ | $X \Rightarrow X$ | $Y \Rightarrow X$ | $SP \Rightarrow X$ |
| 6 | | $sex:A \Rightarrow Y$ SEX A,Y | $sex:B \Rightarrow Y$ SEX B,Y | $sex:CCR \Rightarrow Y$ SEX CCR,Y | $TMP3 \Rightarrow Y$ | $D \Rightarrow Y$ | $X \Rightarrow Y$ | $Y \Rightarrow Y$ | $SP \Rightarrow Y$ |
| 7 | | $sex:A \Rightarrow SP$ SEX A,SP | $sex:B \Rightarrow SP$ SEX B,SP | $sex:CCR \Rightarrow SP$ SEX CCR,SP | $TMP3 \Rightarrow SP$ | $D \Rightarrow SP$ | $X \Rightarrow SP$ | $Y \Rightarrow SP$ | $SP \Rightarrow SP$ |
| Exchanges | | | | | | | | | |
| $\downarrow LS$ | $MS \Rightarrow$ | 8 | 9 | A | B | C | D | E | F |
| 0 | | $A \Leftrightarrow A$ | $B \Leftrightarrow A$ | $CCR \Leftrightarrow A$ | $TMP3_L \Rightarrow A$ $\$00:A \Rightarrow TMP3$ | $B \Rightarrow A$ $A \Rightarrow B$ | $X_L \Rightarrow A$ $\$00:A \Rightarrow X$ | $Y_L \Rightarrow A$ $\$00:A \Rightarrow Y$ | $SP_L \Rightarrow A$ $\$00:A \Rightarrow SP$ |
| 1 | | $A \Leftrightarrow B$ | $B \Leftrightarrow B$ | $CCR \Leftrightarrow B$ | $TMP3_L \Rightarrow B$ $\$FF:B \Rightarrow TMP3$ | $B \Rightarrow B$ $\$FF \Rightarrow A$ | $X_L \Rightarrow B$ $\$FF:B \Rightarrow X$ | $Y_L \Rightarrow B$ $\$FF:B \Rightarrow Y$ | $SP_L \Rightarrow B$ $\$FF:B \Rightarrow SP$ |
| 2 | | $A \Leftrightarrow CCR$ | $B \Leftrightarrow CCR$ | $CCR \Leftrightarrow CCR$ | $TMP3_L \Rightarrow CCR$ $\$FF:CCR \Rightarrow TMP3$ | $B \Rightarrow CCR$ $\$FF:CCR \Rightarrow D$ | $X_L \Rightarrow CCR$ $\$FF:CCR \Rightarrow X$ | $Y_L \Rightarrow CCR$ $\$FF:CCR \Rightarrow Y$ | $SP_L \Rightarrow CCR$ $\$FF:CCR \Rightarrow SP$ |
| 3 | | $\$00:A \Rightarrow TMP2$ $TMP2_L \Rightarrow A$ | $\$00:B \Rightarrow TMP2$ $TMP2_L \Rightarrow B$ | $\$00:CCR \Rightarrow TMP2$ $TMP2_L \Rightarrow CCR$ | $TMP3 \Leftrightarrow TMP2$ | $D \Leftrightarrow TMP2$ | $X \Leftrightarrow TMP2$ | $Y \Leftrightarrow TMP2$ | $SP \Leftrightarrow TMP2$ |
| 4 | | $\$00:A \Rightarrow D$ | $\$00:B \Rightarrow D$ | $\$00:CCR \Rightarrow D$ $B \Rightarrow CCR$ | $TMP3 \Leftrightarrow D$ | $D \Leftrightarrow D$ | $X \Leftrightarrow D$ | $Y \Leftrightarrow D$ | $SP \Leftrightarrow D$ |
| 5 | | $\$00:A \Rightarrow X$ $X_L \Rightarrow A$ | $\$00:B \Rightarrow X$ $X_L \Rightarrow B$ | $\$00:CCR \Rightarrow X$ $X_L \Rightarrow CCR$ | $TMP3 \Leftrightarrow X$ | $D \Leftrightarrow X$ | $X \Leftrightarrow X$ | $Y \Leftrightarrow X$ | $SP \Leftrightarrow X$ |
| 6 | | $\$00:A \Rightarrow Y$ $Y_L \Rightarrow A$ | $\$00:B \Rightarrow Y$ $Y_L \Rightarrow B$ | $\$00:CCR \Rightarrow Y$ $Y_L \Rightarrow CCR$ | $TMP3 \Leftrightarrow Y$ | $D \Leftrightarrow Y$ | $X \Leftrightarrow Y$ | $Y \Leftrightarrow Y$ | $SP \Leftrightarrow Y$ |
| 7 | | $\$00:A \Rightarrow SP$ $SP_L \Rightarrow A$ | $\$00:B \Rightarrow SP$ $SP_L \Rightarrow B$ | $\$00:CCR \Rightarrow SP$ $SP_L \Rightarrow CCR$ | $TMP3 \Leftrightarrow SP$ | $D \Leftrightarrow SP$ | $X \Leftrightarrow SP$ | $Y \Leftrightarrow SP$ | $SP \Leftrightarrow SP$ |

TMP2 and TMP3 registers are for factory use only.

Table A-5. Loop Primitive Postbyte Encoding (lb)

| | | | | | | | | | | | | | | | | | | | | | | | |
|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|
| 00 | A | 10 | A | 20 | A | 30 | A | 40 | A | 50 | A | 60 | A | 70 | A | 80 | A | 90 | A | A0 | A | B0 | A |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |
| 01 | B | 11 | B | 21 | B | 31 | B | 41 | B | 51 | B | 61 | B | 71 | B | 81 | B | 91 | B | A1 | B | B1 | B |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |
| 02 | — | 12 | — | 22 | — | 32 | — | 42 | — | 52 | — | 62 | — | 72 | — | 82 | — | 92 | — | A2 | — | B2 | — |
| — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | — | A3 | — | B3 | — |
| 04 | D | 14 | D | 24 | D | 34 | D | 44 | D | 54 | D | 64 | D | 74 | D | 84 | D | 94 | D | A4 | D | B4 | D |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |
| 05 | X | 15 | X | 25 | X | 35 | X | 45 | X | 55 | X | 65 | X | 75 | X | 85 | X | 95 | X | A5 | X | B5 | X |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |
| 06 | Y | 16 | Y | 26 | Y | 36 | Y | 46 | Y | 56 | Y | 66 | Y | 76 | Y | 86 | Y | 96 | Y | A6 | Y | B6 | Y |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |
| 07 | SP | 17 | SP | 27 | SP | 37 | SP | 47 | SP | 57 | SP | 67 | SP | 77 | SP | 87 | SP | 97 | SP | A7 | SP | B7 | SP |
| DBEQ | (+) | DBEQ | (-) | DBNE | (+) | DBNE | (-) | TBEQ | (+) | TBEQ | (-) | TBNE | (+) | TBNE | (-) | IBEQ | (+) | IBEQ | (-) | IBNE | (+) | IBNE | (-) |

Key to Table A-5**Table A-6. Branch/Complementary Branch**

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

Note: For 16-bit offset long branches precede opcode with a \$18 page prebyte.

A.8 Memory Expansion

Some M68HC12 derivatives support >4 megabytes of program memory.

Memory precedence

— Highest —

On-chip registers (usually \$0000 or \$1000)

BDM ROM (only when BDM active)

On-chip RAM

On-chip EEPROM

On-chip program memory (FLASH or ROM)

Expansion windows (on MCUs with expanded memory)

Other external memory

— Lowest —

CPU sees 64 Kbytes of address space (CPU_ADDR [15:0])

PPAGE 8-bit register to select 1 of 256 — 16-Kbyte program pages

DPAGE 8-bit register to select 1 of 256 — 4-Kbyte data pages

EPAGE 8-bit register to select 1 of 256 — 1-Kbyte extra pages

Extended address is 22 bits (EXT_ADDR [21:0])

Program expansion window works with CALL and RTC instructions to simplify program access to extended memory space. Data and extra expansion windows (when present) use traditional banked expansion memory techniques.

Program window

If CPU_ADDR [15:0] = \$8000–BFFF and PWEN = 1

Then EXT_ADDR [21:0] = PPAGE [7:0]:CPU_ADDR [13:0]

Program window works with CALL/RTC to automate bank switching.

256 pages (banks) of 16 Kbytes each = 4 M.

Data window

If CPU_ADDR [15:0] = \$7000–7FFF and DWEN = 1

Then EXT_ADDR [21:0] = 1:1:DPAGE [7:0]:CPU_ADDR [11:0]

User program controls DPAGE value

Extra window

- If CPU_ADDR [15:0] = \$0000–03FF and EWDIR = 1
 and EWEN = 1
- or CPU_ADDR [15:0] = \$0400–07FF and EWDIR = 0
 and EWEN = 1
- Then EXT_ADDR [21:0] = 1:1:1:1:EPAGE [7:0]:CPU_ADDR [9:0]

User program controls EPAGE value

CPU address not in any enabled window

EXT_ADDR [21:0] = 1:1:1:1:1:CPU_ADDR [15:0]

