STAR12 V1.5 Core
User Guide
Version 1.2.1

Original Release Date: 12 May 2000
Revised: 10 April 2001

M-Core Technology Center
Motorola, Inc

Motorola reserves the right to make changes without further notice to any products herein to improve reliability, function or design. Motorola does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Revision History

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Release Number	Date	Author	Summary of Changes
1.2	10 April 2001	George Grimmer	Update allocated RAM space table.
1.2	13 October 2000	George Grimmer	Security enhancements, add core_exp_t2, core_per_t2 outputs and peri_clk2, peri_clk4, and ram_fmts inputs.
1.1	21 July 2000	George Grimmer	Correct access detail for LSL instruction in appendix B
1.0	12 May 2000	Neil Spake	Original draft. Distributed only within Motorola

Table of Contents

Section 1 Introduction

1.1 Core Overview 23
1.2 Features 23
1.3 Block Diagram 25
1.4 Architectural Summary 26
1.5 Programming Model 26
1.6 Data Format Summary 27
1.6.1 Data Types 27
1.6.2 Memory Organization 28
1.7 Addressing modes 28
1.8 Instruction Set Overview 29
1.8.1 Register and Memory Notation 43
1.8.2 Source Form Notation 43
1.8.3 Operation Notation 45
1.8.4 Address Mode Notation 45
1.8.5 Machine Code Notation 46
1.8.6 Access Detail Notation 47
1.8.7 Condition Code State Notation 49

Section 2 Nomenclature

2.1 References 51
2.2 Units and Measures 51
2.3 Symbology 51
2.4 Terminology 51

Section 3 Core Registers

3.1 Programming Model 53
3.1.1 Accumulators 53
3.1.2 Index Registers (X and Y) 54
3.1.3 Stack Pointer (SP) 55
3.1.4 Program Counter (PC) 56
3.1.5 Condition Code Register (CCR) 56
3.2 Core Register Map 58

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 4 Instructions

4.1 Instruction Types 63
4.2 Addressing Modes 63
4.2.1 Effective Address 64
4.2.2 Inherent Addressing Mode 64
4.2.3 Immediate Addressing Mode 64
4.2.4 Direct Addressing Mode 65
4.2.5 Extended Addressing Mode 65
4.2.6 Relative Addressing Mode 66
4.2.7 Indexed Addressing Modes 66
4.2.8 Instructions Using Multiple Modes 71
4.3 Instruction Descriptions 73
4.3.1 Load and Store Instructions 73
4.3.2 Transfer and Exchange Instructions 74
4.3.3 Move Instructions 74
4.3.4 Add and Subtract Instructions 75
4.3.5 Binary Coded Decimal Instructions 76
4.3.6 Decrement and Increment Instructions 76
4.3.7 Compare and Test Instructions 77
4.3.8 Boolean Logic Instructions 78
4.3.9 Clear, Complement, and Negate Instructions 78
4.3.10 Multiply and Divide Instructions 79
4.3.11 Bit Test and Bit Manipulation Instructions 79
4.3.12 Shift and Rotate Instructions 80
4.3.13 Fuzzy Logic Instructions 81
4.3.14 Maximum and Minimum Instructions 81
4.3.15 Multiply and Accumulate Instruction 83
4.3.16 Table Interpolation Instructions 83
4.3.17 Branch Instructions 84
4.3.18 Jump and Subroutine Instructions 86
4.3.19 Interrupt Instructions 87
4.3.20 Index Manipulation Instructions 88
4.3.21 Stacking Instructions 89
4.3.22 Load Effective Address Instructions 90
4.3.23 Condition Code Instructions 90
4.3.24 STOP and WAI Instructions 91

4.3.25 Background Mode and Null Operation Instructions 91
4.4 High-Level Language Support 92
4.4.1 Data Types 92
4.4.2 Parameters and Variables 92
4.4.3 Increment and Decrement Operators 94
4.4.4 Higher Math Functions 94
4.4.5 Conditional If Constructs 94
4.4.6 Case and Switch Statements 95
4.4.7 Pointers 95
4.4.8 Function Calls 95
4.4.9 Instruction Set Orthogonality 95
4.5 Opcode Map 97
4.6 Transfer and Exchange Postbyte Encoding 99
4.7 Loop Primitive Postbyte (lb) Encoding 100
4.8 Indexed Addressing Postbyte (xb) Encoding 101

Section 5 Instruction Execution

5.1 Normal Instruction Execution 103
5.2 Execution Sequence 103
5.2.1 No Movement 103
5.2.2 Advance and Load from Data Bus 103
5.3 Changes of Flow 104
5.3.1 Exceptions 104
5.3.2 Subroutines 104
5.3.3 Branches 104
5.3.4 Jumps 106
5.4 Instruction Timing 106
5.4.1 Register and Memory Notation 120
5.4.2 Source Form Notation 121
5.4.3 Operation Notation 122
5.4.4 Address Mode Notation 122
5.4.5 Machine Code Notation 123
5.4.6 Access Detail Notation 123
5.4.7 Condition Code State Notation 126
5.5 External Visibility Of Instruction Queue 126
5.5.1 Instruction Queue Status Signals 126

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.5.2 No Movement (0:0) 128
5.5.3 ALD — Advance and Load from Data Bus (1:0) 128
5.5.4 INT — Start Interrupt (0:1) 128
5.5.5 SEV — Start Even Instruction (1:0) 128
5.5.6 SOD — Start Odd Instruction (1:1) 129

Section 6 Exception Processing

6.1 Exception Processing Overview 131
6.1.1 Reset Processing 133
6.1.2 Interrupt Processing 133
6.2 Exception Vectors 135
6.3 Exception Types 136
6.3.1 Resets 136
6.3.2 Interrupts 137

Section 7 Core Interface

7.1 Core Interface Overview 141
7.1.1 Signal Summary 142
7.2 Signal Descriptions 145
7.2.1 Internal Bus Interface Signals 145
7.2.2 External Bus Interface Signals 148
7.2.3 Clock and Reset Signals 150
7.2.4 Vector Request/Acknowledge Signals 151
7.2.5 Stop and Wait Mode Control/Status Signals 151
7.2.6 Background Debug Mode (BDM) Interface Signals 151
7.2.7 Memory Configuration Signals 152
7.2.8 Scan Control Interface Signals 152
7.3 Interface Operation 152
7.3.1 Read Operations 152
7.3.2 Write Operations 155
7.3.3 Multiplexed External Bus Interface 158
7.3.4 General Internal Read Visibility Timing 161
7.3.5 Detecting Access Type from External Signals 162

Section 8 Core Clock and Reset Connections

8.1 Clocking Overview 163

8.1.1 Basic Clock Relationship 164
8.1.2 Reset Relationship 165
8.1.3 Phase-Locked Loop Interface 165
8.1.4 STAR12 CPU Wait and Stop Modes 166
8.2 Signal Summary 166
8.3 Detailed Clock and Reset Signal Descriptions 167
8.3.1 Clock and Reset Signals 167
8.3.2 Stop and Wait Mode Control/Status Signals 168

Section 9 Core Power Connections

9.1 Power Overview 169
9.1.1 Power and Ground Summary 169

Section 10 Interrupt (INT)

10.1 Overview 171
10.1.1 Features 171
10.1.2 Block Diagram 172
10.2 Interface Signals 172
10.3 Registers 173
10.3.1 Interrupt Test Control Register 173
10.3.2 Interrupt Test Registers 174
10.3.3 Highest Priority I Interrupt (Optional) 175
10.4 Operation 175
10.4.1 Interrupt Exception Requests 175
10.4.2 Reset Exception Requests 176
10.4.3 Exception Priority 176
10.5 Modes of Operation 177
10.5.1 Normal Operation 177
10.5.2 Special Operation 177
10.5.3 Emulation Modes 177
10.6 Low-Power Options 177
10.6.1 Run Mode 177
10.6.2 Wait Mode 177
10.6.3 Stop Mode 177
10.7 Motorola Internal Information 177

Section 11 Module Mapping Control (MMC)

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.1 Overview. 179
11.1.1 Features. 179
11.1.2 Block Diagram 180
11.2 Interface Signals. 180
11.3 Registers 181
11.3.1 Initialization of Internal RAM Position Register(INITRM) 182
11.3.2 Initialization of Internal Registers Position Register (INITRG) 182
11.3.3 Initialization of Internal EEPROM Position Register (INITEE) 183
11.3.4 Miscellaneous System Control Register (MISC) 184
11.3.5 Reserved Test Register Zero (MTST0) 185
11.3.6 Reserved Test Register One (MTST1) 185
11.3.7 Memory Size Register Zero (MEMSIZ0) 186
11.3.8 Memory Size Register One (MEMSIZ1). 187
11.3.9 Program Page Index Register (PPAGE) 188
11.4 Operation 189
11.4.1 Bus Control 189
11.4.2 Address Decoding 189
11.4.3 Memory Expansion 191
11.5 Motorola Internal Information 196
11.5.1 Test Registers 196
11.5.2 MMC Bus Control. 198

Section 12 Multiplexed External Bus Interface (MEBI)

12.1 Overview. 201
12.1.1 Features. 201
12.1.2 Block Diagram 202
12.2 Interface Signals. 202
12.2.1 MEBI Signal Descriptions. 203
12.3 Registers 207
12.3.1 Port A Data Register(PORTA) 208
12.3.2 Data Direction Register A (DDRA). 209
12.3.3 Port B Data Register (PORTB) 210
12.3.4 Data Direction Register B (DDRB). 210
12.3.5 Port E Data Register (PORTE) 211
12.3.6 Data Direction Register E (DDRE). 212
12.3.7 Port E Assignment Register (PEAR) 213

12.3.8 MODE Register (MODE) 215
12.3.9 Pullup Control Register (PUCR) 218
12.3.10 Reduced Drive Register (RDRIV) 219
12.3.11 External Bus Interface Control Register (EBICTL) 220
12.3.12 IRQ Control Register (IRQCR) 220
12.3.13 Reserved Registers 222
12.3.14 Port K Data Register (PORTK) 222
12.3.15 Port K Data Direction Register (DDRK) 223
12.4 Operation 224
12.4.1 External Bus Control 224
12.4.2 External Data Bus Interface 224
12.4.3 Control 224
12.4.4 Registers 224
12.4.5 External System Pin Functional Descriptions 225
12.4.6 Detecting Access Type from External Signals 226
12.4.7 Stretched Bus Cycles 227
12.4.8 Modes of Operation 227
12.4.9 Internal Visibility 231
12.4.10 Secure Mode 232
12.5 Low-Power Options 232
12.5.1 Run Mode 232
12.5.2 Wait Mode 232
12.5.3 Stop Mode 232
12.6 Motorola Internal Information 232
12.6.1 Peripheral Mode Operation 232
12.6.2 Special Test Clock 233

Section 13 Breakpoint (BKP)

13.1 Overview 235
13.1.1 Features 235
13.1.2 Block Diagram 236
13.2 Interface Signals 238
13.3 Registers 238
13.3.1 Breakpoint Control Register 0 (BKPCT0) 238
13.3.2 Breakpoint Control Register 1 (BKPCT1) 239
13.3.3 Breakpoint First Address Expansion Register (BKP0X) 242

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

13.3.4 Breakpoint First Address High Byte Register (BKP0H) 243
13.3.5 Breakpoint First Address Low Byte Register (BKP0L) 243
13.3.6 Breakpoint Second Address Expansion Register (BKP1X) 243
13.3.7 Breakpoint Data (Second Address) High Byte Register (BKP1H) 244
13.3.8 Breakpoint Data (Second Address) Low Byte Register (BKP1L) 244
13.4 Operation 245
13.4.1 Modes of Operation 245
13.4.2 Breakpoint Priority 246
13.5 Motorola Internal Information 246

Section 14 Background Debug Mode (BDM)

14.1 Overview. 247
14.1.1 Features 247
14.1.2 Block Diagram 248
14.2 Interface Signals. 248
14.2.1 Background Interface Pin (BKGD) 248
14.2.2 High Byte Instruction Tagging Pin ($\overline{\text{TAGHI}}$) 248
14.2.3 Low Byte Instruction Tagging Pin ($\overline{\text{TAGLO}}$) 249
14.3 Registers 249
14.3.1 BDM Status Register 250
14.3.2 BDM CCR Holding Register 252
14.3.3 BDM Internal Register Position Register 253
14.4 Operation 253
14.4.1 Security 253
14.4.2 Enabling and Activating BDM. 254
14.4.3 BDM Hardware Commands 254
14.4.4 Standard BDM Firmware Commands 255
14.4.5 BDM Command Structure 256
14.4.6 BDM Serial Interface 258
14.4.7 Instruction Tracing 260
14.4.8 Instruction Tagging. 260
14.5 Modes of Operation 261
14.5.1 Normal Operation. 261
14.5.2 Special Operation. 262
14.5.3 Emulation Modes 262
14.6 Low-Power Options 262

14.6.1 Run Mode 262
14.6.2 Wait Mode 262
14.6.3 Stop Mode 262
14.7 Interrupt Operation 262
14.8 Motorola Internal Information 262
14.8.1 Registers 263
14.8.2 BDM Instruction Register (Hardware) 264
14.8.3 BDM Instruction Register (Firmware) 265
14.8.4 BDM Status Register 266
14.8.5 BDM Shift Register 267
14.8.6 BDM Address Register 268
14.8.7 Special Peripheral Mode 268
14.8.8 Standard BDM Firmware Listing 268
14.8.9 Secured Mode BDM Firmware Listing 275

Section 15 Secured Mode of Operation

15.1 Overview 279
15.1.1 Features 279
15.1.2 Block Diagram 280
15.2 Interface Signals 280
15.3 Registers 281
15.4 Operation 281
15.4.1 Normal Single-Chip Mode 281
15.4.2 Expanded Mode 281
15.4.3 Unsecuring The System 281
15.5 Motorola Internal Information 283
15.5.1 BDM Secured Mode Firmware 283

Appendix A Electrical Characteristics

A.1 General 285

Appendix B Instruction Set and Commands

B.1 General 287
B.2 Glossary Notation 287
B.2.1 Condition Code State Notation 287
B.2.2 Register and Memory Notation 288

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

B.2.3 Address Mode Notation 289
B.2.4 Operator Notation. 289
B.2.5 Machine Code Notation 289
B.2.6 Source Form Notation 290
B.2.7 CPU Cycles Notation 291
B.3 Glossary 294

Appendix C Fuzzy Logic Support

C.1 General. 505
C.2 Introduction. 505
C.3 Fuzzy Logic Basics. 505
C.3.1 Fuzzification (MEM) 507
C.3.2 Rule Evaluation (REV and REVW) 508
C.3.3 Defuzzification (WAV) 510
C.4 Example Inference Kernel 510
C.5 MEM Instruction Details 512
C.5.1 Membership Function Definitions. 512
C.5.2 Abnormal Membership Function Definitions 513
C.6 REV, REVW Instruction Details 516
C.6.1 Unweighted Rule Evaluation (REV) 516
C.6.2 Weighted Rule Evaluation (REVW) 520
C.7 WAV Instruction Details 525
C.7.1 Initialization Prior to Executing WAV 525
C.7.2 WAV Interrupt Details. 525
C.7.3 Cycle-by-Cycle Details for WAV and wavr 526
C.8 Custom Fuzzy Logic Programming 529
C.8.1 Fuzzification Variations 529
C.8.2 Rule Evaluation Variations. 531
C.8.3 Defuzzification Variations. 532

Appendix D M68HC11 to STAR12 Upgrade

D.1 General. 533
D.2 Source Code Compatibility. 533
D.3 Programmer’s Model and Stacking 535
D.4 True 16-Bit Architecture 535
D.4.1 Bus Structures 535

D.4.2 Instruction Queue 535
D.4.3 Stack Function 536
D.5 Improved Indexing 537
D.5.1 Constant Offset Indexing 538
D.5.2 Autoincrement/Autodecrement Indexing 539
D.5.3 Accumulator Offset Indexing 539
D.5.4 Indirect Indexing 540
D.6 Improved Performance 540
D.6.1 Reduced Cycle Counts 540
D.6.2 Fast Math 540
D.6.3 Code Size Reduction 541
D.7 Additional Functions 542
D.7.1 New Instructions 542
D.7.2 Memory-to-Memory Moves 544
D.7.3 Universal Transfer and Exchange 544
D.7.4 Loop Construct 544
D.7.5 Long Branches 544
D.7.6 Minimum and Maximum Instructions 545
D.7.7 Fuzzy Logic Support 545
D.7.8 Table Lookup and Interpolation 546
D.7.9 Extended Bit Manipulation 546
D.7.10 Push and Pull D and CCR 546
D.7.11 Compare SP 546
D.7.12 Support for Memory Expansion 546