Instruction Reference

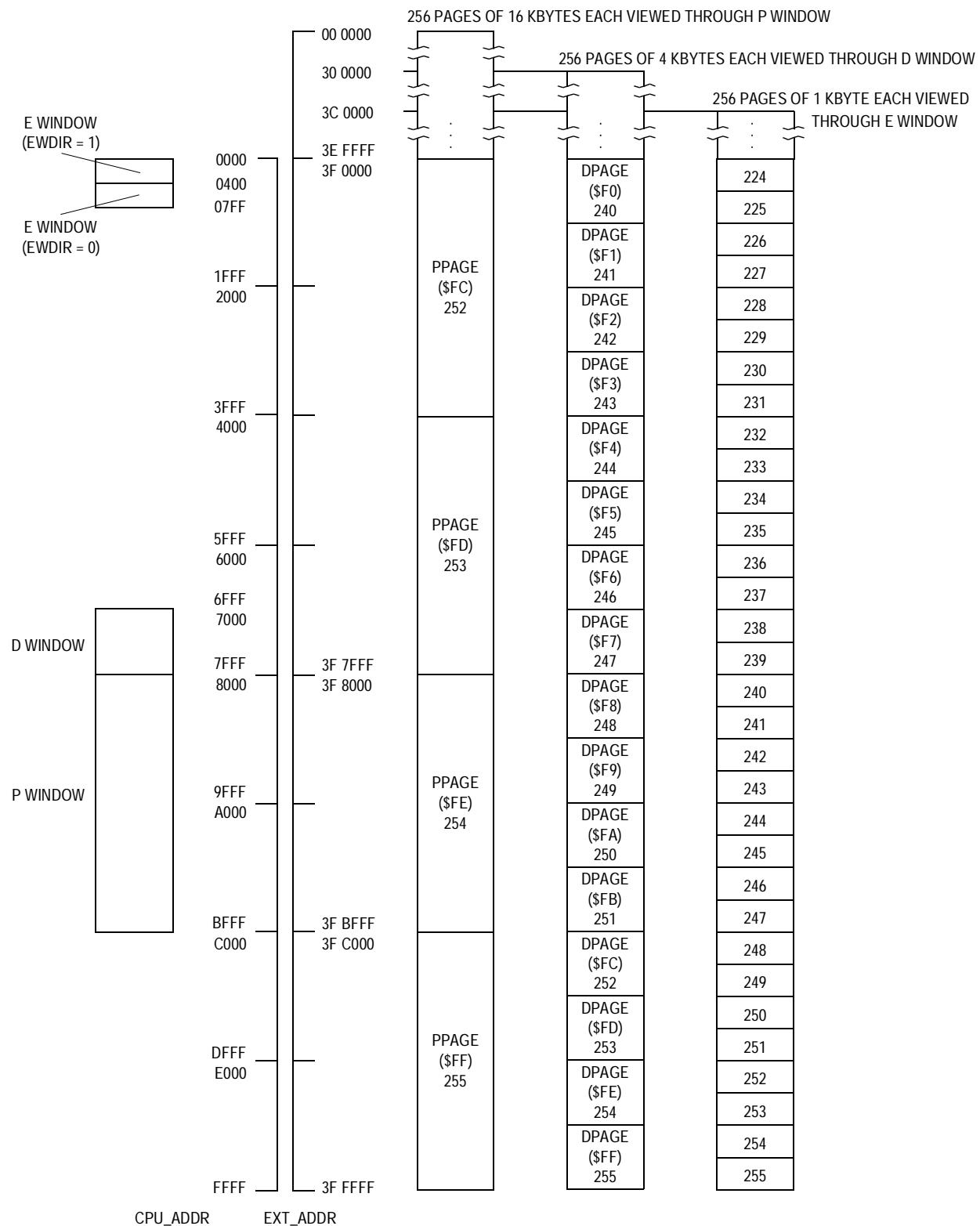
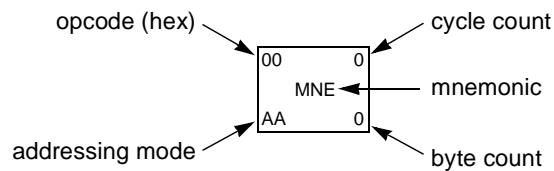


Figure 10-1. Memory Expansion Paging Summary

Refer to this key for the CPU12 opcode map shown in [Table A-7](#).



Addressing mode abbreviations:

- DI — Direct
- EX — Extended
- ID — Indexed
- IH — Inherent
- IM — Immediate
- RL — Relative
- SP — Special

Cycle counts are for single-chip mode with 16-bit internal buses. Stack location (internal or external), external bus width, and operand alignment can affect actual execution time.

Instruction Reference

Table A-7. CPU12 Opcode Map (Sheet 1 of 2)

| | | | | | | | | | | | | | | | |
|---------------------------|-------------------------|-----------------------|-----------------------|---------------------------|-----------------------|--------------------------|----------------------|----------------------|----------------------|--------------------------|-------------------------|-----------------------|-----------------------|---------------------------|-----------------------|
| 00 *5 BGND IH 1 | 10 1 ANDCC IM 2 | 20 3 BRA RL 2 | 30 3 PULX IH 1 | 40 1 NEGA IH 1 | 50 1 NEG B 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 IH 1 | 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 3 MUL IH 1 | 22 3/1 BHI RL 2 | 32 3 PULA IH 1 | 42 1 INCA IH 1 | 52 1 INC B IH 1 | 62 3-6 INC ID 2-4 | 72 4 INC EX 3 | 82 1 SBCA IM 2 | 92 3 SBCA DI 2 | A2 3-6 SBCA ID 2-4 | B2 3 SBCA EX 3 | C2 1 SBCB IM 2 | D2 3 SBCB DI 2 | E2 3-6 SBCB ID 2-4 | F2 3 SBCB EX 3 |
| 03 1 DEY IH 1 | 13 3 EMUL IH 1 | 23 3/1 BLS RL 2 | 33 3 PULB IH 1 | 43 1 DECA IH 1 | 53 1 DEC B 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 1 | 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 2-5 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 ADC B IM 2 | D9 3 ADC B DI 2 | E9 3-6 ADC B ID 2-4 | F9 3 ADC B EX 3 |
| 0A 6 RTC IH 1 | 1A 2 LEAX ID 2-4 | 2A 3/1 BPL RL 2 | 3A 3 PULD IH 1 | 4A 8 CALL EX 4 | 5A 2 STAA DI 2 | 6A 2-5 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 8-10 CALL ID 2-5 | 5B 2 STAB DI 2 | 6B 2-5 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 *9 wavr SP 1 | 4C 4 BSET DI 3 | 5C 2 STD DI 2 | 6C 2-5 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 2-5 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 4-8 BRSET ID 4-6 | 1E 5 BRSET EX 5 | 2E 3/1 BGT RL 2 | 3E *8 WAI IH 1 | 4E 4 BRSET DI 4 | 5E 2 STX DI 2 | 6E 2-5 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 4-8 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 2-5 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 |

Table A-7. CPU12 Opcode Map (Sheet 2 of 2)

| | | | | | | | | | | | | | | | |
|-------------------------|---------------------------|------------------------|-------------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|-----------------------|
| 00 4 MOVW IM-ID 5 | 10 12 IDIV IH 2 | 20 4 LBRA RL 4 | 30 10 TRAP IH 2 | 40 10 TRAP IH 2 | 50 10 TRAP IH 2 | 60 10 TRAP IH 2 | 70 10 TRAP IH 2 | 80 10 TRAP IH 2 | 90 10 TRAP IH 2 | A0 10 TRAP IH 2 | B0 10 TRAP IH 2 | C0 10 TRAP IH 2 | D0 10 TRAP IH 2 | E0 10 TRAP IH 2 | F0 10 TRAP IH 2 |
| 01 5 MOVW EX-ID 5 | 11 12 FDIV IH 2 | 21 3 LBRN RL 4 | 31 10 TRAP IH 2 | 41 10 TRAP IH 2 | 51 10 TRAP IH 2 | 61 10 TRAP IH 2 | 71 10 TRAP IH 2 | 81 10 TRAP IH 2 | 91 10 TRAP IH 2 | A1 10 TRAP IH 2 | B1 10 TRAP IH 2 | C1 10 TRAP IH 2 | D1 10 TRAP IH 2 | E1 10 TRAP IH 2 | F1 10 TRAP IH 2 |
| 02 5 MOVW ID-ID 4 | 12 13 EMACS SP 4 | 22 4/3 LBHI RL 4 | 32 10 TRAP IH 2 | 42 10 TRAP IH 2 | 52 10 TRAP IH 2 | 62 10 TRAP IH 2 | 72 10 TRAP IH 2 | 82 10 TRAP IH 2 | 92 10 TRAP IH 2 | A2 10 TRAP IH 2 | B2 10 TRAP IH 2 | C2 10 TRAP IH 2 | D2 10 TRAP IH 2 | E2 10 TRAP IH 2 | F2 10 TRAP IH 2 |
| 03 5 MOVW IM-EX 6 | 13 3 EMULS IH 2 | 23 4/3 LBLS RL 4 | 33 10 TRAP IH 2 | 43 10 TRAP IH 2 | 53 10 TRAP IH 2 | 63 10 TRAP IH 2 | 73 10 TRAP IH 2 | 83 10 TRAP IH 2 | 93 10 TRAP IH 2 | A3 10 TRAP IH 2 | B3 10 TRAP IH 2 | C3 10 TRAP IH 2 | D3 10 TRAP IH 2 | E3 10 TRAP IH 2 | F3 10 TRAP IH 2 |
| 04 6 MOVW EX-EX 6 | 14 12 EDIVS IH 2 | 24 4/3 LBCC RL 4 | 34 10 TRAP IH 2 | 44 10 TRAP IH 2 | 54 10 TRAP IH 2 | 64 10 TRAP IH 2 | 74 10 TRAP IH 2 | 84 10 TRAP IH 2 | 94 10 TRAP IH 2 | A4 10 TRAP IH 2 | B4 10 TRAP IH 2 | C4 10 TRAP IH 2 | D4 10 TRAP IH 2 | E4 10 TRAP IH 2 | F4 10 TRAP IH 2 |
| 05 5 MOVW ID-EX 5 | 15 12 IDIVS IH 2 | 25 4/3 LBCS RL 4 | 35 10 TRAP IH 2 | 45 10 TRAP IH 2 | 55 10 TRAP IH 2 | 65 10 TRAP IH 2 | 75 10 TRAP IH 2 | 85 10 TRAP IH 2 | 95 10 TRAP IH 2 | A5 10 TRAP IH 2 | B5 10 TRAP IH 2 | C5 10 TRAP IH 2 | D5 10 TRAP IH 2 | E5 10 TRAP IH 2 | F5 10 TRAP IH 2 |
| 06 2 ABA IH 2 | 16 2 SBA IH 2 | 26 4/3 LBNE RL 4 | 36 10 TRAP IH 2 | 46 10 TRAP IH 2 | 56 10 TRAP IH 2 | 66 10 TRAP IH 2 | 76 10 TRAP IH 2 | 86 10 TRAP IH 2 | 96 10 TRAP IH 2 | A6 10 TRAP IH 2 | B6 10 TRAP IH 2 | C6 10 TRAP IH 2 | D6 10 TRAP IH 2 | E6 10 TRAP IH 2 | F6 10 TRAP IH 2 |
| 07 3 DAA IH 2 | 17 2 CBA IH 2 | 27 4/3 LBEQ RL 4 | 37 10 TRAP IH 2 | 47 10 TRAP IH 2 | 57 10 TRAP IH 2 | 67 10 TRAP IH 2 | 77 10 TRAP IH 2 | 87 10 TRAP IH 2 | 97 10 TRAP IH 2 | A7 10 TRAP IH 2 | B7 10 TRAP IH 2 | C7 10 TRAP IH 2 | D7 10 TRAP IH 2 | E7 10 TRAP IH 2 | F7 10 TRAP IH 2 |
| 08 4 MOVB IM-ID 4 | 18 4-7 MAXA ID 3-5 | 28 4/3 LBVC RL 4 | 38 10 TRAP IH 2 | 48 10 TRAP IH 2 | 58 10 TRAP IH 2 | 68 10 TRAP IH 2 | 78 10 TRAP IH 2 | 88 10 TRAP IH 2 | 98 10 TRAP IH 2 | A8 10 TRAP IH 2 | B8 10 TRAP IH 2 | C8 10 TRAP IH 2 | D8 10 TRAP IH 2 | E8 10 TRAP IH 2 | F8 10 TRAP IH 2 |
| 09 5 MOVB EX-ID 5 | 19 4-7 MINA ID 3-5 | 29 4/3 LBVS RL 4 | 39 10 TRAP IH 2 | 49 10 TRAP IH 2 | 59 10 TRAP IH 2 | 69 10 TRAP IH 2 | 79 10 TRAP IH 2 | 89 10 TRAP IH 2 | 99 10 TRAP IH 2 | A9 10 TRAP IH 2 | B9 10 TRAP IH 2 | C9 10 TRAP IH 2 | D9 10 TRAP IH 2 | E9 10 TRAP IH 2 | F9 10 TRAP IH 2 |
| 0A 5 MOVB ID-ID 4 | 1A 4-7 EMAXD ID 3-5 | 2A 4/3 LBPL RL 4 | 3A *3n REV SP 2 | 4A 10 TRAP IH 2 | 5A 10 TRAP IH 2 | 6A 10 TRAP IH 2 | 7A 10 TRAP IH 2 | 8A 10 TRAP IH 2 | 9A 10 TRAP IH 2 | AA 10 TRAP IH 2 | BA 10 TRAP IH 2 | CA 10 TRAP IH 2 | DA 10 TRAP IH 2 | EA 10 TRAP IH 2 | FA 10 TRAP IH 2 |
| 0B 4 MOVB IM-EX 5 | 1B 4-7 EMIND ID 3-5 | 2B 4/3 LBMI RL 4 | 3B *3n REVV 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 *8B WAV SP 2 | 4C 10 TRAP IH 2 | 5C 10 TRAP IH 2 | 6C 10 TRAP IH 2 | 7C 10 TRAP IH 2 | 8C 10 TRAP IH 2 | 9C 10 TRAP IH 2 | AC 10 TRAP IH 2 | BC 10 TRAP IH 2 | CC 10 TRAP IH 2 | DC 10 TRAP IH 2 | EC 10 TRAP IH 2 | FC 10 TRAP IH 2 |
| 0D 5 MOVB ID-EX 5 | 1D 4-7 MINM ID 3-5 | 2D 4/3 LBLT RL 4 | 3D 8 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 *9+5 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 |

* Refer to instruction summary for more information.

‡ The opcode \$04 corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

A.9 Hexadecimal to ASCII Conversion

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.10 Hexadecimal to Decimal Conversion

To convert a hexadecimal number (up to four hexadecimal digits) to decimal, look up the decimal equivalent of each hexadecimal digit in **Table A-9**. The decimal equivalent of the original hexadecimal number is the sum of the weights found in the table for all hexadecimal digits.

Table A-9. Hexadecimal to/from Decimal Conversion

| 15 | | Bit | | 8 | 7 | Bit | | 0 |
|---------------|---------|---------------|---------|---------------|---------|---------------|---------|----|
| 15 | | 12 | 11 | 8 | 7 | 4 | 3 | 0 |
| 4th Hex Digit | | 3rd Hex Digit | | 2nd Hex Digit | | 1st Hex Digit | | |
| Hex | Decimal | Hex | Decimal | Hex | Decimal | Hex | Decimal | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 4,096 | 1 | 256 | 1 | 16 | 1 | 1 | 1 |
| 2 | 8,192 | 2 | 512 | 2 | 32 | 2 | 2 | 2 |
| 3 | 12,288 | 3 | 768 | 3 | 48 | 3 | 3 | 3 |
| 4 | 16,384 | 4 | 1,024 | 4 | 64 | 4 | 4 | 4 |
| 5 | 20,480 | 5 | 1,280 | 5 | 80 | 5 | 5 | 5 |
| 6 | 24,576 | 6 | 1,536 | 6 | 96 | 6 | 6 | 6 |
| 7 | 28,672 | 7 | 1,792 | 7 | 112 | 7 | 7 | 7 |
| 8 | 32,768 | 8 | 2,048 | 8 | 128 | 8 | 8 | 8 |
| 9 | 36,864 | 9 | 2,304 | 9 | 144 | 9 | 9 | 9 |
| A | 40,960 | A | 2,560 | A | 160 | A | 10 | 10 |
| B | 45,056 | B | 2,816 | B | 176 | B | 11 | 11 |
| C | 49,152 | C | 3,072 | C | 192 | C | 12 | 12 |
| D | 53,248 | D | 3,328 | D | 208 | D | 13 | 13 |
| E | 57,344 | E | 3,484 | E | 224 | E | 14 | 14 |
| F | 61,440 | F | 3,840 | F | 240 | F | 15 | 15 |

A.11 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. M68HC11 to M68HC12 Upgrade Path

B.1 Contents

| | | |
|--------|--|-----|
| B.2 | Introduction | 442 |
| B.3 | CPU12 Design Goals | 442 |
| B.4 | Source Code Compatibility | 442 |
| B.5 | Programmer's Model and Stacking | 445 |
| B.6 | True 16-Bit Architecture | 445 |
| B.6.1 | Bus Structures | 446 |
| B.6.2 | Instruction Queue | 446 |
| B.6.3 | Stack Function | 448 |
| B.7 | Improved Indexing | 449 |
| B.7.1 | Constant Offset Indexing | 450 |
| B.7.2 | Auto-Increment Indexing | 451 |
| B.7.3 | Accumulator Offset Indexing | 452 |
| B.7.4 | Indirect Indexing | 453 |
| B.8 | Improved Performance | 453 |
| B.8.1 | Reduced Cycle Counts | 453 |
| B.8.2 | Fast Math | 454 |
| B.8.3 | Code Size Reduction | 455 |
| B.9 | Additional Functions | 456 |
| B.9.1 | Memory-to-Memory Moves | 458 |
| B.9.2 | Universal Transfer and Exchange | 459 |
| B.9.3 | Loop Construct | 459 |
| B.9.4 | Long Branches | 459 |
| B.9.5 | Minimum and Maximum Instructions | 460 |
| B.9.6 | Fuzzy Logic Support | 461 |
| B.9.7 | Table Lookup and Interpolation | 461 |
| B.9.8 | Extended Bit Manipulation | 462 |
| B.9.9 | Push and Pull D and CCR | 462 |
| B.9.10 | Compare SP | 462 |
| B.9.11 | Support for Memory Expansion | 462 |

B.2 Introduction

This appendix discusses similarities and differences between the CPU12 and the M68HC11 CPU. In general, the CPU12 is a proper superset of the M68HC11. Significant changes have been made to improve the efficiency and capabilities of the CPU12 without eliminating compatibility and familiarity for the large community of M68HC11 programmers.

B.3 CPU12 Design Goals

The primary goals of the CPU12 design were:

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

B.4 Source Code Compatibility

Every M68HC11 instruction mnemonic and source code statement can be assembled directly with a CPU12 assembler with no modifications.

The CPU12 supports all M68HC11 addressing modes and includes several new variations of indexed addressing mode. CPU12 instructions affect condition code bits in the same way as M68HC11 instructions.

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

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

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

Table B-1. Translated M68HC11 Mnemonics

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

M68HC11 to M68HC12 Upgrade Path

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

The CPU12 has a 2-page opcode map, rather than the 4-page M68HC11 map. This is largely due to redesign of the indexed addressing modes. Most of pages 2, 3, and 4 of the M68HC11 opcode map are required because Y-indexed instructions use different opcodes than X-indexed instructions. Approximately two-thirds of the M68HC11 page 1 opcodes are unchanged in CPU12, and some M68HC11 opcodes have been moved to page 1 of the CPU12 opcode map. Object code for each of the moved instructions is one byte smaller than object code for the equivalent M68HC11 instruction. **Table B-2** shows instructions that assemble to one byte less object code on the CPU12.

Table B-2. Instructions with Smaller Object Code

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

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

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

It seems useful to mention the results of size comparisons done on C programs. A C program compiled for the CPU12 is about 30 percent smaller than the same program compiled for the M68HC11. The savings are largely due to better indexing.

B.5 Programmer's Model and Stacking

The CPU12 programming model and stacking order are identical to those of the M68HC11.

B.6 True 16-Bit Architecture

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

B.6.1 Bus Structures

The CPU12 is a 16-bit processor with 16-bit data paths. Typical M68HC12 devices have internal and external 16-bit data paths, but some derivatives incorporate operating modes that allow for an 8-bit data bus, so that a system can be built with low-cost 8-bit program memory. M68HC12 MCUs include an on-chip integration module that manages the external bus interface. When the CPU makes a 16-bit access to a resource that is served by an 8-bit bus, the integration module performs two 8-bit accesses, freezes the CPU clocks for part of the sequence, and assembles the data into a 16-bit word. As far as the CPU is concerned, there is no difference between this access and a 16-bit access to an internal resource via the 16-bit data bus. This is similar to the way an M68HC11 can stretch clock cycles to accommodate slow peripherals.

B.6.2 Instruction Queue

The CPU12 has a 2-word instruction queue and a 16-bit holding buffer, which sometimes acts as a third word for queueing program information. All program information is fetched from memory as aligned 16-bit words, even though there is no requirement for instructions to begin or end on even word boundaries. There is no penalty for misaligned instructions. If a program begins on an odd boundary (if the reset vector is an odd address), program information is fetched to fill the instruction queue, beginning with the aligned word at the next address below the misaligned reset vector. The instruction queue logic starts execution with the opcode in the low-order half of this word.