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

List of Figures

Figure 1-1 Core Block Diagram 25
Figure 1-2 Programming Model 27
Figure 3-1 Programming Model 53
Figure 3-2 Accumulator A 54
Figure 3-3 Accumulator B 54
Figure 3-4 Index Register X 54
Figure 3-5 Index Register Y 54
Figure 3-6 Stack Pointer (SP) 55
Figure 3-7 Program Counter (PC) 56
Figure 3-8 Condition Code Register (CCR) 56
Figure 3-9 Core Register Map Summary 61
Figure 5-1 Queue Status Signal Timing 127
Figure 7-1 Core Interface Signals 142
Figure 7-2 Basic 8-bit Peripheral Read Timing 153
Figure 7-3 Basic 16-bit Peripheral Read Timing 153
Figure 7-4 Basic 8-bit Memory Read Timing 154
Figure 7-5 Basic 16-bit Memory Read Timing 154
Figure 7-6 Basic 8-bit Core Register Read Timing 155
Figure 7-7 Basic 16-bit Core Register Read Timing 155
Figure 7-8 Basic 8-bit Peripheral Write Timing 156
Figure 7-9 Basic 16-bit Peripheral Write Timing 156
Figure 7-10 Basic 8-bit Memory Write Timing 157
Figure 7-11 Basic 16-bit Memory Write Timing 157
Figure 7-12 Basic 8-bit Core Register Write Timing 158
Figure 7-13 Basic 16-bit Core Register Write Timing 158
Figure 7-14 General External Bus Timing 159
Figure 7-15 General Internal Read Visibility Timing 161
Figure 8-1 Core Interface Signals 164
Figure 8-2 System Clock Timing Diagram 165
Figure 10-1 Interrupt Block Diagram 172
Figure 10-2 Interrupt Register Summary 173
Figure 10-3 Interrupt Test Control Register (ITCR) 173
Figure 10-4 Interrupt TEST Registers (ITEST) 174

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Figure 10-5 Highest Priority I Interrupt Register (HPRIO) 175
Figure 11-1 Module Mapping Control Block Diagram 180
Figure 11-2 Module Mapping Control Register Summary 181
Figure 11-3 INITRM Register 182
Figure 11-4 INITRG Register 182
Figure 11-5 INITEE Register 183
Figure 11-6 Miscellaneous System Control Register (MISC) 184
Figure 11-7 Reserved Test Register Zero (MTST0) 185
Figure 11-8 Reserved Test Register One (MTST1) 185
Figure 11-9 Memory Size Register Zero 186
Figure 11-10 Memory Size Register One 187
Figure 11-11 Program Page Index Register (PPAGE) 188
Figure 11-13 Mapping Test Register Zero (MTST0) 196
Figure 11-14 Mapping Test Register One (MTST1) 197
Figure 12-1 MEBI Block Diagram 202
Figure 12-2 MEBI Register Map Summary 207
Figure 12-3 Port A Data Register (PORTA) 208
Figure 12-4 Data Direction Register A (DDRA) 209
Figure 12-5 Port B Data Register (PORTB) 210
Figure 12-6 Data Direction Register B (DDRB) 210
Figure 12-7 Port E Data Register (PORTE) 211
Figure 12-8 Data Direction Register E (DDRE) 212
Figure 12-9 Port E Assignment Register (PEAR) 213
Figure 12-10 MODE Register (MODE) 215
Figure 12-11 Pullup Control Register (PUCR) 218
Figure 12-12 Reduced Drive Register (RDRIV) 219
Figure 12-13 External Bus Interface Control Register (EBICTL) 220
Figure 12-14 IRQ Control Register (IRQCR) 220
Figure 12-15 Reserved Registers 222
Figure 12-16 Port K Data Register (PORTK) 222
Figure 12-17 Port K Data Direction Register (DDRK) 223
Figure 13-1 Breakpoint Block Diagram 237
Figure 13-2 Breakpoint Register Summary 238
Figure 13-3 Breakpoint Control Register 0 (BKPCT0) 239
Figure 13-4 Breakpoint Control Register 1 (BKPCT1) 240
Figure 13-5 Breakpoint First Address Expansion Register (BKP0X) 242

Figure 13-6 Breakpoint First Address High Byte Register (BKP0H) 243
Figure 13-7 Breakpoint First Address Low Byte Register (BKP0L). 243
Figure 13-8 Breakpoint Second Address Expansion Register (BKP1X) . . 244
Figure 13-9 Breakpoint Data High Byte Register (BKP1H). 244
Figure 13-10 Breakpoint Data Low Byte Register (BKP1L) 245
Figure 14-1 BDM Block Diagram 248
Figure 14-2 BDM Register Map Summary 249
Figure 14-3 BDM Status Register (BDMSTS). 250
Figure 14-4 BDM CCR Holding Register (BDMCCR) 252
Figure 14-5 BDM Internal Register Position (BDMINR) 253
Figure 14-11 BDM Instruction Register (BDMIST) 264
Figure 14-12 BDM Instruction Register (BDMIST) 265
Figure 14-13 BDM Shift Register (BDMSHTH). 267
Figure 14-14 BDM Shift Register (BDMSHTL) 267
Figure 14-15 BDM Address Register (BDMADDH). 268
Figure 14-16 BDM Address Register (BDMADDL) 268
Figure 15-1 Security Implementation Block Diagram 280
Figure C-1 Block Diagram of a Fuzzy Logic System 506
Figure C-2 Fuzzification Using Membership Functions 508
Figure C-3 Fuzzy Inference Engine. 511
Figure C-4 Defining a Normal Membership Function 513
Figure C-5 MEM Instruction Flow Diagram 514
Figure C-6 Abnormal Membership Function Case 1 515
Figure C-7 Abnormal Membership Function Case 2 516
Figure C-8 Abnormal Membership Function Case 3 516
Figure C-9 REV Instruction Flow Diagram. 519
Figure C-10 REVW Instruction Flow Diagram 524
Figure C-11 WAV and wavr Instruction Flow Diagram 528
Figure C-12 Endpoint Table Handling 530

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

List of Tables

Table 1-1 Addressing Mode Summary. 28
Table 1-2 Instruction Set Summary 29
Table 1-3 Register and Memory Notation 43
Table 1-4 Source Form Notation 44
Table 1-5 Operation Notation. 45
Table 1-6 Address Mode Notation 45
Table 1-7 Machine Code Notation 46
Table 1-8 Access Detail Notation 47
Table 1-9 Condition Code State Notation 49
Table 2-1 Symbols and Operators 51
Table 3-1 Core Register Map Reference 61
Table 4-1 Addressing Mode Summary. 63
Table 4-2 Summary of Indexed Operations 68
Table 4-3 Load and Store Instructions. 73
Table 4-4 Transfer and Exchange Instructions 74
Table 4-5 Move Instructions. 74
Table 4-6 Add and Subtract Instructions 75
Table 4-7 BCD Instructions 76
Table 4-8 Decrement and Increment Instructions 76
Table 4-9 Compare and Test Instructions 77
Table 4-10 Boolean Logic Instructions. 78
Table 4-11 Clear, Complement, and Negate Instructions 78
Table 4-12 Multiplication and Division Instructions 79
Table 4-13 Bit Test and Bit Manipulation Instructions 79
Table 4-14 Shift and Rotate Instructions 80
Table 4-15 Fuzzy Logic Instructions 81
Table 4-16 Minimum and Maximum Instructions 82
Table 4-17 Multiply and Accumulate Instruction 83
Table 4-18 Table Interpolation Instructions 83
Table 4-19 Short Branch Instructions. 84
Table 4-20 Long Branch Instructions 85
Table 4-21 Bit Condition Branch Instructions. 85
Table 4-22 Loop Primitive Instructions. 86

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 4-23 Jump and Subroutine Instructions 87
Table 4-24 Interrupt Instructions 87
Table 4-25 Index Manipulation Instructions 88
Table 4-26 Stacking Instructions 89
Table 4-27 Load Effective Address Instructions. 90
Table 4-28 Condition Code Instructions. 90
Table 4-29 STOP and WAI Instructions. 91
Table 4-30 Background Mode and Null Operation Instructions 91
Table 5-1 Instruction Set Summary 106
Table 5-2 Register and Memory Notation 120
Table 5-3 Source Form Notation 121
Table 5-4 Operation Notation. 122
Table 5-5 Address Mode Notation 122
Table 5-6 Machine Code Notation 123
Table 5-7 Access Detail Notation 123
Table 5-8 Condition Code State Notation 126
Table 5-9 IPIPE[1:0] Decoding when E Clock is High 127
Table 5-10 IPIPE[1:0] Decoding when E Clock is Low 128
Table 6-1 Exception Vector Map and Priority. 135
Table 6-2 Reset Sources 136
Table 6-3 Interrupt Sources 137
Table 7-1 Core Interface Signal Definitions 142
Table 7-2 Multiplexed Expansion Bus Timing - Preliminary Targets 160
Table 7-3 Expansion Bus Timing - Preliminary Targets. 161
Table 7-4 Access Type vs. Bus Control Pins. 162
Table 8-1 Core Clock and Reset Interface Signals 166
Table 10-1 Exception Vector Map and Priority. 176
Table 11-1 External Stretch Bit Definition 184
Table 11-2 Allocated EEPROM Memory Space 186
Table 11-3 Allocated RAM Memory Space 186
Table 11-4 Allocated Flash EEPROM/ROM Physical Memory Space. . . . 187
Table 11-5 Allocated Off-Chip Memory Options 188
Table 11-6 Program Page Index Register Bits. 189
Table 11-7 Select Signal Priority 190
Table 11-8 Allocated Off-Chip Memory Options 191
Table 11-9 External/Internal Page Window Access. 191

Table 11-100K Byte Physical Flash/ROM Allocated 193
Table 11-1116K Byte Physical Flash/ROM Allocated 193
Table 11-1248K Byte Physical Flash/ROM Allocated 193
Table 11-1364K Byte Physical Flash/ROM Allocated 194
Table 11-14Wide Bus Enable Signal Generation 198
Table 11-15Read Data Bus Swapping 199
Table 12-1 MEBI Interface Signal Definitions 203
Table 12-2 MODC, MODB, MODA Write Capability 216
Table 12-3 Mode Select and State of Mode Bits 216
Table 12-4 External System Pins Associated With MEBI 225
Table 12-5 Access Type vs. Bus Control Pins 227
Table 12-6 Mode Pin Setup and Hold Timing 227
Table 12-7 Peripheral Mode Pin Configuration 232
Table 13-1 Breakpoint Mask Bits for First Address 240
Table 13-2 Breakpoint Mask Bits for Second Address (Dual Mode) 241
Table 13-3 Breakpoint Mask Bits for Data Breakpoints (Full Mode) 241
Table 14-1 Hardware Commands 255
Table 14-2 Firmware Commands. 256
Table 14-3 Tag Pin Function 261
Table 14-4 TTAGO Decoding 265
Table 14-5 RNEXT Decoding. 266
Table 15-1 Security Interface Signal Definitions 280
Table A-1 Parametric Table Format 285
Table A-2 Ratings Table Format 285
Table B-1 Condition Code State Notation 287
Table B-2 Register and Memory Notation 288
Table B-3 Address Mode Notation 289
Table B-4 Operator Notation 289
Table B-5 Machine Code Notation 290
Table B-6 Source Form Notation 291
Table B-7 CPU Cycle Notation 292
Table D-1 Translated M68HC11 Mnemonics 533
Table D-2 Instructions with Smaller Object Code 534
Table D-3 Comparison of Math Instruction Speeds 541
Table D-4 New STAR12 Instructions 542

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 1 Introduction

1.1 Core Overview

The STAR12 V1.5 Core is a 16-bit processing core using the 68HC12 instruction set architecture (ISA). This makes the Core instruction set compatible with currently available Motorola 68HC12 based designs and allows for Motorola 68HC11 source code to be directly accepted by assemblers used for the STAR12 Central Processing Unit (CPU). In addition, the Core contains the Interrupt (INT), Module Mapping Control (MMC), Multiplexed External Bus Interface (MEBI), Breakpoint (BKP) and Background Debug Mode (BDM) sub-blocks providing a tightly coupled structure to maximize execution efficiency for integrating into a System-on-a-Chip (SoC) design. These sub-blocks handle all system interfacing with the Core including interrupt and reset processing, register and memory mapping, memory and peripheral interfacing, external bus control and source code debug for code development. A complete functional description of each sub-blocks is included in later sections of this guide.

1.2 Features

The main features of the Core are:

- High-speed, 16-bit processing with the same programming model and instruction set as the Motorola 68HC12 CPU
- Full 16-bit data paths for efficient arithmetic operation and high-speed mathematical execution
- Allows instructions with odd byte counts, including many single-byte instructions for more efficient use of program memory space
- Three stage instruction queue to buffer program information for more efficient CPU execution
- Extensive set of indexed addressing capabilities including:
  - Using the stack pointer as an indexing register in all indexed operations
  - Using the program counter as an indexing register in all but auto increment/decrement mode
  - Accumulator offsets using A, B or D accumulators
  - Automatic index pre-decrement, pre-increment, post-decrement and post-increment (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
- Provides 2 to 122 I bit maskable interrupt vectors, 1 X bit maskable interrupt vector, 2 nonmaskable CPU interrupt vectors and 3 reset vectors
- Optional register configurable highest priority I bit maskable interrupt
- On-chip memory and peripheral block interfacing with internal memory expansion capability and external data chip select
- Configurable system memory and mapping options

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

- External Bus Interface (8-bit or 16-bit, multiplexed or non-multiplexed)
- Multiple modes of operation
- Hardware breakpoint support for forced or tagged breakpoints with two modes of operation:
  - Dual Address Mode to match on either of two addresses
  - Full Breakpoint Mode to match on address and data combination
- Single-wire background debug system implemented in on-chip hardware
- Secured mode of operation
- Fully synthesizable design
- Single Core clock operation
- Full Mux-D scan test implementation

The STAR12 V1.5 Core is designed to interface with the system peripherals through the use of the I.P. Bus and its interface defined by the Motorola Semiconductor Reuse Standards (MSRS). The Core communicates with the on-chip memory blocks either directly through the Core interface signals or via the STAR bus. Interfacing with memories external to the system is provided for through the MEBI sub-block of the Core and the corresponding port/pad logic it is connected to within the system.

1.3 Block Diagram

A block diagram of the Core within a typical SoC system is given in **Figure 1-1** below. This diagram is a general representation of the Core, its sub-blocks and the interfaces to the rest of the blocks within the SoC design. The signals related to BKGD, Port A, Port B, Port E and Port K are direct interfaces to port/pad logic at the top level of the overall system.

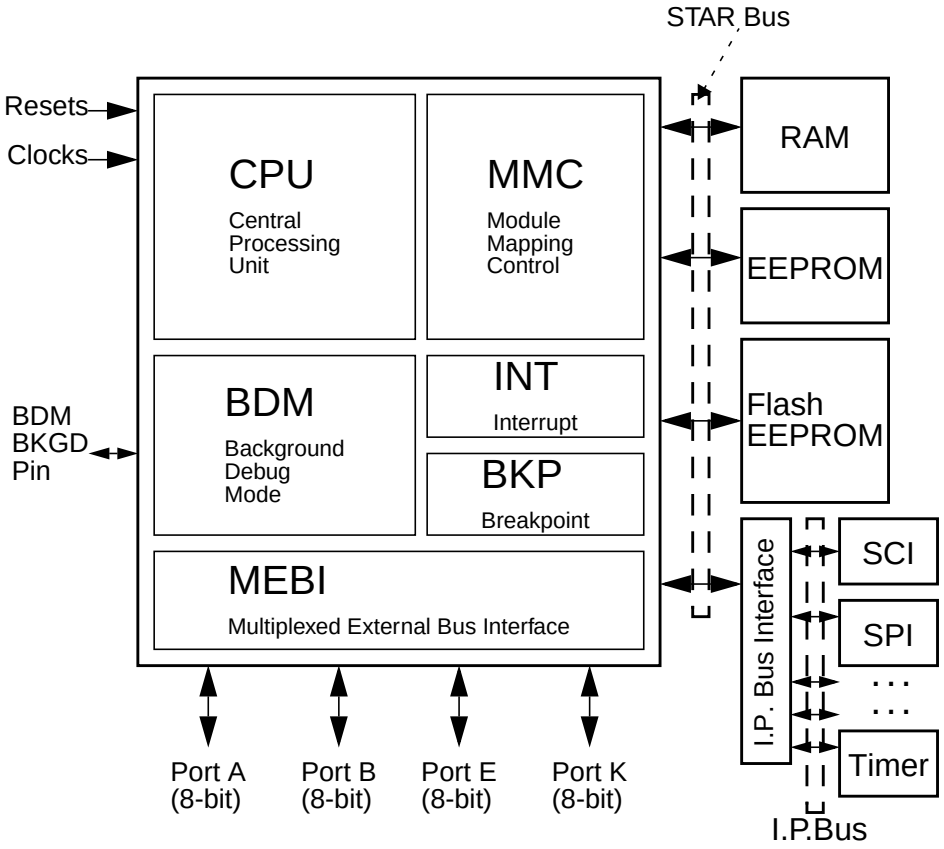


Figure 1-1 Core Block Diagram

The main sub-blocks of the Core are:

- Central Processing Unit (CPU) - 68HC12 ISA compatible
- Interrupt (INT)
- Module Mapping Control (MMC)
- Multiplexed External Bus Interface (MEBI)
- Breakpoint (BKP)
- Background Debug Mode (BDM)

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

1.4 Architectural Summary

As briefly discussed previously, the Core consists of the STAR12 Central Processing Unit (CPU) along with the Interrupt (INT), Module Mapping Control (MMC), Multiplexed External Bus Interface (MEBI), Breakpoint (BKP) and Background Debug Mode (BDM) sub-blocks. The CPU executes the 68HC12 CPU ISA with a three-stage instruction queue to facilitate a high level of code execution efficiency. The INT sub-block interacts with the CPU to provide 2 to 122 I bit maskable (configured at system integration), 1 X bit maskable and 2 nonmaskable CPU interrupt vectors, 3 reset vectors and handles waking-up the system from wait or stop mode due to a serviceable interrupt. The MMC sub-block controls address space mapping and generates memory selects and a single peripheral select (to be decoded by the I.P. Bus) as well as multiplexing the address and data signals for proper interaction with the CPU. The MEBI sub-block functions as the external bus controller with four 8-bit ports (A, B, E and K) as well as handling mode decoding and initialization for the Core. The BKP sub-block serves to assist in debugging of software by providing for hardware breakpoints. The BKP supports dual address and full breakpoint modes for matching on either of two address or on an address and data combination, respectively, to initiate a Software Interrupt (SWI) or put the system into Background Debug Mode. The BKP also supports tagged or forced breakpoints for breaking just before a specific instruction or on the first instruction boundary after a match, respectively. The BDM sub-block provides for a single-wire background debug communication system implemented within the Core with on-chip hardware. The BDM allows for single-wire serial interfacing with a development system host.