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

The CPU is not aware of instruction alignment. The queue logic includes a multiplexer that sorts out the information in the queue to present the opcode and the next two bytes of information as CPU inputs. The

multiplexer determines whether the opcode is in the even or odd half of the word at the head of the queue. Alignment status is also available to the ALU for address calculations. The execution sequence for all instructions is independent of the alignment of the instruction.

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

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

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

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

B.6.3 Stack Function

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

The CPU12 has a “last-used” stack rather than a “next-available” stack like the M68HC11 CPU. That is, the stack pointer points to the last 16-bit stack address used, rather than to the address of the next available stack location. This generally has very little effect, because it is very unusual to access stacked information using absolute addressing. The change allows a 16-bit word of data to be removed from the stack without changing the value of the SP twice.

To illustrate, consider the operation of a PULX instruction. With the next-available M68HC11 stack, if the SP = \$01F0 when execution begins, the sequence of operations is: SP = SP + 1; load X from \$01F1:01F2; SP = SP + 1; and the SP ends up at \$01F2. With the last-used CPU12 stack, if the SP = \$01F0 when execution begins, the sequence is: load X from \$01F0:01F1; SP = SP + 2; and the SP again ends up at \$01F2. The second sequence requires one less stack pointer adjustment.

The stack pointer change also affects operation of the TSX and TXS instructions. In the M68HC11, TSX increments the SP by one during the transfer. This adjustment causes the X index to point to the last stack location used. The TXS instruction operates similarly, except that it decrements the SP by one during the transfer. CPU12 TSX and TXS instructions are ordinary transfers — the CPU12 stack requires no adjustment.

For ordinary use of the stack, such as pushes, pulls, and even manipulations involving TSX and TXS, there are no differences in the way the M68HC11 and the CPU12 stacks look to a programmer. However, the stack change can affect a program algorithm in two subtle ways.

The LDS #\$xxxx instruction is normally used to initialize the stack pointer at the start of a program. In the M68HC11, the address specified in the LDS instruction is the first stack location used. In the CPU12, however, the first stack location used is one address lower than the address specified in the LDS instruction. Since the stack builds downward, M68HC11 programs reassembled for the CPU12 operate normally, but the program stack is one physical address lower in memory.

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

B.7 Improved Indexing

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

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

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

Major improvements that result from this new approach are:

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

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

Table A-2 and **Table A-3** summarize M68HC12 indexed addressing mode capabilities. **Table A-5** shows how the postbyte is encoded.

B.7.1 Constant Offset Indexing

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

The most common constant offset is 0. Offsets of 1, 2, 3, 4 are used fairly often, but with less frequency than 0.

The 5-bit constant offset variation covers the most frequent indexing requirements by including the offset in the postbyte. This reduces a load

accumulator indexed instruction to two bytes of object code, and matches the object code size of the smallest M68HC11 indexed instructions, which can only use X as the index register. The CPU12 can use X, Y, SP, or PC as the index reference with no additional object code size cost.

The signed 9-bit constant offset indexing mode covers the same positive range as the M68HC11 8-bit unsigned offset. The size was increased to nine bits with the sign bit (ninth bit) included in the postbyte, and the remaining 8 bits of the offset in a single extension byte.

The 16-bit constant offset indexing mode allows indexed access to the entire normal 64-Kbyte address space. Since the address consists of 16 bits, the 16-bit offset can be regarded as a signed (-32,768 to +32,767) or unsigned (0 to 65,535) value. In 16-bit constant offset mode, the offset is supplied in two extension bytes after the opcode and postbyte.

B.7.2 Auto-Increment Indexing

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

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

| | | | |
|--------|---------|------|---------|
| 18A600 | LDAA0,Y | A671 | LDA2,Y+ |
| 1808 | INY | | |
| 1808 | INY | | |

The M68HC11 object code requires seven bytes, while the CPU12 requires only two bytes to accomplish the same functions. Three bytes of M68HC11 code were due to the page prebyte for each Y-related instruction (\$18). CPU12 post-increment indexing capability allowed the two INY instructions to be absorbed into the LDAA indexed instruction. The replacement code is not identical to the original 3-instruction sequence because the Z condition code bit is affected by the M68HC11 INY instructions, while the Z bit in the CPU12 would be determined by the value loaded into A.

B.7.3 Accumulator Offset Indexing

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

| | | | |
|------------------------------|--|------------------------|--|
| C605 CE1000 3A A600 | LDAB#\$5[2] LOOPLDX#\$1000[3] ABX[3] LDAA0,X[4] | C605 CE1000 A6E5 | LDAB#\$5[1] LDX#\$1000[2] LOOPLDAAB,X[3] |
| 5A 26F7 | DEC B[2] BNELOOP[3] | 0431FB | DBNEB,LOOP[3] |

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

B.7.4 Indirect Indexing

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

B.8 Improved Performance

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

B.8.1 Reduced Cycle Counts

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

B.8.2 Fast Math

The CPU12 has some of the fastest math ever designed into a Motorola general-purpose MCU. Much of the speed is due to a 20-bit ALU that can perform two smaller operations simultaneously. The ALU can also perform two operations in a single bus cycle in certain cases.

Table B-3 compares the speed of CPU12 and M68HC11 math instructions. The CPU12 requires fewer cycles to perform an operation, and the cycle time is half that of the M68HC11.

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

Table B-3. Comparison of Math Instruction Speeds

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

B.8.3 Code Size Reduction

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

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

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

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

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

B.9 Additional Functions

The CPU12 incorporates a number of new instructions that provide added functionality and code efficiency. Among other capabilities, these new instructions allow efficient processing for fuzzy logic applications and support subroutine processing in extended memory beyond the standard 64-Kbyte address map for M68HC12 devices incorporating this feature. **Table B-4** is a summary of these new instructions. Subsequent paragraphs discuss significant enhancements.

Table B-4. New M68HC12 Instructions (Sheet 1 of 3)

| Mnemonic | Addressing Modes | Brief Functional Description |
|----------|--|---|
| ANDCC | Immediate | AND CCR with mask (replaces CLC, CLI, and CLV) |
| BCLR | Extended | Bit(s) clear (added extended mode) |
| BGND | Inherent | Enter background debug mode, if enabled |
| BRCLR | Extended | Branch if bit(s) clear (added extended mode) |
| BRSET | Extended | Branch if bit(s) set (added extended mode) |
| BSET | Extended | Bit(s) set (added extended mode) |
| CALL | Extended, indexed | Similar to JSR except also stacks PPAGE value with RTC instruction, allows easy access to >64-Kbyte space |
| CPS | Immediate, direct, extended, and indexed | Compare stack pointer |
| DBNE | Relative | Decrement and branch if equal to zero (looping primitive) |
| DBEQ | Relative | Decrement and branch if not equal to zero (looping primitive) |
| EDIV | Inherent | Extended divide Y:D/X = Y(Q) and D(R) (unsigned) |
| EDIVS | Inherent | Extended divide Y:D/X = Y(Q) and D(R) (signed) |
| EMACS | Special | Multiply and accumulate $16 \times 16 \Rightarrow 32$ (signed) |
| EMAXD | Indexed | Maximum of two unsigned 16-bit values |
| EMAXM | Indexed | Maximum of two unsigned 16-bit values |
| EMIND | Indexed | Minimum of two unsigned 16-bit values |
| EMINM | Indexed | Minimum of two unsigned 16-bit values |
| EMUL | Special | Extended multiply $16 \times 16 \Rightarrow 32$; M(idx) * D \Rightarrow Y:D |
| EMULS | Special | Extended multiply $16 \times 16 \Rightarrow 32$ (signed); M(idx) * D \Rightarrow Y:D |
| ETBL | Special | Table lookup and interpolate (16-bit entries) |
| EXG | Inherent | Exchange register contents |

Table B-4. New M68HC12 Instructions (Sheet 2 of 3)

| Mnemonic | Addressing Modes | Brief Functional Description |
|----------|--|--|
| IBEQ | Relative | Increment and branch if equal to zero (looping primitive) |
| IBNE | Relative | Increment and branch if not equal to zero (looping primitive) |
| IDIVS | Inherent | Signed integer divide D/X \Rightarrow X(Q) and D(R) (signed) |
| LBCC | Relative | Long branch if carry clear (same as LBHS) |
| LBCS | Relative | Long branch if carry set (same as LBLO) |
| LBEQ | Relative | Long branch if equal (Z=1) |
| LBGE | Relative | Long branch if greater than or equal to zero |
| LBGT | Relative | Long branch if greater than zero |
| LBHI | Relative | Long branch if higher |
| LBHS | Relative | Long branch if higher or same (same as LBCC) |
| LBLE | Relative | Long branch if less than or equal to zero |
| LBLO | Relative | Long branch if lower (same as LBCS) |
| LBLS | Relative | Long branch if lower or same |
| LBLT | Relative | Long branch if less than zero |
| LBMI | Relative | Long branch if minus |
| LBNE | Relative | Long branch if not equal to zero |
| LBPL | Relative | Long branch if plus |
| LBRA | Relative | Long branch always |
| LBRN | Relative | Long branch never |
| LBVC | Relative | Long branch if overflow clear |
| LBVS | Relative | Long branch if overflow set |
| LEAS | Indexed | Load stack pointer with effective address |
| LEAX | Indexed | Load X index register with effective address |
| LEAY | Indexed | Load Y index register with effective address |
| MAXA | Indexed | Maximum of two unsigned 8-bit values |
| MAXM | Indexed | Maximum of two unsigned 8-bit values |
| MEM | Special | Determine grade of fuzzy membership |
| MINA | Indexed | Minimum of two unsigned 8-bit values |
| MINM | Indexed | Minimum of two unsigned 8-bit values |
| MOVB(W) | Combinations of immediate, extended, and indexed | Move data from one memory location to another |

Table B-4. New M68HC12 Instructions (Sheet 3 of 3)

| Mnemonic | Addressing Modes | Brief Functional Description |
|----------|------------------|--|
| ORCC | Immediate | OR CCR with mask (replaces SEC, SEI, and SEV) |
| PSHC | Inherent | Push CCR onto stack |
| PSHD | Inherent | Push double accumulator onto stack |
| PULC | Inherent | Pull CCR contents from stack |
| PULD | Inherent | Pull double accumulator from stack |
| REV | Special | Fuzzy logic rule evaluation |
| REVVW | Special | Fuzzy logic rule evaluation with weights |
| RTC | Inherent | Restore program page and return address from stack used with CALL instruction, allows easy access to >64-Kbyte space |
| SEX | Inherent | Sign extend 8-bit register into 16-bit register |
| TBEQ | Relative | Test and branch if equal to zero (looping primitive) |
| TBL | Inherent | Table lookup and interpolate (8-bit entries) |
| TBNE | Relative | Test register and branch if not equal to zero (looping primitive) |
| TFR | Inherent | Transfer register contents to another register |
| WAV | Special | Weighted average (fuzzy logic support) |

B.9.1 Memory-to-Memory Moves

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

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

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

B.9.2 Universal Transfer and Exchange

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

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

B.9.3 Loop Construct

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

B.9.4 Long Branches

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

B.9.5 Minimum and Maximum Instructions

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

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

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

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

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

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

The “,PCR” notation is also new for the CPU12. This notation indicates the programmer wants an appropriate offset from the PC reference to the memory location (HILIMIT or LOWLIMIT in this example), and then

to assemble this instruction into a PC-relative indexed MIN or MAX instruction.