The Core is a fully synthesizable single-clock design with full Mux-D scan test implementation. It is designed to be synthesized and timed together as a single block for optimizing speed of execution and minimizing area.

1.5 Programming Model

The STAR12 V1.5 Core CPU12 programming model, shown in **Figure 1-2**, is the same as that of the 68HC12 and 68HC11. For a detailed description of the programming model and associated registers please refer to **Section 3** of this guide.

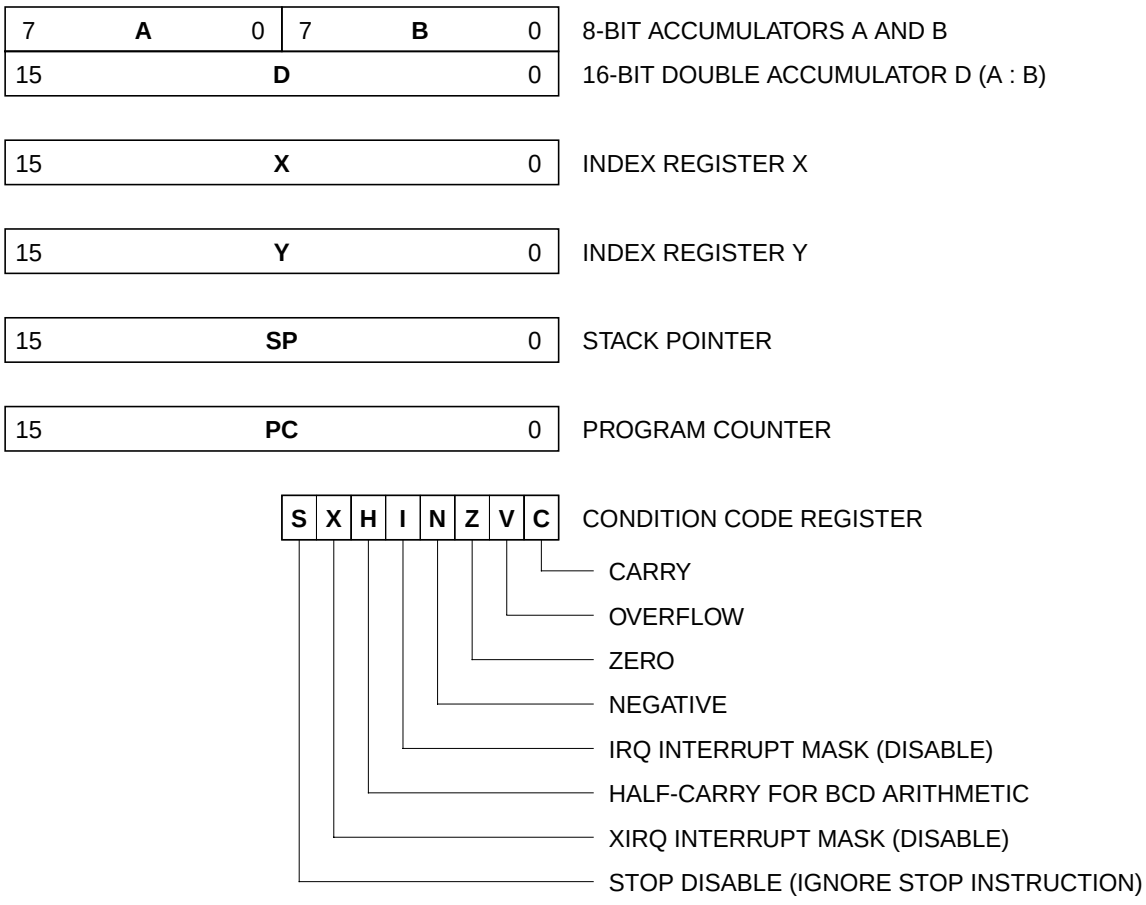


Figure 1-2 Programming Model

1.6 Data Format Summary

Following is a discussion of the data types used and their organization in memory for the Core.

1.6.1 Data Types

The CPU uses the following 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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

- 32-bit signed and unsigned integers

NOTE: *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.

1.6.2 Memory Organization

The standard STAR12 Core address space is 64K bytes. However, the CPU has special instructions to support paged memory expansion which increases the standard area by means of predefined windows within the available address space. See **Section 11 Module Mapping Control (MMC)** for more information.

Eight-bit values can be stored at any odd or even byte address in available memory. Sixteen-bit values occupy two consecutive memory locations; the high byte is in the lowest address, but does not have to be aligned to an even boundary. Thirty-two-bit values occupy four consecutive memory locations; the high byte is in the lowest address, but does not have to be aligned to an even boundary.

All I/O and all on-chip peripherals are memory-mapped. No special instruction syntax is required to access these addresses. On-chip register and memory mapping are determined at the SoC level and are configured during integration of the Core into the system.

1.7 Addressing modes

A summary of the addressing modes used by the Core is given in **Table 1-1** below. The operation of each of these modes is discussed in detail in **Section 4** of this guide.

Table 1-1 Addressing Mode Summary

Addressing Mode	Source Form	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-bit 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	Effective address is the value in PC plus an 8-bit or 16-bit relative offset value.
Indexed (5-bit offset)	INST oprx5,xysp	IDX	Effective address is the value in X, Y, SP, or PC plus a 5-bit signed constant offset.
Indexed (predecrement)	INST oprx3,–xys	IDX	Effective address is the value in X, Y, or SP autodecremented by 1 to 8.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 1-1 Addressing Mode Summary

Addressing Mode	Source Form	Abbreviation	Description
Indexed (preincrement)	INST oprx3,+xys	IDX	Effective address is the value in X, Y, or SP autoincremented by 1 to 8.
Indexed (postdecrement)	INST oprx3,xys–	IDX	Effective address is the value in X, Y, or SP. The value is postdecremented by 1 to 8.
Indexed (postincrement)	INST oprx3,xys+	IDX	Effective address is the value in X, Y, or SP. The value is postincremented by 1 to 8.
Indexed (accumulator offset)	INST abd,xysp	IDX	Effective address is the value in X, Y, SP, or PC plus the value in A, B, or D.
Indexed (9-bit offset)	INST oprx9,xysp	IDX1	Effective address is the value in X, Y, SP, or PC plus a 9-bit signed constant offset.
Indexed (16-bit offset)	INST oprx16,xysp	IDX2	Effective address is the value in X, Y, SP, or PC plus a 16-bit constant offset.
Indexed-indirect (16-bit offset)	INST [oprx16,xysp]	[IDX2]	The value in X, Y, SP, or PC plus a 16-bit constant offset points to the effective address.
Indexed-indirect (D accumulator offset)	INST [D,xysp]	[D,IDX]	The value in X, Y, SP, or PC plus the value in D points to the effective address.

1.8 Instruction Set Overview

All memory and I/O are mapped in a common 64K byte address space, allowing the same set of instructions to access memory, I/O, and control registers. Load, store, transfer, exchange, and move instructions facilitate movement of data to and from memory and peripherals.

There are instructions for signed and unsigned addition, division and multiplication with 8-bit, 16-bit, and some larger operands.

Special arithmetic and logic instructions aid stacking operations, indexing, BCD calculation, and condition code register manipulation. There are also dedicated instructions for multiply and accumulate operations, table interpolation, and specialized mathematical calculations for fuzzy logic operations.

A summary of the CPU instruction set is given in **Table 1-2** below. A detailed overview of the entire instruction set is covered in **Section 4** of this guide along with an instruction-by-instruction detailed description in **Appendix B**.

Table 1-2 Instruction Set Summary

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ABA	Add B to A; (A)+(B)⇒A	INH	18 06	OO	– – Δ – Δ Δ Δ Δ
ABXSame as LEAX B,X	Add B to X; (X)+(B)⇒X	IDX	1A E5	Pf	– – – – – – – –
ABYSame as LEAY B,Y	Add B to Y; (Y)+(B)⇒Y	IDX	19 ED	Pf	– – – – – – – –
ADCA #*opr8i* ADCA *opr8a* ADCA *opr16a* ADCA *oprx0_xysppc* ADCA *oprx9,xysppc* ADCA *oprx16,xysppc* ADCA [D,*xysppc*] ADCA [*oprx16,xysppc*]	Add with carry to A; (A)+(M)+C⇒A or (A)+imm+C⇒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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – Δ – Δ Δ Δ Δ

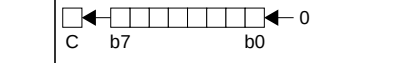
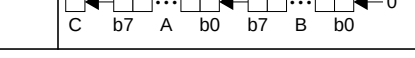
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

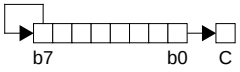
Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ADCB #*opr8i* ADCB *opr8a* ADCB *opr16a* ADCB *oprx0_xysppc* ADCB *oprx9,xysppc* ADCB *oprx16,xysppc* ADCB [D,*xysppc*] ADCB [*oprx16,xysppc*]	Add with carry to B; (B)+(M)+C⇒B or (B)+imm+C⇒B	IMM DIR EXT IDX IDX1 IDX2 [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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – Δ – Δ Δ Δ Δ

ADDA #*opr8i*
ADDA *opr8a*
ADDA *opr16a*
ADDA *oprx0_xysppc*
ADDA *oprx9,xysppc*
ADDA *oprx16,xysppc*
ADDA [D,*xysppc*]
ADDA [*oprx16,xysppc*]
	Add to A; (A)+(M)⇒A
	or (A)+imm⇒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
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – Δ – Δ Δ Δ Δ

ADDB #*opr8i*
ADDB *opr8a*
ADDB *opr16a*
ADDB *oprx0_xysppc*
ADDB *oprx9,xysppc*
ADDB *oprx16,xysppc*
ADDB [D,*xysppc*]
ADDB [*oprx16,xysppc*]
	Add to B; (B)+(M)⇒B
	or (B)+imm⇒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
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – Δ – Δ Δ Δ Δ

ADDD #*opr16i*
ADDD *opr8a*
ADDD *opr16a*
ADDD *oprx0_xysppc*
ADDD *oprx9,xysppc*
ADDD *oprx16,xysppc*
ADDD [D,*xysppc*]
ADDD [*oprx16,xysppc*]
	Add to D; (A:B)+(M:M+1)⇒A:B
	or (A:B)+imm⇒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
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ Δ Δ

ANDA #*opr8i* ANDA *opr8a* ANDA *opr16a* ANDA *oprx0_xysppc* ANDA *oprx9,xysppc* ANDA *oprx16,xysppc* ANDA [D,*xysppc*] ANDA [*oprx16,xysppc*]	AND with A; (A)•(M)⇒A or (A)•imm⇒A	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ANDB #*opr8i* ANDB *opr8a* ANDB *opr16a* ANDB *oprx0_xysppc* ANDB *oprx9,xysppc* ANDB *oprx16,xysppc* ANDB [D,*xysppc*] ANDB [*oprx16,xysppc*]	AND with B; (B)•(M)⇒B or (B)•imm⇒B	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ANDCC #*opr8i*	AND with CCR; (CCR)•imm⇒CCR	IMM	10 ii	P	⇓ ⇓ ⇓ ⇓ ⇓ ⇓ ⇓ ⇓
ASL *opr16a*Same as LSL ASL *oprx0_xysp* ASL *oprx9,xysppc* ASL *oprx16,xysppc* ASL [D,*xysppc*] ASL [*oprx16,xysppc*] ASLASame as LSLA ASLBSame as LSLB	Arithmetic shift left M  Arithmetic shift left A Arithmetic shift left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ Δ Δ
ASLDSame as LSLD	Arithmetic shift left D  	INH	59	0	– – – – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ASR *opr16a* ASR *oprx0_xysppc* ASR *oprx9,xysppc* ASR *oprx16,xysppc* ASR [D,*xysppc*] ASR [*oprx16,xysppc*] ASRA ASRB	Arithmetic shift right M  Arithmetic shift right A Arithmetic shift right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	77 hh ll 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
BCC *rel8*Same as BHS	Branch if C clear; if C=0, then (PC)+2+rel⇒PC	REL	24 rr	PPP (branch) P (no branch)	– – – – – – – –
BCLR *opr8a, msk8* BCLR *opr16a, msk8* BCLR *oprx0_xysppc, msk8* BCLR *oprx9,xysppc, msk8* BCLR *oprx16,xysppc, msk8*	Clear bit(s) in M; (M)•mask byte⇒M	DIR EXT IDX IDX1 IDX2	4D dd mm 1D hh ll mm 0D xb mm 0D xb ff mm 0D xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	– – – – Δ Δ 0 –
BCS *rel8*Same as BLO	Branch if C set; if C=1, then (PC)+2+rel⇒PC	REL	25 rr	PPP (branch) P (no branch)	– – – – – – – –
BEQ *rel8*	Branch if equal; if Z=1, then (PC)+2+rel⇒PC	REL	27 rr	PPP (branch) P (no branch)	– – – – – – – –
BGE *rel8*	Branch if ≥ 0, signed; if N⊕V=0, then (PC)+2+rel⇒PC	REL	2C rr	PPP (branch) P (no branch)	– – – – – – – –
BGND	Enter background debug mode	INH	00	VfPPP	– – – – – – – –
BGT *rel8*	Branch if > 0, signed; if Z | (N⊕V)=0, then (PC)+2+rel⇒PC	REL	2E rr	PPP (branch) P (no branch)	– – – – – – – –
BHI *rel8*	Branch if higher, unsigned; if C | Z=0, then (PC)+2+rel⇒PC	REL	22 rr	PPP (branch) P (no branch)	– – – – – – – –
BHS *rel8*Same as BCC	Branch if higher or same, unsigned; if C=0, then (PC)+2+rel⇒PC	REL	24 rr	PPP (branch) P (no branch)	– – – – – – – –
BITA #*opr8i* BITA *opr8a* BITA *opr16a* BITA *oprx0_xysppc* BITA *oprx9,xysppc* BITA *oprx16,xysppc* BITA [D,*xysppc*] BITA [*oprx16,xysppc*]	Bit test A; (A)•(M) or (A)•imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	85 ii 95 dd B5 hh ll A5 xb A5 xb ff A5 xb ee ff A5 xb A5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
BITB #*opr8i* BITB *opr8a* BITB *opr16a* BITB *oprx0_xysppc* BITB *oprx9,xysppc* BITB *oprx16,xysppc* BITB [D,*xysppc*] BITB [*oprx16,xysppc*]	Bit test B; (B)•(M) or (B)•imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C5 ii D5 dd F5 hh ll E5 xb E5 xb ff E5 xb ee ff E5 xb E5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
BLE *rel8*	Branch if ≤ 0, signed; if Z | (N⊕V)=1, then (PC)+2+rel⇒PC	REL	2F rr	PPP (branch) P (no branch)	– – – – – – – –
BLO *rel8*Same as BCS	Branch if lower, unsigned; if C=1, then (PC)+2+rel⇒PC	REL	25 rr	PPP (branch) P (no branch)	– – – – – – – –
BLS *rel8*	Branch if lower or same, unsigned; if C | Z=1, then (PC)+2+rel⇒PC	REL	23 rr	PPP (branch) P (no branch)	– – – – – – – –
BLT *rel8*	Branch if < 0, signed; if N⊕V=1, then (PC)+2+rel⇒PC	REL	2D rr	PPP (branch) P (no branch)	– – – – – – – –
BMI *rel8*	Branch if minus; if N=1, then (PC)+2+rel⇒PC	REL	2B rr	PPP (branch) P (no branch)	– – – – – – – –
BNE *rel8*	Branch if not equal to 0; if Z=0, then (PC)+2+rel⇒PC	REL	26 rr	PPP (branch) P (no branch)	– – – – – – – –
BPL *rel8*	Branch if plus; if N=0, then (PC)+2+rel⇒PC	REL	2A rr	PPP (branch) P (no branch)	– – – – – – – –
BRA *rel8*	Branch always	REL	20 rr	PPP	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
BRCLR *oprx8a*, *msk8*, *rel8* BRCLR *oprx16a*, *msk8*, *rel8* BRCLR *oprx0_xysppc*, *msk8*, *rel8* BRCLR *oprx9,xysppc*, *msk8*, *rel8* BRCLR *oprx16,xysppc*, *msk8*, *rel8*	Branch if bit(s) clear; if (M)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4F dd mm rr 1F hh ll mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BRN *rel8*	Branch never	REL	21 rr	P	– – – – – – – –
BRSET *oprx8*, *msk8*, *rel8* BRSET *oprx16a*, *msk8*, *rel8* BRSET *oprx0_xysppc*, *msk8*, *rel8* BRSET *oprx9,xysppc*, *msk8*, *rel8* BRSET *oprx16,xysppc*, *msk8*, *rel8*	Branch if bit(s) set; if ($\overline{M}$)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4E dd mm rr 1E hh ll mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BSET *oprx8*, *msk8* BSET *oprx16a*, *msk8* BSET *oprx0_xysppc*, *msk8* BSET *oprx9,xysppc*, *msk8* BSET *oprx16,xysppc*, *msk8*	Set bit(s) in M (M) | mask byte⇒M	DIR EXT IDX IDX1 IDX2	4C dd mm 1C hh ll mm 0C xb mm 0C xb ff mm 0C xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	– – – – Δ Δ 0 –
BSR *rel8*	Branch to subroutine; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (PC)+2+rel⇒PC	REL	07 rr	SPPP	– – – – – – – –
BVC *rel8*	Branch if V clear; if V=0, then (PC)+2+rel⇒PC	REL	28 rr	PPP (branch) P (no branch)	– – – – – – – –
BVS *rel8*	Branch if V set; if V=1, then (PC)+2+rel⇒PC	REL	29 rr	PPP (branch) P (no branch)	– – – – – – – –
CALL *opr16a*, *page* CALL *oprx0_xysppc*, *page* CALL *oprx9,xysppc*, *page* CALL *oprx16,xysppc*, *page* CALL [D,*xysppc*] CALL [*oprx16*, *xysppc*]	Call subroutine in expanded memory (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (PPG)⇒M_{SP} pg⇒PPAGE register subroutine address⇒PC	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	4A hh ll pg 4B xb pg 4B xb ff pg 4B xb ee ff pg 4B xb 4B xb ee ff	gnSsPPP gnSsPPP gnSsPPP fgnSsPPP fIignSsPPP fIignSsPPP	– – – – – – – –
CBA	Compare A to B; (A)–(B)	INH	18 17	OO	– – – – Δ Δ Δ Δ
CLCSame as ANDCC #$FE	Clear C bit	IMM	10 FE	P	– – – – – – – 0
CLISame as ANDCC #$EF	Clear I bit	IMM	10 EF	P	– – – 0 – – – –
CLR *opr16a* CLR *oprx0_xysppc* CLR *oprx9,xysppc* CLR *oprx16,xysppc* CLR [D,*xysppc*] CLR [*oprx16,xysppc*] CLRA CLRB	Clear M; $00⇒M Clear A; $00⇒A Clear B; $00⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	79 hh ll 69 xb 69 xb ff 69 xb ee ff 69 xb 69 xb ee ff 87 C7	PwO Pw PwO PwP PIfw PIPw O O	– – – – 0 1 0 0
CLVSame as ANDCC #$FD	Clear V	IMM	10 FD	P	– – – – – – 0 –

CMPA #*opr8i*
CMPA *opr8a*
CMPA *opr16a*
CMPA *oprx0_xysppc*
CMPA *oprx9,xysppc*
CMPA *oprx16,xysppc*
CMPA [D,*xysppc*]
CMPA [*oprx16,xysppc*]
	Compare A
	(A)–(M) or (A)–imm
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			81 ii
			91 dd
			B1 hh ll
			A1 xb
			A1 xb ff
			A1 xb ee ff
			A1 xb
			A1 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

CMPB #*opr8i*
CMPB *opr8a*
CMPB *opr16a*
CMPB *oprx0_xysppc*
CMPB *oprx9,xysppc*
CMPB *oprx16,xysppc*
CMPB [D,*xysppc*]
CMPB [*oprx16,xysppc*]
	Compare B
	(B)–(M) or (B)–imm
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C1 ii
			D1 dd
			F1 hh ll
			E1 xb
			E1 xb ff
			E1 xb ee ff
			E1 xb
			E1 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
COM *opr16a* COM *oprx0_xysppc* COM *oprx9,xysppc* COM *oprx16,xysppc* COM [D,*xysppc*] COM [*oprx16,xysppc*] COMA COMB	Complement M; ($\overline{M}$)=$FF–(M)⇒M Complement A; ($\overline{A}$)=$FF–(A)⇒A Complement B; ($\overline{B}$)=$FF–(B)⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	71 hh ll 61 xb 61 xb ff 61 xb ee ff 61 xb 61 xb ee ff 41 51	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ 0 1
CPD #*opr16i* CPD *opr8a* CPD *opr16a* CPD *oprx0_xysppc* CPD *oprx9,xysppc* CPD *oprx16,xysppc* CPD [D,*xysppc*] CPD [*oprx16,xysppc*]	Compare D (A:B)–(M:M+1) or (A:B)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8C jj kk 9C dd BC hh ll AC xb AC xb ff AC xb ee ff AC xb AC xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPS #*opr16i* CPS *opr8a* CPS *opr16a* CPS *oprx0_xysppc* CPS *oprx9,xysppc* CPS *oprx16,xysppc* CPS [D,*xysppc*] CPS [*oprx16,xysppc*]	Compare SP (SP)–(M:M+1) or (SP)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8F jj kk 9F dd BF hh ll AF xb AF xb ff AF xb ee ff AF xb AF xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPX #*opr16i* CPX *opr8a* CPX *opr16a* CPX *oprx0_xysppc* CPX *oprx9,xysppc* CPX *oprx16,xysppc* CPX [D,*xysppc*] CPX [*oprx16,xysppc*]	Compare X (X)–(M:M+1) or (X)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8E jj kk 9E dd BE hh ll AE xb AE xb ff AE xb ee ff AE xb AE xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPY #*opr16i* CPY *opr8a* CPY *opr16a* CPY *oprx0_xysppc* CPY *oprx9,xysppc* CPY *oprx16,xysppc* CPY [D,*xysppc*] CPY [*oprx16,xysppc*]	Compare Y (Y)–(M:M+1) or (Y)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8D jj kk 9D dd BD hh ll AD xb AD xb ff AD xb ee ff AD xb AD xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
DAA	Decimal adjust A for BCD	INH	18 07	0f0	– – – – Δ Δ ? Δ
DBEQ *abdxysp*, *rel9*	Decrement and branch if equal to 0 (counter)–1⇒counter if (counter)=0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PP0 (no branch)	– – – – – – – –
DBNE *abdxysp*, *rel9*	Decrement and branch if not equal to 0; (counter)–1⇒counter; if (counter)≠0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PP0 (no branch)	– – – – – – – –
DEC *opr16a* DEC *oprx0_xysppc* DEC *oprx9,xysppc* DEC *oprx16,xysppc* DEC [D,*xysppc*] DEC [*oprx16,xysppc*] DECA DECB	Decrement M; (M)–1⇒M Decrement A; (A)–1⇒A Decrement B; (B)–1⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	73 hh ll 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff 43 53	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ Δ –
DESSame as LEAS –1,SP	Decrement SP; (SP)–1⇒SP	IDX	1B 9F	Pf	– – – – – – – –
DEX	Decrement X; (X)–1⇒X	INH	09	0	– – – – – Δ – –
DEY	Decrement Y; (Y)–1⇒Y	INH	03	0	– – – – – Δ – –
EDIV	Extended divide, unsigned; 32 by 16 to 16-bit; (Y:D)÷(X)⇒Y; remainder⇒D	INH	11	fffffffff0	– – – – Δ Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
EDIVS	Extended divide,signed; 32 by 16 to 16-bit; (Y:D)÷(X)⇒Y remainder⇒D	INH	18 14	OfffffffffO	– – – – Δ Δ Δ Δ
EMACS *opr16a*	Extended multiply and accumulate, signed; $(M_X:M_{X+1})\times(M_Y:M_{Y+1})+$ (M~M+3)⇒M~M+3; 16 by 16 to 32-bit	Special	18 12 hh ll	ORROfffRRfWWP	– – – – Δ Δ Δ Δ
EMAXD *oprx0_xysppc* EMAXD *oprx9,xysppc* EMAXD *oprx16,xysppc* EMAXD [D,*xysppc*] EMAXD [*oprx16,xysppc*]	Extended maximum in D; put larger of 2 unsigned 16-bit values in D MAX[(D), (M:M+1)]⇒D N, Z, V, C bits reflect result of internal compare [(D)–(M:M+1)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1A xb 18 1A xb ff 18 1A xb ee ff 18 1A xb 18 1A xb ee ff	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – – Δ Δ Δ Δ
EMAXM *oprx0_xysppc* EMAXM *oprx9,xysppc* EMAXM *oprx16,xysppc* EMAXM [D,*xysppc*] EMAXM [*oprx16,xysppc*]	Extended maximum in M; put larger of 2 unsigned 16-bit values in M MAX[(D), (M:M+1)]⇒M:M+1 N, Z, V, C 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_xysppc* EMIND *oprx9,xysppc* EMIND *oprx16,xysppc* EMIND [D,*xysppc*] EMIND [*oprx16,xysppc*]	Extended minimum in D; put smaller of 2 unsigned 16-bit values in D MIN[(D), (M:M+1)]⇒D N, Z, V, C bits reflect result of internal compare [(D)–(M:M+1)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1B xb 18 1B xb ff 18 1B xb ee ff 18 1B xb 18 1B xb ee ff	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – – Δ Δ Δ Δ
EMINM *oprx0_xysppc* EMINM *oprx9,xysppc* EMINM *oprx16,xysppc* EMINM [D,*xysppc*] EMINM [*oprx16,xysppc*]	Extended minimum in M; put smaller of 2 unsigned 16-bit values in M MIN[(D), (M:M+1)]⇒M:M+1 N, Z, V, C 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	Extended multiply, unsigned (D)×(Y)⇒Y:D; 16 by 16 to 32-bit	INH	13	ffO	– – – – Δ Δ – Δ
EMULS	Extended multiply, signed (D)×(Y)⇒Y:D; 16 by 16 to 32-bit	INH	18 13	OfO OffO (if followed by page 2 instruction)	– – – – Δ Δ – Δ
EORA #*opr8i* EORA *opr8a* EORA *opr16a* EORA *oprx0_xysppc* EORA *oprx9,xysppc* EORA *oprx16,xysppc* EORA [D,*xysppc*] EORA [*oprx16,xysppc*]	Exclusive OR A (A)⊕(M)⇒A or (A)⊕imm⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	88 ii 98 dd B8 hh ll A8 xb A8 xb ff A8 xb ee ff A8 xb A8 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
EORB #*opr8i* EORB *opr8a* EORB *opr16a* EORB *oprx0_xysppc* EORB *oprx9,xysppc* EORB *oprx16,xysppc* EORB [D,*xysppc*] EORB [*oprx16,xysppc*]	Exclusive OR B (B)⊕(M)⇒B or (B)⊕imm⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C8 ii D8 dd F8 hh ll E8 xb E8 xb ff E8 xb ee ff E8 xb E8 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ETBL *oprx0_xysppc*	Extended table lookup and interpolate, 16-bit; (M:M+1)+ [(B)×((M+2:M+3)–(M:M+1))]⇒D	IDX	18 3F xb	ORRffffffP	– – – – Δ Δ – Δ
Before executing ETBL, initialize B with fractional part of lookup value; initialize index register to point to first table entry (M:M+1). No extensions or indirect addressing allowed.					
EXG *abcdxysp,abcdxysp*	Exchange register contents (r1)⇔(r2) r1 and r2 same size $00:(r1)⇒r2r1=8-bit; r2=16-bit $(r1_L)⇔(r2)r1=16-bit; r2=8-bit	INH	B7 eb	P	– – – – – – – –
FDIV	Fractional divide; (D)÷(X)⇒X remainder⇒D; 16 by 16-bit	INH	18 11	OfffffffffO	– – – – Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
IBEQ *abdxysp*, *rel9*	Increment and branch if equal to 0 (counter)+1⇒counter If (counter)=0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
IBNE *abdxysp*, *rel9*	Increment and branch if not equal to 0 (counter)+1⇒counter If (counter)≠0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
IDIV	Integer divide, unsigned; (D)÷(X)⇒X Remainder⇒D; 16 by 16-bit	INH	18 10	Offfffffffo	– – – – – Δ 0 Δ
IDIVS	Integer divide, signed; (D)÷(X)⇒X Remainder⇒D; 16 by 16-bit	INH	18 15	Offfffffffo	– – – – Δ Δ Δ Δ
INC *opr16a* INC *oprx0_xysppc* INC *oprx9,xysppc* INC *oprx16,xysppc* INC [D,*xysppc*] INC [*oprx16,xysppc*] INCA INCB	Increment M; (M)+1⇒M Increment A; (A)+1⇒A Increment B; (B)+1⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	72 hh ll 62 xb 62 xb ff 62 xb ee ff 62 xb 62 xb ee ff 42 52	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ –
INSSame as LEAS 1,SP	Increment SP; (SP)+1⇒SP	IDX	1B 81	Pf	– – – – – – – –
INX	Increment X; (X)+1⇒X	INH	08	O	– – – – – Δ – –
INY	Increment Y; (Y)+1⇒Y	INH	02	O	– – – – – Δ – –
JMP *opr16a* JMP *oprx0_xysppc* JMP *oprx9,xysppc* JMP *oprx16,xysppc* JMP [D,*xysppc*] JMP [*oprx16,xysppc*]	Jump Subroutine address⇒PC	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	06 hh ll 05 xb 05 xb ff 05 xb ee ff 05 xb 05 xb ee ff	PPP PPP PPP fPPP fIfPPP fIfPPP	– – – – – – – –
JSR *opr8a* JSR *opr16a* JSR *oprx0_xysppc* JSR *oprx9,xysppc* JSR *oprx16,xysppc* JSR [D,*xysppc*] JSR [*oprx16,xysppc*]	Jump to subroutine (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} Subroutine address⇒PC	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	17 dd 16 hh ll 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff	SPPP SPPP PPPS PPPS fPPPS fIfPPPS fIfPPPS	– – – – – – – –
LBCC *rel16*Same as LBHS	Long branch if C clear; if C=0, then (PC)+4+rel⇒PC	REL	18 24 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBCS *rel16*Same as LBLO	Long branch if C set; if C=1, then (PC)+4+rel⇒PC	REL	18 25 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBEQ *rel16*	Long branch if equal; if Z=1, then (PC)+4+rel⇒PC	REL	18 27 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBGE *rel16*	Long branch if ≥ 0, signed If N⊕V=0, then (PC)+4+rel⇒PC	REL	18 2C qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBGT *rel16*	Long branch if > 0, signed If Z | (N⊕V)=0, then (PC)+4+rel⇒PC	REL	18 2E qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBHI *rel16*	Long branch if higher, unsigned If C | Z=0, then (PC)+4+rel⇒PC	REL	18 22 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBHS *rel16*Same as LBCC	Long branch if higher or same, unsigned; If C=0, (PC)+4+rel⇒PC	REL	18 24 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLE *rel16*	Long branch if ≤ 0, signed; if Z | (N⊕V)=1, then (PC)+4+rel⇒PC	REL	18 2F qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLO *rel16*Same as LBCS	Long branch if lower, unsigned; if C=1, then (PC)+4+rel⇒PC	REL	18 25 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLS *rel16*	Long branch if lower or same, unsigned; If C | Z=1, then (PC)+4+rel⇒PC	REL	18 23 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLT *rel16*	Long branch if < 0, signed If N⊕V=1, then (PC)+4+rel⇒PC	REL	18 2D qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
LBMI *rel16*	Long branch if minus If N=1, then (PC)+4+rel⇒PC	REL	18 2B qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBNE *rel16*	Long branch if not equal to 0 If Z=0, then (PC)+4+rel⇒PC	REL	18 26 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBPL *rel16*	Long branch if plus If N=0, then (PC)+4+rel⇒PC	REL	18 2A qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBRA *rel16*	Long branch always	REL	18 20 qq rr	OPPP	– – – – – – – –
LBRN *rel16*	Long branch never	REL	18 21 qq rr	OPO	– – – – – – – –
LBVC *rel16*	Long branch if V clear If V=0,then (PC)+4+rel⇒PC	REL	18 28 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBVS *rel16*	Long branch if V set If V=1,then (PC)+4+rel⇒PC	REL	18 29 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –

LDAA #*opr8i*
LDAA *opr8a*
LDAA *opr16a*
LDAA *oprx0_xysppc*
LDAA *oprx9,xysppc*
LDAA *oprx16,xysppc*
LDAA [D,*xysppc*]
LDAA [*oprx16,xysppc*]
	Load A
	(M)⇒A
	or imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			86 ii
			96 dd
			B6 hh ll
			A6 xb
			A6 xb ff
			A6 xb ee ff
			A6 xb
			A6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ 0 –

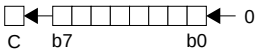
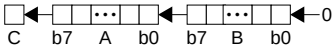
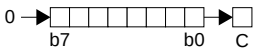
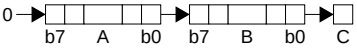
LDAB #*opr8i*
LDAB *opr8a*
LDAB *opr16a*
LDAB *oprx0_xysppc*
LDAB *oprx9,xysppc*
LDAB *oprx16,xysppc*
LDAB [D,*xysppc*]
LDAB [*oprx16,xysppc*]
	Load B
	(M)⇒B
	or imm⇒B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C6 ii
			D6 dd
			F6 hh ll
			E6 xb
			E6 xb ff
			E6 xb ee ff
			E6 xb
			E6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ 0 –