B.9.6 Fuzzy Logic Support

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

B.9.7 Table Lookup and Interpolation

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

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

B.9.8 Extended Bit Manipulation

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

B.9.9 Push and Pull D and CCR

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

B.9.10 Compare SP

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

B.9.11 Support for Memory Expansion

Bank switching is a common method of expanding memory beyond the 64-Kbyte limit of a CPU with a 64-Kbyte address space, but there are some known difficulties associated with bank switching. One problem is that interrupts cannot take place during the bank switching operation. This increases worst case interrupt latency and requires extra programming space and execution time.

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

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

M68HC11 to M68HC12 Upgrade Path

Appendix C. High-Level Language Support

C.1 Contents

| | | |
|-------|---|-----|
| C.2 | Introduction | 465 |
| C.3 | Data Types | 466 |
| C.4 | Parameters and Variables | 466 |
| C.4.1 | Register Pushes and Pulls | 467 |
| C.4.2 | Allocating and Deallocating Stack Space | 467 |
| C.4.3 | Frame Pointer | 468 |
| C.5 | Increment and Decrement Operators | 469 |
| C.6 | Higher Math Functions | 469 |
| C.7 | Conditional If Constructs | 470 |
| C.8 | Case and Switch Statements | 470 |
| C.9 | Pointers | 470 |
| C.10 | Function Calls | 471 |
| C.11 | Instruction Set Orthogonality | 472 |

C.2 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.

C.3 Data Types

The CPU12 supports the bit-sized data type with bit manipulation instructions which are available in extended, direct, and indexed variations. The char data type is a simple 8-bit value that is commonly used to specify variables in a small microcontroller system because it requires less memory space than a 16-bit integer (provided the variable has a range small enough to fit into eight bits). The 16-bit CPU12 can easily handle 16-bit integer types and the available set of conditional branches (including long branches) allow branching based on signed or unsigned arithmetic results. Some of the higher math functions allow for division and multiplication involving 32-bit values, although it is somewhat less common to use such long values in a microcontroller system.

The CPU12 has special sign extension instructions to allow easy type-casting from smaller data types to larger ones, such as from char to integer. This sign extension is automatically performed when an 8-bit value is transferred to a 16-bit register.

C.4 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 all CPU registers, stack pointer based indexing should be allowed, and that direct arithmetic manipulation of the stack pointer value should be allowed. The CPU12 instruction set provided for all of these needs with improved indexed addressing, the addition of an LEAS instruction, and the addition of push and pull instructions for the D accumulator and the CCR.

C.4.1 Register Pushes and Pulls

The M68HC11 has push and pull instructions for A, B, X, and Y, but requires separate 8-bit pushes and pulls of accumulators A and B to stack or unstack the 16-bit D accumulator (the concatenated combination of A:B). The PSHD and PULD instructions allow directly stacking the D accumulator in the expected 16-bit order.

Adding PSHC and PULC improved orthogonality by completing the set of stacking instructions so that any of the CPU registers can be pushed or pulled. These instructions are also useful for preserving the CCR value during a function call subroutine.

C.4.2 Allocating and Deallocating Stack Space

The LEAS instruction can be used to allocate or deallocate space on the stack for temporary variables:

```
LEAS    -10,S    ;Allocate space for 5 16-bit integers
LEAS    10,S     ;Deallocate space for 5 16-bit ints
```

The (de)allocation can even be combined with a register push or pull as in this example:

```
LDX    8,S+    ;Load return value and deallocate
```

X is loaded with the 16-bit integer value at the top of the stack, and the stack pointer is adjusted up by eight to deallocate space for eight bytes worth of temporary storage. Post-increment indexed addressing is used in this example, but all four combinations of pre/post increment/decrement are available (offsets from -8 to +8 inclusive, from X, Y, or SP). This form of indexing can often be used to get an index (or stack pointer) adjustment for free during an indexed operation (the instruction requires no more code space or cycles than a zero-offset indexed instruction).

C.4.3 Frame Pointer

In the C language, it is common to have a frame pointer in addition to the CPU stack pointer. The frame is an area of memory within the system stack which is used for parameters and local storage of variables used within a function subroutine. The following is a description of how a frame pointer can be set up and used.

First, parameters (typically values in CPU registers) are pushed onto the system stack prior to using a JSR or CALL to get to the function subroutine. At the beginning of the called subroutine, the frame pointer of the calling program is pushed onto the stack. Typically, an index register, such as X, is used as the frame pointer, so a PSHX instruction would save the frame pointer from the calling program.

Next, the called subroutine establishes a new frame pointer by executing a TFR S,X. Space is allocated for local variables by executing an LEAS –n,S, where n is the number of bytes needed for local variables.

Notice that parameters are at positive offsets from the frame pointer while locals are at negative offsets. In the M68HC11, the indexed addressing mode uses only positive offsets, so the frame pointer always points to the lowest address of any parameter or local. After the function subroutine finishes, calculations are required to restore the stack pointer to the mid-frame position between the locals and the parameters before returning to the calling program. The CPU12 only requires execution of TFR X,S to deallocate the local storage and return.

The concept of a frame pointer is supported in the CPU12 through a combination of improved indexed addressing, universal transfer/exchange, and the LEA instruction. These instructions work together to achieve more efficient handling of frame pointers. It is important to consider the complete instruction set as a complex system with subtle interrelationships rather than simply examining individual instructions when trying to improve an instruction set. Adding or removing a single instruction can have unexpected consequences.

C.5 Increment and Decrement Operators

In C, the notation `++ i` or `i--` is often used to form loop counters. Within limited constraints, the CPU12 loop primitives can be used to speed up the loop count and branch function.

The CPU12 includes a set of six basic loop control instructions which decrement, increment, or test a loop count register, and then branch if it is either equal to zero or not equal to zero. The loop count register can be A, B, D, X, Y, or SP. A or B could be used if the loop count fits in an 8-bit char variable; the other choices are all 16-bit registers. The relative offset for the loop branch is a 9-bit signed value, so these instructions can be used with loops as long as 256 bytes.

In some cases, the pre- or post-increment operation can be combined with an indexed instruction to eliminate the cost of the increment operation. This is typically done by post-compile optimization because the indexed instruction that could absorb the increment/decrement operation may not be apparent at compile time.

C.6 Higher Math Functions

In the CPU12, subtle characteristics of higher math operations such as IDIVS and EMUL are arranged so a compiler can handle inputs and outputs more efficiently.

The most apparent case is the IDIVS instruction, which divides two 16-bit signed numbers to produce a 16-bit result. While the same function can be accomplished with the EDIVS instruction (a 32 by 16 divide), doing so is much less efficient because extra steps are required to prepare inputs to the EDIVS, and because EDIVS uses the Y index register. EDIVS uses a 32-bit signed numerator and the C compiler would typically want to use a 16-bit value (the size of an integer data type). The 16-bit C value would need to be sign-extended into the upper 16 bits of the 32-bit EDIVS numerator before the divide operation.

Operand size is also a potential problem in the extended multiply operations but the difficulty can be minimized by putting the results in CPU registers. Having higher precision math instructions is not necessarily a requirement for supporting high-level language because these functions can be performed as library functions. However, if an

application requires these functions, the code is much more efficient if the MCU can use native instructions instead of relatively large, slow routines.

C.7 Conditional If Constructs

In the CPU12 instruction set, most arithmetic and data manipulation instructions automatically update the condition code register, unlike other architectures that only change condition codes during a few specific compare instructions. The CPU12 includes branch instructions that perform conditional branching based on the state of the indicators in the condition codes register. Short branches use a single byte relative offset that allows branching to a destination within about ± 128 locations from the branch. Long branches use a 16-bit relative offset that allows conditional branching to any location in the 64-Kbyte map.

C.8 Case and Switch Statements

Case and switch statements (and computed GOTOs) can use PC-relative indirect addressing to determine which path to take. Depending upon the situation, cases can use either the constant offset variation or the accumulator D offset variation of indirect indexed addressing.

C.9 Pointers

The CPU12 supports pointers by allowing direct arithmetic operations on the 16-bit index registers (LEAS, LEAX, and LEAY instructions) and by allowing indexed indirect addressing modes.

C.10 Function Calls

Bank switching is a fairly common way of adapting a CPU with a 16-bit address bus to accommodate more than 64 Kbytes of program memory space. One of the most significant drawbacks of this technique has been the requirement to mask (disable) interrupts while the bank page value was being changed. Another problem is that the physical location of the bank page register can change from one MCU derivative to another (or even due to a change to mapping controls by a user program). In these situations, an operating system program has to keep track of the physical location of the page register. The CPU12 addresses both of these problems with the uninterruptible CALL and return-from-call (RTC) instructions.

The CALL instruction is similar to a JSR instruction, except that the programmer supplies a destination page value as part of the instruction. When CALL executes, the old page value is saved on the stack and the new page value is written to the bank page register. Since the CALL instruction is uninterruptible, this eliminates the need to separately mask off interrupts during the context switch.

The CPU12 has dedicated signal lines that allow the CPU to access the bank page register without having to use an address in the normal 64-Kbyte address space. This eliminates the need for the program to know where the page register is physically located.

The RTC instruction is similar to the RTS instruction, except that RTC uses the byte of information that was saved on the stack by the corresponding CALL instruction to restore the bank page register to its old value. Although a CALL/RTC pair can be used to access any function subroutine regardless of the location of the called routine (on the current bank page or a different page), it is most efficient to access some subroutines with JSR/RTS instructions when the called subroutine is on the current page or in an area of memory that is always visible in the 64-Kbyte map regardless of the bank page selection.