LDD #*opr16i*
LDD *opr8a*
LDD *opr16a*
LDD *oprx0_xysppc*
LDD *oprx9,xysppc*
LDD *oprx16,xysppc*
LDD [D,*xysppc*]
LDD [*oprx16,xysppc*]
	Load D
	(M:M+1)⇒A:B
	or imm⇒A:B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CC jj kk
			DC dd
			FC hh ll
			EC xb
			EC xb ff
			EC xb ee ff
			EC xb
			EC xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

LDS #*opr16i*
LDS *opr8a*
LDS *opr16a*
LDS *oprx0_xysppc*
LDS *oprx9,xysppc*
LDS *oprx16,xysppc*
LDS [D,*xysppc*]
LDS [*oprx16,xysppc*]
	Load SP
	(M:M+1)⇒SP
	or imm⇒SP
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CF jj kk
			DF dd
			FF hh ll
			EF xb
			EF xb ff
			EF xb ee ff
			EF xb
			EF xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

LDX #*opr16i*
LDX *opr8a*
LDX *opr16a*
LDX *oprx0_xysppc*
LDX *oprx9,xysppc*
LDX *oprx16,xysppc*
LDX [D,*xysppc*]
LDX [*oprx16,xysppc*]
	Load X
	(M:M+1)⇒X
	or imm⇒X
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CE jj kk
			DE dd
			FE hh ll
			EE xb
			EE xb ff
			EE xb ee ff
			EE xb
			EE xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

LDY #*opr16i*
LDY *opr8a*
LDY *opr16a*
LDY *oprx0_xysppc*
LDY *oprx9,xysppc*
LDY *oprx16,xysppc*
LDY [D,*xysppc*]
LDY [*oprx16,xysppc*]
	Load Y
	(M:M+1)⇒Y
	or imm⇒Y
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CD jj kk
			DD dd
			FD hh ll
			ED xb
			ED xb ff
			ED xb ee ff
			ED xb
			ED xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
LEAS *oprx0_xysppc* LEAS *oprx9,xysppc* LEAS *oprx16,xysppc*	Load effective address into SP EA⇒SP	IDX IDX1 IDX2	1B xb 1B xb ff 1B xb ee ff	Pf PO PP	– – – – – – – –
LEAX *oprx0_xysppc* LEAX *oprx9,xysppc* LEAX *oprx16,xysppc*	Load effective address into X EA⇒X	IDX IDX1 IDX2	1A xb 1A xb ff 1A xb ee ff	Pf PO PP	– – – – – – – –
LEAY *oprx0_xysppc* LEAY *oprx9,xysppc* LEAY *oprx16,xysppc*	Load effective address into Y EA⇒Y	IDX IDX1 IDX2	19 xb 19 xb ff 19 xb ee ff	Pf PO PP	– – – – – – – –
LSL *opr16a*Same as ASL LSL *oprx0_xysppc* LSL *oprx9,xysppc* LSL *oprx16,xysppc* LSL [D,*xysppc*] LSL [*oprx16,xysppc*] LSLASame as ASLA LSLBSame as ASLB	Logical shift left M  Logical shift left A Logical shift left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
LSLDSame as ASLD	Logical shift left D  	INH	59	O	– – – – Δ Δ Δ Δ
LSR *opr16a* LSR *oprx0_xysppc* LSR *oprx9,xysppc* LSR *oprx16,xysppc* LSR [D,*xysppc*] LSR [*oprx16,xysppc*] LSRA LSRB	Logical shift right M  Logical shift right A Logical shift right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	74 hh ll 64 xb 64 xb ff 64 xb ee ff 64 xb 64 xb ee ff 44 54	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – 0 Δ Δ Δ
LSRD	Logical shift right D  	INH	49	O	– – – – 0 Δ Δ Δ
MAXA *oprx0_xysppc* MAXA *oprx9,xysppc* MAXA *oprx16,xysppc* MAXA [D,*xysppc*] MAXA [*oprx16,xysppc*]	Maximum in A; put larger of 2 unsigned 8-bit values in A MAX[(A), (M)]⇒A N, Z, V, C 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	OrPf OrPO OfrPP OfIfrPf OfIPrPf	– – – – Δ Δ Δ Δ
MAXM *oprx0_xysppc* MAXM *oprx9,xysppc* MAXM *oprx16,xysppc* MAXM [D,*xysppc*] MAXM [*oprx16,xysppc*]	Maximum in M; put larger of 2 unsigned 8-bit values in M MAX[(A), (M)]⇒M N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1C xb 18 1C xb ff 18 1C xb ee ff 18 1C xb 18 1C xb ee ff	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	– – – – Δ Δ Δ Δ
MEM	Determine grade of membership; μ (grade)⇒M_Y; (X)+4⇒X; (Y)+1⇒Y If (A)<P1 or (A)>P2, then μ=0; else μ= MIN[((A)–P1)×S1, (P2–(A))×S2, $FF] (A)=current crisp input value; X points at 4 data bytes (P1, P2, S1, S2) of a trapezoidal membership function; Y points at fuzzy input (RAM location)	Special	01	RRfOw	– – ? – ? ? ? ?
MINA *oprx0_xysppc* MINA *oprx9,xysppc* MINA *oprx16,xysppc* MINA [D,*xysppc*] MINA [*oprx16,xysppc*]	Minimum in A; put smaller of 2 unsigned 8-bit values in A MIN[(A), (M)]⇒A N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 19 xb 18 19 xb ff 18 19 xb ee ff 18 19 xb 18 19 xb ee ff	OrPf OrPO OfrPP OfIfrPf OfIPrPf	– – – – Δ Δ Δ Δ
MINM *oprx0_xysppc* MINM *oprx9,xysppc* MINM *oprx16,xysppc* MINM [D,*xysppc*] MINM [*oprx16,xysppc*]	Minimum in N; put smaller of two unsigned 8-bit values in M MIN[(A), (M)]⇒M N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1D xb 18 1D xb ff 18 1D xb ee ff 18 1D xb 18 1D xb ee ff	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	– – – – Δ Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

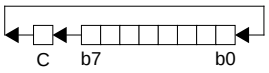
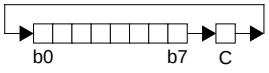
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C

MOVB #*opr8*, *opr16a*
MOVB #*opr8i*, *oprx0_xysppc*
MOVB *opr16a*, *opr16a*
MOVB *opr16a*, *oprx0_xysppc*
MOVB *oprx0_xysppc*, *opr16a*
MOVB *oprx0_xysppc*, *oprx0_xysppc*
	Move byte
	Memory-to-memory 8-bit byte-move
	$(M_1)\Rightarrow M_2$
	First operand specifies byte to move
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			18 0B ii hh ll
			18 08 xb ii
			18 0C hh ll hh ll
			18 09 xb hh ll
			18 0D xb hh ll
			18 0A xb xb
				OPwP
				OPwO
				OrPwPO
				OPrPw
				OrPwP
				OrPwO
					– – – – – – – –

MOVW #*oprx16*, *opr16a*
MOVW #*opr16i*, *oprx0_xysppc*
MOVW *opr16a*, *opr16a*
MOVW *opr16a*, *oprx0_xysppc*
MOVW *oprx0_xysppc*, *opr16a*
MOVW *oprx0_xysppc*, *oprx0_xysppc*
	Move word
	Memory-to-memory 16-bit word-move
	$(M_1{:}M_1{+}1)\Rightarrow M_2{:}M_2{+}1$
	First operand specifies word to move
		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	Multiply, unsigned (A)×(B)⇒A:B; 8 by 8-bit	INH	12	0	– – – – – – – Δ
NEG *opr16a* NEG *oprx0_xysppc* NEG *oprx9*,*xysppc* NEG *oprx16*,*xysppc* NEG [D,*xysppc*] NEG [*oprx16*,*xysppc*] NEGA NEGB	Negate M; 0–(M)⇒M or $(\overline{M})+1\Rightarrow M$ Negate A; 0–(A)⇒A or $(\overline{A})+1\Rightarrow A$ Negate B; 0–(B)⇒B or $(\overline{B})+1\Rightarrow B$	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	70 hh ll 60 xb 60 xb ff 60 xb ee ff 60 xb 60 xb ee ff 40 50	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
NOP	No operation	INH	A7	0	– – – – – – – –
ORAA #*opr8i* ORAA *opr8a* ORAA *opr16a* ORAA *oprx0_xysppc* ORAA *oprx9*,*xysppc* ORAA *oprx16*,*xysppc* ORAA [D,*xysppc*] ORAA [*oprx16*,*xysppc*]	OR accumulator A (A) | (M)⇒A or (A) | imm⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8A ii 9A dd BA hh ll AA xb AA xb ff AA xb ee ff AA xb AA xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ORAB #*opr8i* ORAB *opr8a* ORAB *opr16a* ORAB *oprx0_xysppc* ORAB *oprx9*,*xysppc* ORAB *oprx16*,*xysppc* ORAB [D,*xysppc*] ORAB [*oprx16*,*xysppc*]	OR accumulator B (B) | (M)⇒B or (B) | imm⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CA ii DA dd FA hh ll EA xb EA xb ff EA xb ee ff EA xb EA xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ORCC #*opr8i*	OR CCR; (CCR) | imm⇒CCR	IMM	14 ii	P	⇑ – ⇑ ⇑ ⇑ ⇑ ⇑ ⇑
PSHA	Push A; (SP)–1⇒SP; (A)⇒M_{SP}	INH	36	Os	– – – – – – – –
PSHB	Push B; (SP)–1⇒SP; (B)⇒M_{SP}	INH	37	Os	– – – – – – – –
PSHC	Push CCR; (SP)–1⇒SP; (CCR)⇒M_{SP}	INH	39	Os	– – – – – – – –
PSHD	Push D (SP)–2⇒SP; (A:B)⇒$M_{SP}{:}M_{SP+1}$	INH	3B	OS	– – – – – – – –
PSHX	Push X (SP)–2⇒SP; $(X_H{:}X_L)\Rightarrow M_{SP}{:}M_{SP+1}$	INH	34	OS	– – – – – – – –
PSHY	Push Y (SP)–2⇒SP; $(Y_H{:}Y_L)\Rightarrow M_{SP}{:}M_{SP+1}$	INH	35	OS	– – – – – – – –
PULA	Pull A $(M_{SP})\Rightarrow$A; (SP)+1⇒SP	INH	32	uf0	– – – – – – – –
PULB	Pull B $(M_{SP})\Rightarrow$B; (SP)+1⇒SP	INH	33	uf0	– – – – – – – –
PULC	Pull CCR $(M_{SP})\Rightarrow$CCR; (SP)+1⇒SP	INH	38	uf0	Δ ⇓ Δ Δ Δ Δ Δ
PULD	Pull D $(M_{SP}{:}M_{SP+1})\Rightarrow$A:B; (SP)+2⇒SP	INH	3A	Uf0	– – – – – – – –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
PULX	Pull X $(M_{SP}:M_{SP+1})\Rightarrow X_H:X_L$; (SP)+2⇒SP	INH	30	Ufo	– – – – – – – –
PULY	Pull Y $(M_{SP}:M_{SP+1})\Rightarrow Y_H:Y_L$; (SP)+2⇒SP	INH	31	Ufo	– – – – – – – –
REV	Rule evaluation, unweighted; find smallest rule input; store to rule outputs unless fuzzy output is larger	Special	18 3A	Orf(t^tx)O* ff+Orft^**	– – ? – ? ? Δ ?
*The t^tx loop is executed once for each element in the rule list. The ^ denotes a check for pending interrupt requests. **These are additional cycles caused by an interrupt: ff is the exit sequence and Orft^ is the re-entry sequence.					
REVW	Rule evaluation, weighted; rule weights optional; find smallest rule input; store to rule outputs unless fuzzy output is larger	Special	18 3B	ORf(t^Tx)O* or ORf(r^ffRf)O** ffff+ORft^*** ffff+ORfr^****	– – ? – ? ? Δ !
*With weighting not enabled, the t^Tx loop is executed once for each element in the rule list. The ^ denotes a check for pending interrupt requests. **With weighting enabled, the t^Tx loop is replaced by r^ffRf. ***Additional cycles caused by an interrupt when weighting is not enabled: ffff is the exit sequence and ORft^ is the re-entry sequence. **** Additional cycles caused by an interrupt when weighting is enabled: ffff is the exit sequence and ORfr^ is the re-entry sequence.					
ROL *opr16a* ROL *oprx0_xysppc* ROL *oprx9*,*xysppc* ROL *oprx16*,*xysppc* ROL [D,*xysppc*] ROL [*oprx16*,*xysppc*] ROLA ROLB	Rotate left M  Rotate left A Rotate left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	75 hh ll 65 xb 65 xb ff 65 xb ee ff 65 xb 65 xb ee ff 45 55	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
ROR *opr16a* ROR *oprx0_xysppc* ROR *oprx9*,*xysppc* ROR *oprx16*,*xysppc* ROR [D,*xysppc*] ROR [*oprx16*,*xysppc*] RORA RORB	Rotate right M  Rotate right A Rotate right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	76 hh ll 66 xb 66 xb ff 66 xb ee ff 66 xb 66 xb ee ff 46 56	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
RTC	Return from call; $(M_{SP})\Rightarrow$PPAGE (SP)+1⇒SP; $(M_{SP}:M_{SP+1})\Rightarrow PC_H:PC_L$ (SP)+2⇒SP	INH	0A	uUnfPPP	– – – – – – – –
RTI	Return from interrupt $(M_{SP})\Rightarrow$CCR; (SP)+1⇒SP $(M_{SP}:M_{SP+1})\Rightarrow$B:A;(SP)+2⇒SP $(M_{SP}:M_{SP+1})\Rightarrow X_H:X_L$;(SP)+4⇒SP $(M_{SP}:M_{SP+1})\Rightarrow PC_H:PC_L$;(SP)+2⇒SP $(M_{SP}:M_{SP+1})\Rightarrow Y_H:Y_L$;(SP)+4⇒SP	INH	0B	uUUUUPPP or uUUUUfVfPPP*	Δ ⇓ Δ Δ Δ Δ Δ Δ
*RTI takes 11 cycles if an interrupt is pending.					
RTS	Return from subroutine $(M_{SP}:M_{SP+1})\Rightarrow PC_H:PC_L$; (SP)+2⇒SP	INH	3D	UfPPP	– – – – – – – –
SBA	Subtract B from A; (A)–(B)⇒A	INH	18 16	OO	– – – – Δ Δ Δ Δ
SBCA #*opr8i* SBCA *opr8a* SBCA *opr16a* SBCA *oprx0_xysppc* SBCA *oprx9*,*xysppc* SBCA *oprx16*,*xysppc* SBCA [D,*xysppc*] SBCA [*oprx16*,*xysppc*]	Subtract with carry from A (A)–(M)–C⇒A or (A)–imm–C⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	82 ii 92 dd B2 hh ll A2 xb A2 xb ff A2 xb ee ff A2 xb A2 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
SBCB #*opr8i* SBCB *opr8a* SBCB *opr16a* SBCB *oprx0_xysppc* SBCB *oprx9,xysppc* SBCB *oprx16,xysppc* SBCB [D,*xysppc*] SBCB [*oprx16,xysppc*]	Subtract with carry from B (B)–(M)–C⇒B or (B)–imm–C⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C2 ii D2 dd F2 hh ll E2 xb E2 xb ff E2 xb ee ff E2 xb E2 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ
SECSame as ORCC #$01	Set C bit	IMM	14 01	P	– – – – – – – 1
SEISame as ORCC #$10	Set I bit	IMM	14 10	P	– – – 1 – – – –
SEVSame as ORCC #$02	Set V bit	IMM	14 02	P	– – – – – – 1 –
SEX *abc,dxysp*Same as TFR r1, r2	Sign extend; 8-bit r1 to 16-bit r2 $00:(r1)⇒r2 if bit 7 of r1 is 0 $FF:(r1)⇒r2 if bit 7 of r1 is 1	INH	B7 eb	P	– – – – – – – –
STAA *opr8a* STAA *opr16a* STAA *oprx0_xysppc* STAA *oprx9,xysppc* STAA *oprx16,xysppc* STAA [D,*xysppc*] STAA [*oprx16,xysppc*]	Store accumulator A (A)⇒M	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5A dd 7A hh ll 6A xb 6A xb ff 6A xb ee ff 6A xb 6A xb ee ff	Pw PwO Pw PwO PwP PIfw PIPw	– – – – Δ Δ 0 –
STAB *opr8a* STAB *opr16a* STAB *oprx0_xysppc* STAB *oprx9,xysppc* STAB *oprx16,xysppc* STAB [D,*xysppc*] STAB [*oprx16,xysppc*]	Store accumulator B (B)⇒M	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5B dd 7B hh ll 6B xb 6B xb ff 6B xb ee ff 6B xb 6B xb ee ff	Pw PwO Pw PwO PwP PIfw PIPw	– – – – Δ Δ 0 –
STD *opr8a* STD *opr16a* STD *oprx0_xysppc* STD *oprx9,xysppc* STD *oprx16,xysppc* STD [D,*xysppc*] STD [*oprx16,xysppc*]	Store D (A:B)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5C dd 7C hh ll 6C xb 6C xb ff 6C xb ee ff 6C xb 6C xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –
STOP	Stop processing; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP} Stop all clocks	INH	18 3E	00SSSSsf (enter stop mode) fVfPPP (exit stop mode) ff (continue stop mode) 00 (if stop mode disabled by S=1)	– – – – – – – –
STS *opr8a* STS *opr16a* STS *oprx0_xysppc* STS *oprx9,xysppc* STS *oprx16,xysppc* STS [D,*xysppc*] STS [*oprx16,xysppc*]	Store SP (SP_H:SP_L)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5F dd 7F hh ll 6F xb 6F xb ff 6F xb ee ff 6F xb 6F xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –
STX *opr8a* STX *opr16a* STX *oprx0_xysppc* STX *oprx9,xysppc* STX *oprx16,xysppc* STX [D,*xysppc*] STX [*oprx16,xysppc*]	Store X (X_H:X_L)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5E dd 7E hh ll 6E xb 6E xb ff 6E xb ee ff 6E xb 6E xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
STY *opr8a* STY *opr16a* STY *oprx0_xysppc* STY *oprx9,xysppc* STY *oprx16,xysppc* STY [D,*xysppc*] STY [*oprx16,xysppc*]	Store Y (Y_H:Y_L)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5D dd 7D hh ll 6D xb 6D xb ff 6D xb ee ff 6D xb 6D xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –

SUBA #*opr8i*
SUBA *opr8a*
SUBA *opr16a*
SUBA *oprx0_xysppc*
SUBA *oprx9,xysppc*
SUBA *oprx16,xysppc*
SUBA [D,*xysppc*]
SUBA [*oprx16,xysppc*]
	Subtract from A
	(A)–(M)⇒A
	or (A)–imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			80 ii
			90 dd
			B0 hh ll
			A0 xb
			A0 xb ff
			A0 xb ee ff
			A0 xb
			A0 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

SUBB #*opr8i*
SUBB *opr8a*
SUBB *opr16a*
SUBB *oprx0_xysppc*
SUBB *oprx9,xysppc*
SUBB *oprx16,xysppc*
SUBB [D,*xysppc*]
SUBB [*oprx16,xysppc*]
	Subtract from B
	(B)–(M)⇒B
	or (B)–imm⇒B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C0 ii
			D0 dd
			F0 hh ll
			E0 xb
			E0 xb ff
			E0 xb ee ff
			E0 xb
			E0 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

SUBD #*opr16i*
SUBD *opr8a*
SUBD *opr16a*
SUBD *oprx0_xysppc*
SUBD *oprx9,xysppc*
SUBD *oprx16,xysppc*
SUBD [D,*xysppc*]
SUBD [*oprx16,xysppc*]
	Subtract from D
	(A:B)–(M:M+1)⇒A:B
	or (A:B)–imm⇒A:B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			83 jj kk
			93 dd
			B3 hh ll
			A3 xb
			A3 xb ff
			A3 xb ee ff
			A3 xb
			A3 xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ Δ Δ

SWI	Software interrupt; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP};1⇒I (SWI vector)⇒PC	INH	3F	VSPSSPSsP*	– – – 1 – – – –
*The CPU also uses VSPSSPSsP for hardware interrupts and unimplemented opcode traps.					
TAB	Transfer A to B; (A)⇒B	INH	18 0E	OO	– – – – Δ Δ 0 –
TAP	Transfer A to CCR; (A)⇒CCR Assembled as TFR A, CCR	INH	B7 02	P	Δ ⇓ Δ Δ Δ Δ Δ Δ
TBA	Transfer B to A; (B)⇒A	INH	18 0F	OO	– – – – Δ Δ 0 –
TBEQ *abdxysp,rel9*	Test and branch if equal to 0 If (counter)=0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TBL *oprx0_xysppc*	Table lookup and interpolate, 8-bit (M)+[(B)×((M+1)–(M))]⇒A	IDX	18 3D xb	ORfffP	– – – – Δ Δ – Δ
TBNE *abdxysp,rel9*	Test and branch if not equal to 0 If (counter)≠0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TFR *abcdxysp,abcdxysp*	Transfer from register to register (r1)⇒r2r1 and r2 same size $00:(r1)⇒r2r1=8-bit; r2=16-bit ($r1_L$)⇒r2r1=16-bit; r2=8-bit	INH	B7 eb	P	– – – – – – – – or Δ ⇓ Δ Δ Δ Δ Δ Δ
TPASame as TFR CCR ,A	Transfer CCR to A; (CCR)⇒A	INH	B7 20	P	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
TRAP *trapnum*	Trap unimplemented opcode; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP} 1⇒I; (trap vector)⇒PC	INH	18 tn tn = $30–$39 or tn = $40–$FF	OVSPSSPSsP	– – – 1 – – – –
TST *opr16a* TST *oprx0_xysppc* TST *oprx9,xysppc* TST *oprx16,xysppc* TST [D,*xysppc*] TST [*oprx16,xysppc*] TSTA TSTB	Test M; (M)–0 Test A; (A)–0 Test B; (B)–0	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	F7 hh ll E7 xb E7 xb ff E7 xb ee ff E7 xb E7 xb ee ff 97 D7	rPO rPf rPO frPP fIfrPf fIPrPf O O	– – – – Δ Δ 0 0
TSXSame as TFR SP,X	Transfer SP to X; (SP)⇒X	INH	B7 75	P	– – – – – – – –
TSYSame as TFR SP,Y	Transfer SP to Y; (SP)⇒Y	INH	B7 76	P	– – – – – – – –
TXSSame as TFR X,SP	Transfer X to SP; (X)⇒SP	INH	B7 57	P	– – – – – – – –
TYSSame as TFR Y,SP	Transfer Y to SP; (Y)⇒SP	INH	B7 67	P	– – – – – – – –
WAI	Wait for interrupt; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP}	INH	3E	OSSSSsf (before interrupt) fVfPPP (after interrupt)	– – – – – – – – or – – – 1 – – – – or – 1 – 1 – – – –
WAV	$\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ $\sum_{i=1}^{B} F_i \Rightarrow X$ Calculate weighted average; sum of products (SOP) and sum of weights (SOW)*	Special	18 3C	Of(frr^ffff)O** SSS+UUUrr^***	– – ? – ? Δ ? ?
*Initialize B, X, and Y: B=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-bit values. **The frr^ffff sequence is the loop for one iteration of SOP and SOW accumulation. The ^ denotes a check for pending interrupt requests. ***Additional cycles caused by an interrupt: SSS is the exit sequence and UUUrr^ is the re-entry sequence. Intermediate values use six stack bytes.					
wavr*	Resume executing interrupted WAV	Special	3C	UUUrr^ffff(frr^ ffff)O** SSS+UUUrr^***	– – ? – ? Δ ? ?
*wavr is a pseudoinstruction that recovers intermediate results from the stack rather than initializing them to 0. **The frr^ffff sequence is the loop for one iteration of SOP and SOW recovery. The ^ denotes a check for pending interrupt requests. ***These are additional cycles caused by an interrupt: SSS is the exit sequence and UUUrr^ is the re-entry sequence.					
XGDXSame as EXG D, X	Exchange D with X; (D)⇔(X)	INH	B7 C5	P	– – – – – – – –
XGDYSame as EXG D, Y	Exchange D with Y; (D)⇔(Y)	INH	B7 C6	P	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

1.8.1 Register and Memory Notation

Table 1-3 Register and Memory Notation

A or *a*	Accumulator A
An	Bit n of accumulator A
B or *b*	Accumulator B
Bn	Bit n of accumulator B
D or *d*	Accumulator D
Dn	Bit n of accumulator D
X or *x*	Index register X
X_H	High byte of index register X
X_L	Low byte of index register X
Xn	Bit n of index register X
Y or *y*	Index register Y
Y_H	High byte of index register Y
Y_L	Low byte of index register Y
Yn	Bit n of index register Y
SP or *sp*	Stack pointer
SPn	Bit n of stack pointer
PC or *pc*	Program counter
PC_H	High byte of program counter
PC_L	Low byte of program counter
CCR or *c*	Condition code register
M	Address of 8-bit memory location
Mn	Bit n of byte at memory location M
Rn	Bit n of the result of an arithmetic or logical operation
In	Bit n of the intermediate result of an arithmetic or logical operation
RTN_H	High byte of return address
RTN_L	Low byte of return address
( )	Contents of

1.8.2 Source Form Notation

The **Source Form** column of the summary in **Table 1-2** gives essential information about assembler source forms. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Everything in the **Source Form** column, *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 (A, B, D), are literal characters.

The groups of italic characters shown in **Table 1-4** 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 *xysppc* and *oprx0_xysppc* are both valid, but the two groups *oprx0 xysppc* are not valid because there is a space between them.

Table 1-4 Source Form Notation

abc	Register designator for A, B, or CCR
abcdxysp	Register designator for A, B, CCR, D, X, Y, or SP
abd	Register designator for A, B, or D
abdxysp	Register designator for A, B, D, X, Y, or SP
dxysp	Register designator for D, X, Y, or SP
msk8	8-bit mask value Some assemblers require the # symbol before the mask value.
opr8i	8-bit immediate value
opr16i	16-bit immediate value
opr8a	8-bit address value used with direct address mode
opr16a	16-bit address value
oprx0_xysp	Indexed addressing postbyte code: *oprx3,–xysp* — Predecrement X , Y, or SP by 1–8 *oprx3,+xysp* — Preincrement X , Y, or SP by 1–8 *oprx3,xysp–* — Postdecrement X, Y, or SP by 1–8 *oprx3,xysp+* — Postincrement X, Y, or SP by 1–8 *oprx5,xysppc* — 5-bit constant offset from X, Y, SP, or PC *abd,xysppc* — Accumulator A, B, or D offset from X, Y, SP, or PC
oprx3	Any positive integer from 1 to 8 for pre/post increment/decrement
oprx5	Any integer from –16 to +15
oprx9	Any integer from –256 to +255
oprx16	Any integer from –32,768 to +65,535
page	8-bit value for PPAGE register Some assemblers require the # 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 the 64-Kbyte memory space
trapnum	Any 8-bit integer from $30 to $39 or from $40 to $FF
xysp	Register designator for X or Y or SP
xysppc	Register designator for X or Y or SP or PC

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

1.8.3 Operation Notation

Table 1-5 Operation Notation

+	Add
–	Subtract
•	AND
|	OR
⊕	Exclusive OR
×	Multiply
÷	Divide
:	Concatenate
⇒	Transfer
⇔	Exchange

1.8.4 Address Mode Notation

Table 1-6 Address Mode Notation

INH	Inherent; no operands in instruction stream
IMM	Immediate; operand immediate value in instruction stream
DIR	Direct; operand is lower byte of address from $0000 to $00FF
EXT	Operand is a 16-bit address
REL	Two’s complement relative offset; for branch instructions
IDX	Indexed (no extension bytes); includes: 5-bit constant offset from X, Y, SP or PC Pre/post increment/decrement by 1–8 Accumulator A, B, or D offset
IDX1	9-bit signed offset from X, Y, SP, or PC; 1 extension byte
IDX2	16-bit signed offset from X, Y, SP, or PC; 2 extension bytes
[IDX2]	Indexed-indirect; 16-bit offset from X, Y, SP, or PC
[D, IDX]	Indexed-indirect; accumulator D offset from X, Y, SP, or PC

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

1.8.5 Machine Code Notation

In the **Machine Code (Hex)** column of the summary in **Table 1-2**, digits 0–9 and upper case letters A–F represent hexadecimal values. Pairs of lower-case letters represent 8-bit values as shown in **Table 1-7**.

Table 1-7 Machine Code Notation

dd	8-bit direct address from $0000 to $00FF; high byte is $00
ee	High byte of a 16-bit constant offset for indexed addressing
eb	Exchange/transfer postbyte
ff	Low eight bits of a 9-bit signed constant offset in indexed addressing, or low byte of a 16-bit constant offset in indexed addressing
hh	High byte of a 16-bit extended address
ii	8-bit immediate data value
jj	High byte of a 16-bit immediate data value
kk	Low byte of a 16-bit immediate data value
lb	Loop primitive (DBNE) postbyte
ll	Low byte of a 16-bit extended address
mm	8-bit immediate mask value for bit manipulation instructions; bits that are set indicate bits to be affected
pg	Program page or bank number used in CALL instruction
qq	High byte of a 16-bit relative offset for long branches
tn	Trap number from $30 to $39 or from $40 to $FF
rr	Signed relative offset $80 (–128) to $7F (+127) relative to the byte following the relative offset byte, or low byte of a 16-bit relative offset for long branches
xb	Indexed addressing postbyte

1.8.6 Access Detail Notation

A single-letter code in the **Access Detail** column of **Table 1-2** represents a single CPU access cycle. An upper-case letter indicates a 16-bit access.

Table 1-8 Access Detail Notation

f	Free cycle. During an f cycle, the CPU does not use the bus. An f cycle is always one cycle of the system bus clock. An f cycle can be used by a queue controller or the background debug system to perform a single-cycle access without disturbing the CPU.
g	Read PPAGE register. A g cycle is used only in CALL instructions and is not visible on the external bus. Since PPAGE is an internal 8-bit register, a g cycle is never stretched.
I	Read indirect pointer. Indexed-indirect instructions use the 16-bit indirect pointer from memory to address the instruction operand. An I cycle is a 16-bit read that can be aligned or misaligned. An I cycle is 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. An I cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
i	Read indirect PPAGE value. An i cycle is used only in indexed-indirect CALL instructions. The 8-bit PPAGE value for the CALL destination is fetched from an indirect memory location. An i cycle is stretched only when controlled by a chip-select circuit that is programmed for slow memory.
n	Write PPAGE register. An n cycle is used only in CALL and RTC instructions to write the destination value of the PPAGE register and is not visible on the external bus. Since the PPAGE register is an internal 8-bit register, an n cycle is never stretched.
O	Optional cycle. An O cycle adjusts instruction alignment in the instruction queue. An O cycle can be a free cycle (f) or a program word access cycle (P). When the first byte of an instruction with an odd number of bytes is misaligned, the O cycle becomes a P cycle to maintain queue order. If the first byte is aligned, the O cycle is an f cycle. The $18 prebyte for a page-two opcode is treated as a special one-byte instruction. If the prebyte is misaligned, the O cycle at the beginning of the instruction becomes a P cycle to maintain queue order. If the prebyte is aligned, the O cycle is an f cycle. If the instruction has an odd number of bytes, it has a second O cycle at the end. If the first O cycle is a P cycle (prebyte misaligned), the second O cycle is an f cycle. If the first O cycle is an f cycle (prebyte aligned), the second O cycle is a P cycle. An O cycle that becomes a P cycle 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. An O cycle that becomes an f cycle is never stretched.
P	Program word access. Program information is fetched as aligned 16-bit words. A P cycle is 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. An r cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
R	16-bit data read. An R cycle is 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. An R cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
s	Stack 8-bit data. An s cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 1-8 Access Detail Notation (Continued)

S	Stack 16-bit data. An S cycle is 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. An S cycle is also stretched if it corresponds to a misaligned access 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. A w cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
W	16-bit data write. A W cycle is 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. A W cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
u	Unstack 8-bit data. A W cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
U	Unstack 16-bit data. A U cycle is 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. A U cycle is also stretched if it corresponds to a misaligned access 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	16-bit vector fetch. Vectors are always aligned 16-bit words. A V cycle is 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. A t cycle is either a data read cycle or a free cycle, depending on the data and flow of the REVW instruction. A t cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
T	16-bit conditional read. A T cycle is either a data read cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. A T cycle is 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. A T cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
x	8-bit conditional write. An x cycle is either a data write cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. An x cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
Special Notation for Branch Taken/Not Taken	
PPP/P	A short branch requires 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	A long branch requires four cycles if taken, three cycles if not taken. An O cycle is required because all long branches are page two opcodes and thus include the $18 prebyte. The prebyte is treated as a one-byte instruction. If the prebyte is misaligned, the O cycle is a P cycle; if the prebyte is aligned, the O cycle is an f cycle. As a result, both the taken and not-taken cases use one O cycle for the prebyte. In the not-taken case, the queue must advance so that execution can continue with the next instruction, and another O 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.

1.8.7 Condition Code State Notation

Table 1-9 Condition Code State Notation

–	Not changed by operation
0	Cleared by operation
1	Set by operation
Δ	Set or cleared by operation
⇓	May be cleared or remain set, but not set by operation
⇑	May be set or remain cleared, but not cleared by operation
?	May be changed by operation but final state not defined
!	Used for a special purpose

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 2 Nomenclature

This section describes the conventions and notation used to describe the Core operation.

2.1 References

This document uses the *Sematech Official Dictionary* and the JEDEC/EIA *Reference Guide to Letter Symbols for Semiconductor Devices* as references for terminology and symbology.

2.2 Units and Measures

SIU units and abbreviations are used throughout this guide.

2.3 Symbology

The symbols and operators used throughout this guide are shown in **Table 2-1**.

Table 2-1 Symbols and Operators

Symbol	Function
+	Addition
-	Subtraction (two's complement) or negation
*	Multiplication
/	Division
>	Greater
<	Less
=	Equal
≥	Equal or greater
≤	Equal or less
	Not equal
•	AND
+	Inclusive OR (OR)
⊕	Exclusive OR (EOR)
$\overline{\text{NOT}}$	Complementation
:	Concatenation
⇒	Transferred
⇔	Exchanged
	Tolerance
0b0011	Binary value
0x0F	Hexadecimal value

2.4 Terminology

Logic level one is a voltage that corresponds to Boolean true (1) state.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Logic level zero is a voltage that corresponds to Boolean false (0) state.