Push and pull instructions can be used to stack some or all the CPU registers during a function call. The CPU12 can push and pull any of the CPU registers A, B, CCR, D, X, Y, or SP.

C.11 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 makes it possible to implement compilers more efficiently, because operation is more consistent, and fewer special cases must be handled.

Index

A

| | |
|-----------------------------------|--------|
| ADDR mnemonic | 26 |
| Addressing modes | 38 |
| Direct | 40 |
| Extended | 41 |
| Immediate | 39 |
| Indexed | 29, 43 |
| Inherent | 39 |
| Memory expansion | 405 |
| Relative | 41 |
| ANDA instruction | 118 |
| ANDB instruction | 119 |
| ANDCC instruction | 120 |
| ASL instruction | 121 |
| ASLA instruction | 122 |
| ASLB instruction | 123 |
| ASLD instruction | 124 |
| ASR instruction | 125 |
| ASRA instruction | 126 |
| ASRB instruction | 127 |
| Asserted | 26 |
| Automatic indexing | 47 |
| Automatic program stack | 29 |

B

| | |
|-------------------------------------|---------------|
| Background debugging mode | 96, 344 |
| BKGD pin | 345, 346, 347 |
| Commands | 347, 348 |
| Enabling and disabling | 344 |
| Instruction | 96, 133, 344 |
| Registers | 350 |
| ROM | 344 |
| Serial interface | 345, 346, 347 |
| Base index register | 45–50 |
| BCC instruction | 128 |
| BCLR instruction | 129 |
| BCS instruction | 130 |

| | |
|--|--|
| BEQ instruction | 131 |
| BGE instruction | 132 |
| BGND instruction. | 96, 133, 344 |
| BGT instruction | 134 |
| BHI instruction | 135 |
| BHS instruction | 136 |
| Binary-coded decimal instructions. | 70, 110, 113, 114, 115, 116, 172, 173 |
| Bit manipulation instructions | 76, 129, 150, 462, 466 |
| Mask operand | 52, 129, 150 |
| Multiple addressing modes | 52, 129, 150 |
| Bit test instructions | 76, 137, 138, 466 |
| BITA instruction | 137 |
| BITB instruction | 138 |
| Bit-condition branches. | 86, 147, 149 |
| BKGD pin. | 345, 346, 347 |
| BLE instruction | 139 |
| BLO instruction | 140 |
| BLS instruction | 141 |
| BLT instruction. | 142 |
| BMI instruction. | 143 |
| BNE instruction | 144 |
| Boolean logic instructions | 73 |
| AND | 118, 119, 120 |
| Complement. | 165, 166, 167 |
| Exclusive OR | 191, 192 |
| Inclusive OR. | 256, 257, 258 |
| Negate | 252, 253, 254 |
| BPL instruction | 145 |
| BRA instruction | 146 |
| Branch instructions | 41, 60, 61, 62, 83, 470 |
| Bit-condition | 60, 61, 62, 86, 147, 149 |
| Long | 60, 61, 62, 85, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 459 |
| Loop primitive. | 62, 87, 174, 175, 196, 197, 306, 308 |
| Offset values | 84, 85, 86, 87 |

| | |
|----------------------------------|--|
| Offsets | 42 |
| Short. | 60, 61, 62, 84, 128, 130, 131, 132, 134, 135, 136, 139, 140, 141, 142, 143, 144, 145, 146, 148, 152, 153 |
| Signed | 83, 132, 134, 139, 142, 211, 212, 215, 218 |
| Simple | 83, 128, 130, 131, 143, 144, 145, 152, 153, 208, 209, 210, 219, 220, 221, 224, 225 |
| Subroutine | 88, 151 |
| Taken/not-taken cases. | 60, 109 |
| Unary | 83, 146, 148, 222, 223 |
| Unsigned | 83, 135, 136, 140, 141, 213, 214, 216, 217 |
| BRCLR instruction. | 147 |
| BRN instruction | 148 |
| BRSET instruction. | 149 |
| BSET instruction | 150 |
| BSR instruction | 59, 151 |
| Bus cycles | 104 |
| Bus structure | 446 |
| BVC instruction | 152 |
| BVS instruction | 153 |
| Byte moves | 249 |
| Byte order in memory | 35 |
| Byte-sized instructions. | 61 |

C

| | |
|------------------------------------|---|
| C status bit. | 34, 121, 122, 123, 124, 125, 126, 127, 128, 130, 135, 136, 140, 141, 157, 172, 176, 177, 178, 183, 185, 186, 187, 188, 189, 190, 199, 200, 201, 202, 208, 209, 213, 214, 216, 217, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 247, 248, 273, 276, 277, 278, 279, 280, 281, 285, 286, 287, 288, 299, 300, 301 |
| CALL instruction | 52, 59, 88, 154, 155, 397, 398, 400, 462, 471 |
| Case statements | 470 |
| CBA instruction | 156 |
| Changes in execution flow. | 58, 59, 60, 62, 206, 207, 282, 283, 284, 302, 324, 326, 328, 329, 330, 331 |
| CLC instruction | 157 |

| | |
|---|--|
| Clear instructions | 74, 159, 160, 161 |
| Cleared | 26 |
| CLI instruction | 158 |
| Clock monitor reset | 327 |
| CLR instruction | 159 |
| CLRA instruction | 160 |
| CLRB instruction | 161 |
| CLV instruction | 162 |
| CMPA instruction | 163 |
| CMPB instruction | 164 |
| Code size. | 455 |
| COM instruction. | 165 |
| COMA instruction | 166 |
| COMB instruction | 167 |
| Compare instructions. | 72, 156, 163, 164, 168, 169, 170, 171 |
| Complement instructions. | 74, 165, 166, 167 |
| Computer operating properly monitor | 327 |
| Condition codes instructions | 94, 120, 157, 158, 162, 258, 261 , 267, 288, 289, 290, 304, 309, 310, 462 |
| Condition codes register | 28, 30, 120, 157, 158, 162, 194, 232, 258, 261, 267, 283, 289, 290, 291, 304, 309, 310, 312, 313, 314, 470 |
| C status bit. | 34, 121, 122, 123, 124, 125, 126, 127, 128, 130, 135, 136, 140, 141, 157, 172, 176, 177, 178, 183, 185, 186, 187, 188, 189, 190, 199, 200, 201, 202, 208, 209, 213, 214, 216, 217, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 247, 248, 273, 276, 277, 278, 279, 280, 281, 285, 286, 287, 288, 299, 300, 301 |
| H status bit. | 32, 110, 113, 114, 115, 116, 172 |
| I mask bit | 33, 120, 158, 289, 302, 311, 319, 326, 329 |
| Manipulation. | 94, 120, 157, 158, 162, 258, 288, 289, 290, 304, 310 |
| N status bit. | 33, 143, 145, 219, 221 |
| S control bit | 31, 295 |
| Stacking | 261, 267 |
| V status bit | 34, 152, 153, 162, 224, 225, 271, 272, 273, 274, 290 |

| | |
|---|---|
| X mask bit | 32, 194, 267, 283, 295, 304, 309, 319, 325, 328 |
| Z status bit | 33, 131, 144, 185, 186, 187, 188, 204, 205, 210, 220, 243, 244, 247, 248 |
| Conditional 16-bit read cycle | 108, 410 |
| Conditional 8-bit read cycle | 108, 410 |
| Conditional 8-bit write cycle | 108, 410 |
| Conserving power | 95, 295 |
| Constant indirect indexed addressing mode | 47 |
| Constant offset indexed addressing mode | 45, 46 |
| COP reset | 327 |
| CPD instruction | 168 |
| CPS instruction | 169 |
| CPU wait | 319 |
| CPX instruction | 170 |
| CPY instruction | 171 |
| Cycle code letters | 104 |
| Cycle counts | 453 |
| Cycle-by-cycle operation | 104 |

D

| | |
|----------------------------------|----------------------------------|
| DAA instruction | 172, 173 |
| DATA mnemonic | 26 |
| Data types | 34 |
| DBEQ instruction | 174, 431 |
| DBNE instruction | 175, 431 |
| DEC instruction | 176 |
| DECA instruction | 177 |
| DECB instruction | 178 |
| Decrement instructions | 71, 176, 177, 178, 179, 180, 181 |
| Defuzzification | 363, 383, 384, 385, 387 |
| DES instruction | 179 |
| DEX instruction | 180 |
| DEY instruction | 181 |
| Direct addressing mode | 40 |

| | |
|---------------------------------|----------|
| Division instructions | 75 |
| 16-bit fractional | 195 |
| 16-bit integer | 198, 199 |
| 32-bit extended | 182, 183 |
| Division instructions | 469 |

E

| | |
|--|-----------------------------|
| EDIV instruction | 182 |
| EDIVS instruction | 183 |
| Effective address | 39, 45, 232, 233, 234 |
| EMACS instruction | 82, 184 |
| EMAXD | 185 |
| EMAXD instruction | 185 |
| EMAXM instruction | 186, 356 |
| EMIND instruction | 187, 356 |
| EMINM instruction | 188 |
| EMUL instruction | 189 |
| EMULS instruction | 190 |
| Enabling maskable interrupts | 33 |
| EORA instruction | 191 |
| EORB instruction | 192 |
| ETBL instruction | 82, 193, 356 |
| Even bytes | 35 |
| Exceptions | 58, 324 |
| Interrupts | 328 |
| Maskable interrupts | 329, 330 |
| Non-maskable interrupts | 328 |
| Priority | 325 |
| Processing flow | 331 |
| Resets | 324, 326, 327 |
| Software interrupts | 89, 302, 331 |
| Unimplemented opcode trap | 324, 326, 331 |
| Vectors | 324, 333 |
| Exchange instructions | 67, 194, 321, 322, 455, 459 |
| Postbyte encoding | 430 |

| | |
|---|--------------------------------------|
| Execution cycles | 104 |
| Conditional 16-bit read | 108, 410 |
| Conditional 8-bit read | 108, 410 |
| Conditional 8-bit write | 108, 410 |
| Free | 104, 410 |
| Optional | 61, 106, 410 |
| Program word access | 106, 410 |
| Read 16-bit data | 107, 410 |
| Read 8-bit data | 106, 410 |
| Read indirect pointer | 105, 410 |
| Read indirect PPAGE value | 105, 410 |
| Read PPAGE | 105, 410 |
| Stack 16-bit data | 107, 410 |
| Stack 8-bit data | 107, 410 |
| Unstack 16-bit data | 108, 410 |
| Unstack 8-bit data | 107, 410 |
| Vector fetch | 108, 410 |
| Write 16-bit data | 107, 410 |
| Write 8-bit data | 107, 410 |
| Write PPAGE | 105, 410 |
| Execution time | 104 |
| EXG instruction | 194 |
| Expanded memory | 52, 59, 396, 462, 471 |
| Addressing modes | 52, 401, 403, 404 |
| Bank switching | 52, 396, 397, 399, 401, 403, 404 |
| Instructions | 52, 88, 154, 155, 282, 397, 398, 400 |
| Overlay windows | 397, 399, 401, 403, 404 |
| Page registers | 52, 396, 397, 401, 403, 404 |
| Registers | 403, 404 |
| Subroutines | 88, 397, 471 |
| Extended addressing mode | 41 |
| Extended division | 75 |
| Extension byte | 43 |
| External interrupts | 330 |
| External queue reconstruction | 336 |
| External reset | 327 |

F

| | |
|---------------------------------------|--|
| Fast math | 454 |
| FDIV instruction | 195 |
| Fractional division | 75 |
| Frame pointer | 468 |
| Free cycle | 104, 410 |
| Fuzzy logic. | 356 |
| Antecedents | 362 |
| Consequents | 362 |
| Custom programming. | 388 |
| Defuzzification | 79, 363, 383, 384, 385, 387 |
| Fuzzification | 78, 359, 388 |
| Inference kernel | 78, 358, 364 |
| Inputs | 78, 392 |
| Instructions. | 78, 245, 246, 271, 272, 273, 274, 275, 320, 356, 366, 372, 374, 377, 378, 379, 383, 461 |
| Interrupts | 380, 384, 385, 387 |
| Knowledge base. | 358, 362 |
| Membership functions | 78, 245, 246, 356, 357, 358, 359, 366, 368, 370, 371, 388, 389 |
| Outputs. | 78, 392 |
| Rule evaluation | 78, 271, 272, 273, 274, 275, 356, 361, 372, 374, 377, 378, 379, 380, 381, 391 |
| Rules | 358, 362 |
| Sets | 357 |
| Tabular membership functions. | 82, 388 |
| Weighted average | 79, 320, 356, 363, 383, 384, 385, 387 |

G

| | |
|--|----|
| General purpose accumulators | 28 |
|--|----|

H

| | |
|--------------------------|----------------------------------|
| H status bit. | 32, 110, 113, 114, 115, 116, 172 |
| High-level language. | 465, 469 |
| Addressing modes | 465, 468, 470 |
| Condition codes register | 470 |
| Expanded memory | 471 |
| Instructions | 465 |
| Loop primitives | 469 |
| Stack | 466, 467 |

I

| | |
|---------------------------------|---|
| I mask bit. | 33, 120, 158, 289, 302, 311, 319, 326 |
| IBEQ instruction. | 196, 431 |
| IBNE | 431 |
| IBNE instruction. | 197 |
| IDIV instruction | 198 |
| IDIVS instruction | 199, 469 |
| Immediate addressing mode | 39 |
| INC instruction | 200 |
| INCA instruction | 201 |
| INCB instruction | 202 |
| Increment instructions | 71, 200, 201, 202, 203, 204, 205 |
| Index calculation instructions | 93, 111, 112, 180, 181, 204, 205, 233, 234, 455 |
| Index manipulation instructions | 91, 170, 171, 194, 230, 231, 263, 264, 269, 270, 297, 298, 309, 315, 316, 317, 318, 321, 322 |
| Index registers | 28, 29, 91, 468 |
| X | 45, 111, 170, 174, 175, 180, 194, 195, 196, 197, 198, 199, 204, 230, 232, 233, 234, 263, 269, 271, 274, 283, 291, 297, 302, 306, 307, 308, 309, 315, 317, 321 |
| Y | 45, 112, 171, 174, 175, 181, 182, 183, 184, 189, 190, 194, 196, 197, 205, 231, 232, 233, 234, 264, 270, 271, 273, 283, 291, 298, 302, 306, 307, 308, 309, 316, 318, 322 |

| | |
|---|-------------------------------------|
| Indexed addressing modes | 29, 43, 44, 449, 450, 451, 452, 453 |
| 16-bit constant indirect | 47 |
| 16-bit constant offset | 46 |
| 5-bit constant offset | 45 |
| 9-bit constant offset | 46 |
| Accumulator direct | 49 |
| Accumulator offset | 49 |
| Automatic indexing | 47 |
| Base index register | 45–50 |
| Extension byte | 43 |
| Postbyte | 45 |
| Postbyte encoding | 43 |
| Inference kernel, fuzzy logic | 364 |
| Inherent addressing mode | 39 |
| INS instruction | 203 |
| Instruction queue | 22, 35, 56, 336, 446 |
| Alignment | 56 |
| Buffer | 56 |
| Debugging | 336 |
| Movement cycles | 57 |
| Reconstruction | 336, 337, 340, 342 |
| Stages | 56, 336 |
| Status registers | 341, 342 |
| Status signals | 56, 337, 338, 339, 340, 343 |
| Instruction set | 413 |
| Integer division | 75 |
| Interrupt instructions | 89 |
| Interrupts | 328 |
| Enabling and disabling | 32, 33, 158, 289, 326 |
| External | 330 |
| I mask bit | 33, 158, 289, 302, 319, 330 |
| Instructions | 89, 158, 283, 289, 302, 311 |
| Low-power stop | 95, 295 |
| Maskable | 33, 329 |
| Non-maskable | 32, 325, 328 |
| Recognition | 329 |
| Return | 33, 90, 283, 330 |
| Service routines | 330 |
| Software | 89, 302, 331 |

| | |
|----------------------------|-------------------|
| Stacking | 329 |
| Vectors | 328 |
| Wait instruction | 95, 319 |
| X mask bit | 32, 295, 319, 330 |
| INX instruction | 204 |
| INY instruction | 205 |

J

| | |
|-----------------------------|---------|
| JMP instruction | 62, 206 |
| JSR instruction | 59, 207 |
| Jump instructions | 88 |
| Jumps | 62 |

K

| | |
|--------------------------|-----|
| Knowledge base | 358 |
|--------------------------|-----|

L

| | |
|----------------------------|-----|
| LBCC instruction | 208 |
| LBCS instruction | 209 |
| LBEQ instruction | 210 |
| LBGE instruction | 211 |
| LBGT instruction | 212 |
| LBHI instruction | 213 |
| LBHS instruction | 214 |
| LBLE instruction | 215 |
| LBLO instruction | 216 |
| LBLS instruction | 217 |
| LBLT instruction | 218 |
| LBMI instruction | 219 |
| LBNE instruction | 220 |
| LBPL instruction | 221 |
| LBRA instruction | 222 |
| LBRN instruction | 223 |
| LBVC instruction | 224 |
| LBVS instruction | 225 |

| | |
|---------------------------------------|--|
| LDAA instruction | 226 |
| LDAB instruction | 227 |
| LDD instruction | 228 |
| LDS instruction | 229 |
| LDX instruction | 230 |
| LDY instruction | 231 |
| LEAS instruction | 232, 467, 470 |
| Least significant byte | 26 |
| Least significant word | 26 |
| LEAX instruction | 233, 470 |
| LEAY instruction | 234, 470 |
| Legal label | 101 |
| Literal expression | 101 |
| Load instructions | 66, 226, 227, 228, 229, 230, 231, 232, 233, 234 |
| Logic level one | 26 |
| Logic level zero | 26 |
| Loop primitive instructions | 62, 174, 175, 196, 197, 306, 308, 431, 459, 469 |
| Offset values | 87 |
| Postbyte encoding | 431 |
| Low-power stop | 95, 295 |
| LSL instruction | 235 |
| LSL mnemonics | 77 |
| LSLA instruction | 236 |
| LSLB instruction | 237 |
| LSLD instruction | 238 |
| LSR instruction | 239 |
| LSRA instruction | 240 |
| LSRB instruction | 241 |
| LSRD instruction | 242 |

M

| | |
|---|---------------------------------------|
| M68HC11 compatibility | 39, 442 |
| M68HC11 instruction mnemonics | 443 |
| Maskable interrupts | 329 |
| MAXA instruction | 243 |
| Maximum instructions | 81, 460 |
| 16-bit | 185, 186 |
| 8-bit | 243, 244 |
| MAXM instruction | 244, 356 |
| MEM instruction. | 78, 245, 246, 356, 366, 368, 370, 371 |
| Membership functions | 358 |
| Memory and addressing symbols | 24 |
| Memory expansion | |
| Addressing | 405 |
| Bank switching | 405 |
| Overlay windows | 405 |
| Page registers | 399, 405 |
| MINA instruction | 247, 356 |
| Minimum instructions. | 81, 460 |
| 16-bit | 187, 188 |
| 8-bit | 247, 248 |
| MINM instruction | 248 |
| Misaligned instructions | 61 |
| Mnemonic | 26 |
| Mnemonic ranges | 26 |
| Most significant byte | 26 |
| MOVB instruction. | 249 |
| Move instructions. | 68, 249, 250, 455, 458 |
| Destination | 50 |
| Multiple addressing modes | 50 |
| PC relative addressing. | 50 |
| Reference index register | 50 |
| Source | 50 |
| MOVW instruction | 250 |
| MUL instruction | 251 |

| | |
|---|--------------|
| Multiple addressing modes | |
| Bit manipulation instructions | 52, 129, 150 |
| Move instructions | 50, 249, 250 |
| Multiplication instructions..... | 75 |
| 16-bit | 189, 190 |
| 8-bit | 251 |
| Multiply and accumulate instructions..... | 82, 184, 320 |

N

| | |
|---------------------------------------|------------------------|
| N status bit | 33, 143, 145, 219, 221 |
| NEG instruction | 252 |
| NEGA instruction..... | 253 |
| Negate instructions | 74, 252, 253, 254 |
| Negated..... | 26 |
| Negative integers..... | 34 |
| NEGB instruction..... | 254 |
| Non-maskable interrupts | 325, 328 |
| NOP instruction | 96, 255 |
| Notation | |
| Branch taken/not taken | 109, 410 |
| Changes in CCR bits | 99 |
| Cycle-by-cycle operation | 104 |
| Memory and addressing..... | 24 |
| Object code | 100 |
| Operators..... | 25 |
| Source forms | 101 |
| System resources | 23 |
| Null operation instruction..... | 96, 255 |
| Numeric range of branch offsets | 42 |