To **set** a bit or bits means to establish logic level one on them.

To **clear** a bit or bits means to establish logic level zero on them.

A **signal** is an electronic construct whose state or changes in state convey information.

A **pin** is an external physical connection. The same pin can be used to connect a number of signals.

Asserted means that a discrete signal is in active logic state.

- **Active low** signals change from logic level one to logic level zero.
- **Active high** signals change from logic level zero to logic level one.

Negated means that an asserted discrete signal changes logic state.

- **Active low** signals change from logic level zero to logic level one.
- **Active high** signals change from logic level one to logic level zero.

LSB means least significant bit or bits. **MSB** means most significant bit or bits. References to low and high bytes or words are spelled out.

Memory and registers use **big-endian** ordering. The most significant byte (byte 0) of word 0 is located at address 0. Bits within a word are numbered downward from the MSB, bit 15.

Signal, bit field, and control bit mnemonics follow a general numbering scheme:

- A **range of mnemonics** is referred to by mnemonic and numbers that define the range, from highest to lowest. For example, *p_addr[4:0]* are lines four to zero of an address bus.
- A **single mnemonic** stands alone or includes a single numeric designator when appropriate. For example, *m_rst* is a unique mnemonic, while *p_addr15* represents line 15 of an address bus.

Section 3 Core Registers

This section provides detailed descriptions of the Core programming model, registers and accumulators. In addition, a general description of the complete Core register map which includes all Core sub-blocks is included.

3.1 Programming Model

The Core CPU12 programming model, shown in **Figure 3-1**, is the same as that of the 68HC12 and 68HC11. The register set and data types used in the model are covered in the subsections that follow.

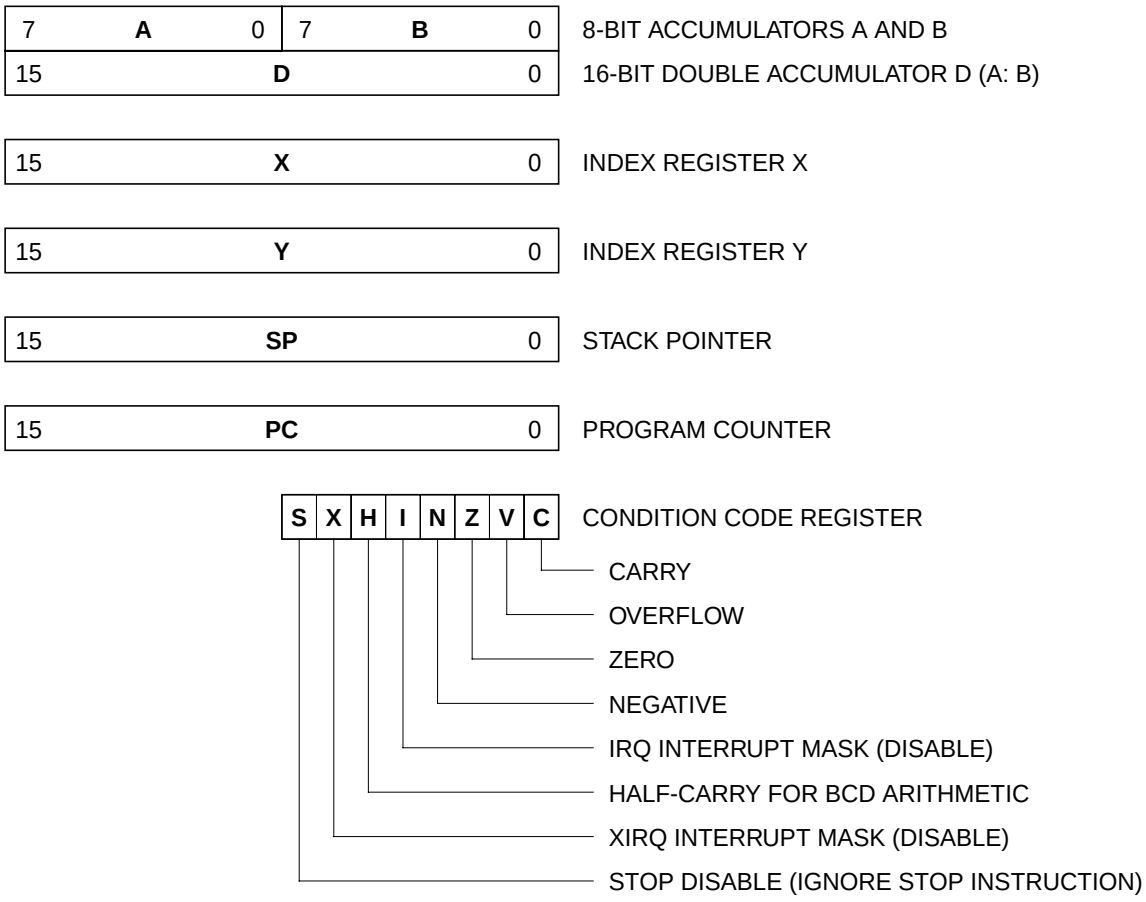


Figure 3-1 Programming Model

3.1.1 Accumulators

General-purpose 8-bit accumulators A and B hold operands and results of operations. Some instructions use the combined 8-bit accumulators, A:B, as a 16-bit double accumulator, D, with the most significant byte in A.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

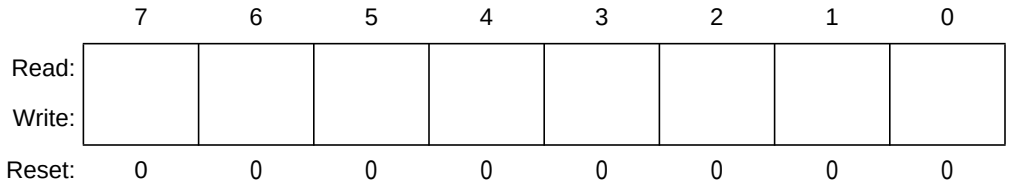


Figure 3-2 Accumulator A

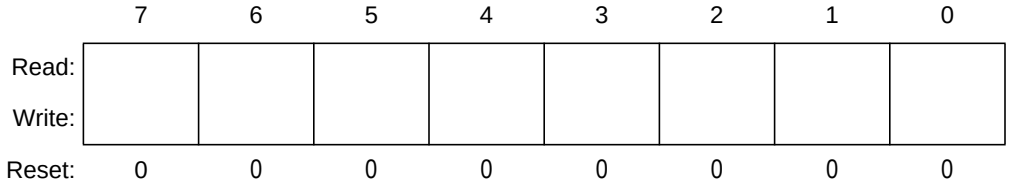


Figure 3-3 Accumulator B

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 verify that 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.

3.1.2 Index Registers (X and Y)

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

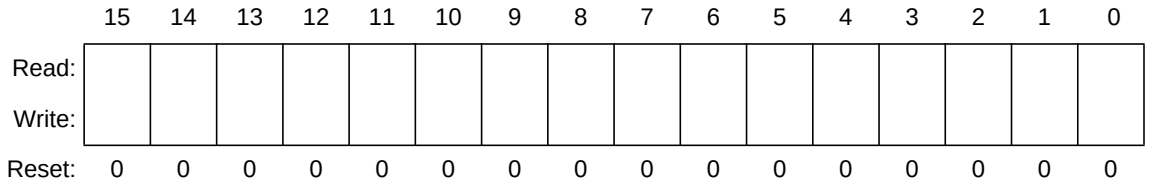


Figure 3-4 Index Register X

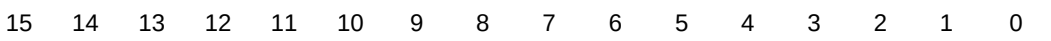


Figure 3-5 Index Register Y

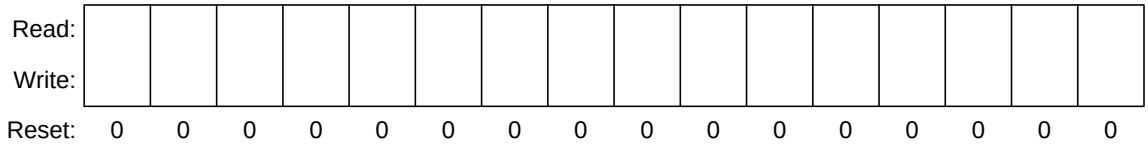


Figure 3-5 Index Register Y

3.1.3 Stack Pointer (SP)

The stack stores system context during subroutine calls and interrupts, and can also be used for temporary data storage. It can be located anywhere in the standard 64K byte address space and can grow to any size up to the total amount of memory available in the system.

SP holds the 16-bit address of the last stack location used. Normally, SP is initialized by one of the first instructions in an application program. The stack grows downward from the address pointed to by 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) 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.

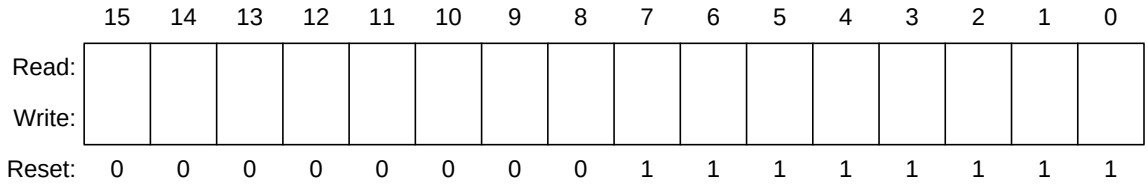


Figure 3-6 Stack Pointer (SP)

When an interrupt occurs, the CPU:

- Completes execution of the current instruction
- Calculates the address of the next instruction and pushes it onto the stack
- Pushes the contents of all the CPU registers onto the stack
- Loads the program counter with the address pointed to by the interrupt vector, and begins execution at that address

The stacked CPU registers are referred to as an interrupt stack frame. The Core stack frame is the same as that of the CPU.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

3.1.4 Program Counter (PC)

PC is a 16-bit register that holds the address of the next instruction to be executed. The address in PC is automatically incremented each time an instruction is executed.

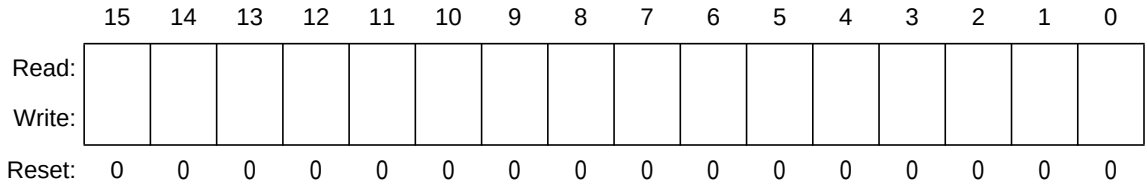


Figure 3-7 Program Counter (PC)

3.1.5 Condition Code Register (CCR)

CCR has five status bits, two interrupt mask bits, and a STOP instruction mask bit. It is named for the five conditions indicated by the status bits.

The status bits reflect the results of CPU operations. The five status bits are half-carry (H), negative (N), zero (Z), overflow (V), and carry/borrow (C). The half-carry bit is used only for BCD arithmetic operations. The N, Z, V, and C status bits allow for branching based on the results of a CPU operation.

Most instructions automatically update condition codes, so it is rarely necessary to execute extra instructions to load and test a variable. The condition codes affected by each instruction are shown in **Appendix B** of this guide.

The following paragraphs describe common 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 guide and in **Appendix B**.

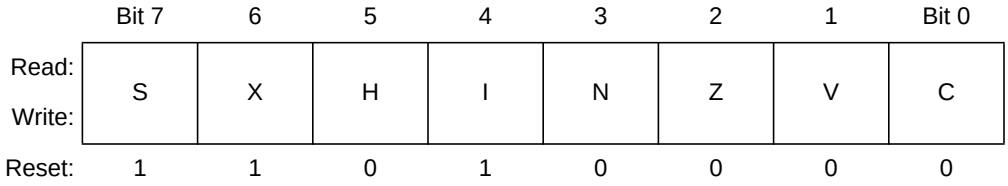


Figure 3-8 Condition Code Register (CCR)

S — STOP Mask Bit

Clearing the S bit enables the STOP instruction. Execution of a STOP instruction causes the on-chip oscillator to stop. This may be undesirable in some applications. When the S bit is set, the CPU treats the STOP instruction as a no-operation (NOP) instruction and continues on to the next instruction. Reset sets the S bit.

1 = STOP instruction disabled
0 = STOP instruction enabled

X — $\overline{\text{XIRQ}}$ Mask Bit

Clearing the X bit enables interrupt requests on the $\overline{\text{XIRQ}}$ pin. The $\overline{\text{XIRQ}}$ input is an updated version of the nonmaskable interrupt ($\overline{\text{NMI}}$) input found on earlier generations of Motorola microcontroller units (MCUs). Nonmaskable interrupts are typically used to deal with major system failures such as loss of power. However, enabling nonmaskable interrupts before a system is fully powered and initialized can lead to spurious interrupts. The X bit provides a mechanism for masking nonmaskable interrupts until the system is stable.

Reset sets the X bit. As long as the X bit remains set, interrupt service requests made via the $\overline{\text{XIRQ}}$ pin are not recognized. Software must clear the X bit to enable interrupt service requests from the $\overline{\text{XIRQ}}$ pin. Once software clears the X bit, enabling $\overline{\text{XIRQ}}$ interrupt requests, only a reset can set it again. The X bit does not affect I bit maskable interrupt requests.

When the X bit is clear and an $\overline{\text{XIRQ}}$ interrupt request occurs, the CPU stacks the cleared X bit. It then automatically sets the X and I bits in the CCR to disable $\overline{\text{XIRQ}}$ and maskable interrupt requests during the $\overline{\text{XIRQ}}$ interrupt service routine.

An RTI instruction at the end of the interrupt service routine restores the cleared X bit to the CCR, re-enabling $\overline{\text{XIRQ}}$ interrupt requests.

1 = $\overline{\text{XIRQ}}$ interrupt requests disabled
0 = $\overline{\text{XIRQ}}$ interrupt requests enabled

H — Half-Carry Bit

The H bit indicates a carry from bit 3 of the result during an addition operation. The DAA instruction uses the value of the H bit to adjust the result in accumulator A to BCD format. ABA, ADD, and ADC are the only instructions that update the H bit.

1 = Carry from bit 3 after ABA, ADD, or ADC instruction
0 = No carry from bit 3 after ABA, ADD, or ADC instruction

I — Interrupt Mask Bit

Clearing the I bit enables maskable interrupt sources. Reset sets the I bit. To enable maskable interrupt requests, software must clear the I bit. Maskable interrupt requests that occur while the I bit is set remain pending until the I bit is cleared.

When the I bit is clear and a maskable interrupt request occurs, the CPU stacks the cleared I bit. It then automatically sets the I bit in the CCR to prevent other maskable interrupt requests during the interrupt service routine.

An RTI instruction at the end of the interrupt service routine restores the cleared I bit to the CCR, reenabling maskable interrupt requests. The I bit can be cleared within the service routine, but implementing a nested interrupt scheme requires great care, and seldom improves system performance.

1 = Maskable interrupt requests disabled
0 = Maskable interrupt requests enabled

N — Negative Bit

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

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

1 = MSB of result set
0 = MSB of result clear

Z — Zero Bit

The Z bit is set when all the bits of the result are zeros. Compare instructions perform an internal implied subtraction, and the condition codes, including Z, reflect the results of that subtraction. The INX, DEX, INY, and DEY instructions affect the Z bit and no other condition bits. These operations can only determine = and ≠.

1 = Result all zeros
0 = Result not all zeros

V — Overflow Bit

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

1 = Overflow
0 = No overflow

C — Carry 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.

1 = Carry or borrow
0 = No carry or borrow

3.2 Core Register Map

The Core registers are those that are part of the sub-blocks that support the CPU to makeup the entire Core block. In addition to the registers contributed by the Core sub-blocks, sections of the Core space are reserved for registers contributed by the system peripherals and memory sub-blocks. These registers are configured at integration of the Core into the SoC design. The Core register map summary is shown in **Figure 3-9** below.

The Core registers, with the exception of those associated with the BDM sub-block (addresses $FF00 through $FF07), can be mapped to any 2K byte block within the first 32K byte space of the standard 64K byte address area by configuring the INITRG register.