O

| | |
|----------------------------|--------------|
| Object code notation | 100 |
| Odd bytes | 35 |
| Opcodes | 444, 453 |
| Operators..... | 25 |
| Optional cycles | 61, 106, 410 |

| | |
|----------------------------|-----|
| ORAA instruction | 256 |
| ORAB instruction | 257 |
| ORCC instruction | 258 |
| Orthogonality | 472 |

P

| | |
|--|---|
| Pointer calculation instructions | 93, 232, 233, 234 |
| Pointers | 470 |
| Postbyte | 45, 194, 291, 309 |
| Postbyte encoding | |
| Exchange instructions | 430 |
| Loop primitive instruction | 431 |
| Transfer instructions | 430 |
| Post-decrement indexed addressing mode | 47 |
| Post-increment indexed addressing mode | 47 |
| Power conservation | 95, 295, 319 |
| Power-on reset | 327 |
| Pre-decrement indexed addressing mode | 47 |
| Pre-increment indexed addressing mode | 47 |
| Priority, exception | 325 |
| Program counter | 28, 30, 43, 133, 151, 154, 207, 232, 233, 234, 249, 250, 255, 283, 284, 302, 307, 311 |
| Program word access cycle | 106, 410 |
| Programming model | 21, 28, 445 |
| Pseudo-non-maskable interrupt | 325 |
| PSHA instruction | 259 |
| PSHB instruction | 260 |
| PSHC instruction | 261 |
| PSHD instruction | 262 |
| PSHX instruction | 263 |
| PSHY instruction | 264 |
| PULA instruction | 265 |
| PULB instruction | 266 |
| PULC instruction | 267 |
| PULD instruction | 268, 467 |

| | |
|-----------------------------|-----|
| Pull instructions | 471 |
| PULX instruction | 269 |
| PULY instruction | 270 |
| Push instructions | 471 |
| PUSHD instruction | 467 |

R

| | |
|-------------------------------------|---|
| Range of mnemonics | 26 |
| Read 8-bit data cycle | 106, 410 |
| Read indirect PPAGE cycle | 105, 410 |
| Read PPAGE cycle | 105, 410 |
| Read 16-bit data cycle | 107, 410 |
| Register designators | 101 |
| Relative addressing mode | 41 |
| Relative offset | 41 |
| Resets | 324, 326 |
| Clock monitor | 327 |
| COP | 327 |
| External | 327 |
| Power-on | 327 |
| REV instruction | 78, 271, 272, 356, 361, 372, 374, 377, 378, 379, 380, 381, 391 |
| REVVW instruction | 78, 273, 274, 275, 356, 361, 372, 374, 377, 378, 379, 380, 381, 391 |
| ROL instruction | 276 |
| ROLA instruction | 277 |
| ROLB instruction | 278 |
| ROM, BDM | 344 |
| ROR instruction | 279 |
| RORA instruction | 280 |
| RORB instruction | 281 |
| Rotate instructions | 77, 276, 277, 278, 279, 280, 281 |
| RTC instruction | 52, 59, 88, 282, 397, 398, 400, 462, 471 |
| RTI instruction | 33, 90, 283, 330 |
| RTS instruction | 59, 284 |

S

| | |
|--|--|
| S control bit | 31, 295 |
| SBA instruction | 285 |
| SBCA instruction | 286 |
| SBCB instruction | 287 |
| SEC instruction | 288 |
| SEI instruction | 289 |
| Set | 26 |
| Setting memory bits | 150 |
| SEV instruction | 290 |
| SEX instruction | 67, 291 |
| Shift instructions | 77 |
| Arithmetic | 121, 122, 123, 124, 125, 126, 127 |
| Logical | 235, 236, 237, 238, 239, 240, 241, 242 |
| Sign extension instruction | 291 |
| Signed branches | 83 |
| Signed integers | 34 |
| Signed multiplication | 75 |
| Sign-extension instruction | 67, 466 |
| Simple branches | 83 |
| Software interrupts | 302 |
| Source code compatibility | 21, 442 |
| Source form notation | 101 |
| Specific mnemonic | 26 |
| STAA instruction | 292 |
| STAB instruction | 293 |
| Stack | 29, 448 |
| Interrupts | 283, 302 |
| Stop and wait | 295, 319 |
| Subroutines | 151, 154, 207, 282, 284 |
| Traps | 311 |
| Stack 16-bit data cycle | 107, 410 |
| Stack 8-bit data cycle | 107, 410 |
| Stack operation instructions | 92, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270 |

| | |
|-----------------------------------|--|
| Stack pointer | 28, 29, 43, 151, 154, 169, 174, 175, 179, 194, 196, 197, 203, 207, 229, 232, 233, 234, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 284, 291, 296, 306, 307, 308, 309, 315, 316, 317, 318, 466 |
| Initialization | 29 |
| Manipulation..... | 92, 169, 179, 203, 229, 232, 259, 260, 296, 315, 316, 317, 318 |
| Stacking order | 29, 448 |
| Stack pointer instructions | 92, 169, 179, 203, 229, 232, 296, 309, 315, 316, 317, 318, 462, 466 |
| Stacking instructions | 259, 260 |
| Standard CPU12 address space..... | 35 |
| STD instruction | 294 |
| STOP instruction | 31, 95, 295 |
| Store instructions..... | 66, 292, 293, 294, 296, 297, 298 |
| STS instruction | 296 |
| STX instruction | 297 |
| STY instruction | 298 |
| SUBA instruction | 299 |
| SUBB instruction | 300 |
| SUBD instruction | 301 |
| Subroutine instructions | 88 |
| Subroutines | 59, 207, 471 |
| Expanded memory..... | 59, 88, 154, 155, 282 |
| Instructions..... | 88, 151, 207, 471 |
| Return | 282, 284 |
| Subtraction instructions..... | 69, 285, 286, 287, 299, 300, 301 |
| SWI instruction | 89, 302, 331 |
| Switch statements | 470 |
| Symbols and notation | 23 |

T

| | |
|--|-------------------|
| TAB instruction | 303 |
| Table interpolation instructions | 82, 193, 307, 461 |
| Tabular membership functions | 388, 389 |
| TAP instruction | 304 |

| | |
|--|--|
| TBA instruction | 305 |
| TBEQ instruction | 306, 431 |
| TBL instruction..... | 82, 307, 356, 388, 389 |
| TBNE instruction | 308, 431 |
| Termination of interrupt service routines..... | 90, 283, 330 |
| Termination of subroutines | 282, 284 |
| Test instructions | 72, 137, 138, 306, 308, 312, 313, 314 |
| TFR instruction | 291, 304, 309, 310, 315, 316, 317, 318 |
| TPA instruction | 310 |
| Transfer and exchange instructions | 466 |
| Transfer instructions | 67, 303, 304, 305, 309, 310, 315, 316, 317, 318, 455, 459 |
| Postbyte encoding | 430 |
| TRAP instruction | 90, 311, 331 |
| TST | 312 |
| TST instruction | 312 |
| TSTA instruction | 313 |
| TSTB instruction | 314 |
| TSX instruction | 315 |
| TSY instruction | 316 |
| Twos-complement form..... | 34 |
| TXS instruction | 317 |
| Types of instructions | |
| Addition and Subtraction | 69 |
| Background and null | 96 |
| Binary-coded decimal..... | 70 |
| Bit test and manipulation | 76 |
| Boolean logic | 73 |
| Branch | 83 |
| Clear, complement, and negate | 74 |
| Compare and test..... | 72 |
| Condition code | 94 |
| Decrement and increment | 71 |
| Fuzzy logic | 78 |
| Index manipulation..... | 91 |
| Interrupt | 89, 90 |
| Jump and subroutine | 88 |

| | |
|---|-----|
| Load and store | 66 |
| Loop primitives | 87 |
| Maximum and minimum | 81 |
| Move | 68 |
| Multiplication and division | 75 |
| Multiply and accumulate | 82 |
| Pointer and index calculation | 93 |
| Shift and rotate | 77 |
| Sign extension | 67 |
| Stacking | 92 |
| Stop and wait | 95 |
| Table interpolation | 82 |
| Transfer and exchange | 67 |
| TYS instruction | 318 |

U

| | |
|--------------------------------------|---|
| Unary branches | 83 |
| Unimplemented opcode trap | 90, 311, 324, 326 |
| Unsigned branches | 83 |
| Unsigned multiplication | 75 |
| Unstack 16-bit data cycle | 108, 410 |
| Unstack 8-bit data cycle | 107, 410 |
| Unweighted rule evaluation | 271, 272, 361, 372, 374, 377, 378, 379, 380, 381, 391 |

V

| | |
|------------------------------|--|
| V status bit. | 34, 152, 153, 162, 224, 225, 271, 272, 273, 274, 290 |
| Vector fetch cycle | 108, 410 |
| Vectors, exception | 324, 333 |

W

| | |
|------------------------------------|--|
| WAI instruction | 95, 319 |
| WAV instruction | 79, 82, 320, 356, 363, 383, 384, 385, 387 |
| Wavr pseudoinstruction | 384, 385, 387 |
| Weighted average | 320 |
| Weighted rule evaluation | 273, 274, 275, 361, 372, 374, 377, 378, 379, 380, 381, 391 |

| | |
|------------------------------|----------|
| Word moves | 250 |
| Write 16-bit data cycle..... | 107, 410 |
| Write 8-bit data cycle..... | 107, 410 |
| Write PPAGE cycle | 105, 410 |

X

| | |
|-----------------------|---------------------------------------|
| X mask bit | 32, 194, 267, 283, 295, 304, 309, 319 |
| XGDX instruction..... | 321 |
| XGDY instruction..... | 322 |

Z

| | |
|----------------------------|---|
| Z status bit..... | 33, 131, 144, 185, 186, 187, 188, 204, 205, 210, 220, 243, 244, 247, 248 |
| Zero-page addressing | 40 |

How to Reach Us:

USA/EUROPE/LOCATIONS NOT LISTED:

Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-303-675-2140
1-800-441-2447

TECHNICAL INFORMATION CENTER: 1-800-521-6274

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://www.motorola.com/semiconductors/>



MOTOROLA

**CPU12RM/AD
REV 2**