For detailed descriptions of the Core register and bit functionality please refer to Core sub-block description sections of this guide. To assist in locating this more detailed information, **Table 3-1** below lists the Core registers, the sub-block they are associated with and a brief description of function.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$0000	PORTA	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0001	PORTB	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0002	DDRA	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0003	DDRB	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0004	Reserved	read write	0	0	0	0	0	0	0	0
$0005	Reserved	read write	0	0	0	0	0	0	0	0
$0006	Reserved	read write	0	0	0	0	0	0	0	0
$0007	Reserved	read write	0	0	0	0	0	0	0	0
$0008	PORTE	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0009	DDRE	read write	Bit 7	6	5	4	3	2	0	0
$000A	PEAR	read write	NOACCE	0	PIPOE	NECLK	LSTRE	RDWE	0	0
$000B	MODE	read write	MODC	MODB	MODA	0	IVIS	0	EMK	EME
$000C	PUCR	read write	PUPKE	0	0	PUPEE	0	0	PUPBE	PUPAE
$000D	RDRIV	read write	RDPK	0	0	RDPE	0	0	RDPB	RDPA
$000E	EBICTL	read write	0	0	0	0	0	0	0	ESTR
$000F	Reserved	read write	0	0	0	0	0	0	0	0
$0010	INITRM	read write	RAM15	RAM14	RAM13	RAM12	RAM11	0	0	RAMHAL
$0011	INITRG	read write	0	REG14	REG13	REG12	REG11	0	0	0
$0012	INITEE	read write	EE15	EE14	EE13	EE12	EE11	0	0	EEON
$0013	MISC	read write	0	0	0	0	EXSTR1	EXSTR0	ROMHM	ROMON
$0014	Reserved	read write	0	0	0	0	0	0	0	0
$0015	ITCR	read write	0	0	0	WRTINT	ADR3	ADR2	ADR1	ADR0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

$0016	ITEST	read write	INTE	INTC	INTA	INT8	INT6	INT4	INT2	INT0
$0017	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0018 to $001B	Reserved		Reserved for Peripheral Block Registers							
$001C	MEMSIZ0	read	reg_sw0	0	eeo_sw1	eep_sw0	0	ram_sw2	ram_sw1	ram_sw0
		write								
$001D	MEMSIZ1	read	rom_sw1	rom_sw0	0	0	0	0	pag_sw1	pag_sw0
		write								
$001E	IRQCR	read	IRQE	IRQEN	0	0	0	0	0	0
		write								
$001F	HPRIO	read	PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0
		write								
$0020 to $0027	Reserved		Reserved for Peripheral Block Registers							
$0028	BKPCT0	read	BKEN	BKFULL	BKBDM	BKTAG	0	0	0	0
		write								
$0029	BKPCT1	read write	BK0MBH	BK0MBL	BK1MBH	BK1MBL	BK0RWE	BK0RW	BK1RWE	BK1RW
$002A	BKP0X	read	0	0	BK0V5	BK0V4	BK0V3	BK0V2	BK0V1	BK0V0
		write								
$002B	BKP0H	read write	Bit 15	14	13	12	11	10	9	Bit 8
$002C	BKP0L	read write	Bit 7	6	5	4	3	2	1	Bit 0
$002D	BKP1X	read	0	0	BK1V5	BK1V4	BK1V3	BK1V2	BK1V1	BK1V0
		write								
$002E	BKP1H	read write	Bit 15	14	13	12	11	10	9	Bit 8
$002F	BKP1L	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0030	PPAGE	read	0	0	PIX5	PIX4	PIX3	PIX2	PIX1	PIX0
		write								
$0031	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0032	PORTK	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0033	DDRK	read write	Bit 7	6	5	4	3	2	1	Bit 0
$0034 to $00FF	Reserved		Reserved for Peripheral Block Registers							
$0100 to $010F	Reserved		Reserved for Flash EEPROM or ROM Registers							

$0110 to $011B	Reserved		Reserved for EEPROM Registers							
$011C to $011F	Reserved		Reserved for RAM Registers							
$0120 to $07FF	Reserved		Reserved for Peripheral Block Registers							
$FF00	Reserved	read	X	X	X	X	X	X	0	0
		write								
$FF01	BDMSTS	read	ENBDM	BDMACT	ENTAG	SDV	TRACE	CLKSW	UNSEC	CORE
		write								
$FF02	Reserved	read	X	X	X	X	X	X	X	X
		write								
$FF03	Reserved	read	X	X	X	X	X	X	X	X
		write								
$FF04	Reserved	read	X	X	X	X	X	X	X	X
		write								
$FF05	Reserved	read	X	X	X	X	X	X	X	X
		write								
$FF06	BDMCCR	read	CCR7	CCR6	CCR5	CCR4	CCR3	CCR2	CCR1	CCR0
		write								
$FF07	BDMINR	read	REG15	REG14	REG13	REG12	REG11	0	0	0
		write								

= Unimplemented X = Indeterminate

Figure 3-9 Core Register Map Summary

Table 3-1 Core Register Map Reference

Address	Name	Sub-block	Description
$0000	PORTA	MEBI	Port A 8-bit Data Register
$0001	PORTB	MEBI	Port B 8-bit Data Register
$0002	DDRA	MEBI	Port A 8-bit Data Direction Register
$0003	DDRB	MEBI	Port B 8-bit Data Direction Register
$0008	PORTE	MEBI	Port E 8-bit Data Register
$0009	DDRE	MEBI	Port E 8-bit Data Direction Register
$000A	PEAR	MEBI	Port E Assignment Register - configures functionality of Port E as general purpose I/O and/or alternate functions
$000B	MODE	MEBI	Used to establish mode of operation of the Core and configure other miscellaneous functions
$000C	PUCR	MEBI	Pullup Control Register to configure state of pullups on Ports A, B, E and K

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 3-1 Core Register Map Reference

Address	Name	Sub-block	Description
$000D	RDRIV	MEBI	Reduced Drive Register to configure drive strength of pins associated with Ports A, B, E and K
$000E	EBICTL	MEBI	External Bus Interface Control Register to configure functionality of external E-clock signal
$0010	INITRM	MMC	Initialization of Internal RAM Position Register
$0011	INITRG	MMC	Initialization of Internal Registers Position Register
$0012	INITEE	MMC	Initialization of Internal EEPROM Registers Position Register
$0013	MISC	MMC	Miscellaneous Register to configure various system functions
$0015	ITCR	INT	Interrupt Test Control Register used in special modes of operation for testing interrupt logic
$0016	ITEST	INT	Interrupt Test Register used in special modes of operation testing interrupt logic
$001C	MEMSIZ0	MMC	Memory Size Register 0 to allow capability to read the state of the system memory configuration switches
$001D	MEMSIZ1	MMC	Memory Size Register 1 to allow capability to read the state of the system memory configuration switches
$001E	IRQCR	MEBI	IRQ Control Register to configure IRQ pin functionality
$001F	HPRIO	INT	Highest Priority I Interrupt Register (optional)
$0028	BKPCT0	BKP	Breakpoint Control Register 0 to configure mode of operation of breakpoint functions
$0029	BKPCT1	BKP	Breakpoint Control Register 1 to configure mode of operation of breakpoint functions
$002A	BKP0X	BKP	First Address Memory Expansion Breakpoint Register to assign first address match value for expanded addresses
$002B	BKP0H	BKP	First Address High Byte Breakpoint Register to assign high byte of first address within system memory space to be matched
$002C	BKP0L	BKP	First Address Low Byte Breakpoint Register to assign low byte of first address within system memory space to be matched
$002D	BKP1X	BKP	Second Address Memory Expansion Breakpoint Register to assign second address match value for expanded addresses
$002E	BKP1H	BKP	Second Address High Byte Breakpoint Register to assign high byte of first address within system memory space to be matched
$002F	BKP1L	BKP	Second Address Low Byte Breakpoint Register to assign low byte of first address within system memory space to be matched
$0030	PPAGE	MMC	Program Page Index Register to configure the active memory page viewed through the program page window from $8000-$BFFF
$0032	PORTK	MEBI	Port K 8-bit Data Register
$0033	DDRK	MEBI	Port K 8-bit Data Direction Register
$FF01	BDMSTS	BDM	BDM Status Register
$FF06	BDMCCR	BDM	BDM CCR Holding Register for interaction of BDM with CPU
$FF07	BDMINR	BDM	BDM Internal Register Position Register to configure BDM register mapping

Section 4 Instructions

This section describes the instruction set of the Core. This discussion includes descriptions of instructions grouped by type, the addressing modes used and the opcode map. Please refer to **Appendix B** of this guide for a detailed instruction-by-instruction description of each opcode.

4.1 Instruction Types

All memory and I/O are mapped in a common 64K byte address space, allowing the same set of instructions to access memory, I/O, and control registers. Load, store, transfer, exchange, and move instructions facilitate movement of data to and from memory and peripherals.

There are instructions for signed and unsigned addition, division and multiplication with 8-bit, 16-bit, and some larger operands.

Special arithmetic and logic instructions aid stacking operations, indexing, BCD calculation, and condition code register manipulation. There are also dedicated instructions for multiply and accumulate operations, table interpolation, and specialized mathematical calculations for fuzzy logic operations.

4.2 Addressing Modes

A summary of the addressing modes used by the Core is given in **Table 4-1** below. The operation of each of these modes is discussed in the subsections that follow.

Table 4-1 Addressing Mode Summary

Addressing Mode	Source Form	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-bit 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	Effective address is the value in PC plus an 8-bit or 16-bit relative offset value.
Indexed (5-bit offset)	INST oprx5,xysp	IDX	Effective address is the value in X, Y, SP, or PC plus a 5-bit signed constant offset.
Indexed (predecrement)	INST oprx3,–xys	IDX	Effective address is the value in X, Y, or SP autodecremented by 1 to 8.
Indexed (preincrement)	INST oprx3,+xys	IDX	Effective address is the value in X, Y, or SP autoincremented by 1 to 8.
Indexed (postdecrement)	INST oprx3,xys–	IDX	Effective address is the value in X, Y, or SP. The value is postdecremented by 1 to 8.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Table 4-1 Addressing Mode Summary

Addressing Mode	Source Form	Abbreviation	Description
Indexed (postincrement)	INST oprx3,xys+	IDX	Effective address is the value in X, Y, or SP. The value is postincremented by 1 to 8.
Indexed (accumulator offset)	INST abd,xysp	IDX	Effective address is the value in X, Y, SP, or PC plus the value in A, B, or D.
Indexed (9-bit offset)	INST oprx9,xysp	IDX1	Effective address is the value in X, Y, SP, or PC plus a 9-bit signed constant offset.
Indexed (16-bit offset)	INST oprx16,xysp	IDX2	Effective address is the value in X, Y, SP, or PC plus a 16-bit constant offset.
Indexed-indirect (16-bit offset)	INST [oprx16,xysp]	[IDX2]	The value in X, Y, SP, or PC plus a 16-bit constant offset points to the effective address.
Indexed-indirect (D accumulator offset)	INST [D,xysp]	[D,IDX]	The value in X, Y, SP, or PC plus the value in D points to the effective address.

4.2.1 Effective Address

Every addressing mode except inherent mode generates a 16-bit effective address. The effective address is the address of the memory location that the instruction acts on. Effective address computations do not require extra execution cycles.

4.2.2 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.

```
NOP                ;this instruction has no operands

INX                ;operand is a CPU register
```

4.2.3 Immediate Addressing Mode

Operands for immediate mode instructions are included in the instruction 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 very common programming error is to accidentally omit the # symbol. This causes the assembler to misinterpret the following expression 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 instruction.

```
LDAA           #$55
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

```
LDX           #$1234

LDY           #$67
```

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 generates the 16-bit value $0067 because the CPU expects a 16-bit value in the instruction stream.

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

In this example, extended addressing is used to access the operand FOO, immediate addressing is used to access the mask value $03, and relative addressing 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.

4.2.4 Direct Addressing Mode

This addressing mode is sometimes called zero-page addressing because it accesses operands in the address range $0000 through $00FF. Since these addresses always begin with $00, only the low byte of the address needs 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 low byte of the operand address is supplied with the instruction and the high byte of the address is assumed to be zero.

```
LDAA          $55
```

The value $55 is taken to be the low byte of an address in the range $0000 through $00FF. The high byte of the address is assumed to be zero. During execution, 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.

```
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, the X index register has the value from address $0020 in its high byte and the value from address $0021 in its low byte.

4.2.5 Extended Addressing Mode

In extended addressing, the full 16-bit address of the memory location to be operated on is provided in the instruction. Extended addressing can access any location in the 64K byte memory map.

```
LDAA          $F03B
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

The value from address $F03B is loaded into the A accumulator.

4.2.6 Relative Addressing Mode

Relative addressing is for branch instructions only. Relative addressing determines the branch destination. The short and long versions of conditional branch instructions use relative addressing exclusively. The branching bit-condition instructions, BRSET and BRCLR, use multiple addressing modes, including relative mode.

A conditional branch instruction tests a status bit in the condition code register. If the bit tests true, execution begins at the destination formed by adding an offset to the address of the memory location after the offset. If the bit does not test true, execution continues with the instruction that follows the branch instruction.

A short conditional branch instruction has an 8-bit opcode and a signed 8-bit relative offset in the byte that follows the opcode. A long conditional branch instruction has an 8-bit prebyte, an 8-bit opcode and a signed 16-bit relative offset in the two bytes that follow the opcode.

A branching bit-condition instruction, BRCLR or BRSET, tests the state of one or more bits in a memory byte. Direct, extended, or indexed addressing can determine the location of the memory byte. The instruction includes an immediate 8-bit mask operand to test the bits and an 8-bit relative offset. If the bits test true, execution begins at the destination formed by adding the 8-bit offset to the address of the memory location after the offset. If the bits do not test true, execution continues with the instruction that follows 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 (–32768) to $7FFF (32767). If the offset is zero, the CPU executes the instruction that follows the branch instruction.

Since the offset is at the end of a branch instruction, using a negative offset value can cause the 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 branching bit-condition instruction to repeat execution until the specified bit condition is satisfied. Since branching bit-condition instructions can consist of four, five, or six bytes depending on the addressing mode used, the offset value that sets up a loop can vary. For instance, an offset of $FC in a 4-byte BRCLR instruction sets up a loop that executes until all the bits in the tested memory byte are clear.

4.2.7 Indexed Addressing Modes

There are seven indexed addressing modes:

- 5-bit constant offset
- Autodecrement/increment
- 9-bit constant offset

- 16-bit constant offset
- 16-bit constant offset indexed-indirect
- Accumulator offset
- Accumulator D offset indexed-indirect

Features of indexed addressing include:

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

4.2.7.1 Indexed Addressing Postbyte

A postbyte follows all indexed addressing opcodes. There may be 0, 1, or 2 extension bytes after the postbyte. The postbyte and extensions do the following tasks:

1. Select a register for indexing (X, Y, SP, PC, A, B, or D)
2. Enable automatic pre- or postincrementing or decrementing of X, Y, or SP and select the pre- or postincrement value
3. Select 5-bit, 9-bit, or 16-bit signed constant offsets

Table 4-2 shows how the postbyte enhances indexed addressing capabilities.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 4-2 Summary of Indexed Operations

5-bit constant offset indexed addressing (IDX)

	7	6	5	4	3	2	1	0
Postbyte:	rr[1]		0	5-bit signed offset				

Effective address = 5-bit signed offset + (X, Y, SP, or PC)

Accumulator offset addressing (IDX)

	7	6	5	4	3	2	1	0
Postbyte:	1	1	1	rr[1]		1	aa[2]	

Effective address = (X, Y, SP, or PC) + (A, B, or D)

Autodecrement/autoincrement) indexed addressing (IDX)

	7	6	5	4	3	2	1	0
Postbyte:	rr[1,3]		1	p[4]	4-bit inc/dec value[5]			

Effective address = (X, Y, or SP) ± 1 to 8

9-bit constant offset indexed addressing (IDX1)

	7	6	5	4	3	2	1	0
Postbyte:	1	1	1	rr[1]		0	0	s[6]

Effective address = s:(offset extension byte) + (X, Y, SP, or PC)

16-bit constant offset indexed addressing (IDX2)

	7	6	5	4	3	2	1	0
Postbyte:	1	1	1	rr[1]		0	1	0

Effective address = (two offset extension bytes) + (X, Y, SP, or PC)

16-bit constant offset indexed-indirect addressing ([IDX2])

	7	6	5	4	3	2	1	0
Postbyte:	1	1	1	rr[1]		0	1	1

(two offset extension bytes) + (X, Y, SP, or PC) is address of pointer to effective address

Accumulator D offset indexed-indirect addressing ([D,IDX])

	7	6	5	4	3	2	1	0
Postbyte:	1	1	1	rr[1]		1	1	1

(X, Y, SP, or PC) + (D) is address of pointer to effective address

NOTES:
1. rr selects X (00), Y (01), SP (10), or PC (11).
2. aa selects A (00), B (01), or D (10).
3. In autoincrement/decrement indexed addressing, PC is not a valid selection.
4. p selects pre- (0) or post- (1) increment/decrement.
5. Increment values range from 0000 (+1) to 0111 (+8). Decrement values range from 1111 (–1) to 1000 (–8).
6. s is the sign bit of the offset extension byte.

All indexed addressing modes use a 16-bit CPU register and additional information to create an indexed address. In most cases the indexed address is the effective address of the instruction, that is, the address of the memory location that the instruction acts on. In indexed-indirect addressing, the indexed address is the location of a value that points to the effective address.

PC offsets are calculated from the location immediately following the current instruction.

```
1000  18 09 C2 20 00 MOVB  $2000  2,PC
1005  A7              NOP
```

This example moves a byte of data from $2000 to $1007.

4.2.7.2 5-Bit Constant Offset Indexed Addressing

This addressing mode calculates the effective address by adding a 5-bit signed offset in the postbyte to the indexing register (X, Y, SP, or PC). The value in the indexing register does not change. The 5-bit signed offset gives a range of –16 through +15 from the value in the indexing register. The majority of indexed instructions use offsets that fit in the 5-bit offset range.

For these examples, assume X contains $1000 and Y contains $2000:

```
LDAA          0,X
```

The value at address $1000 is loaded into A.

```
STAB          -8,Y
```

The value in B is stored at address $2000 – $8, or $1FF8.

4.2.7.3 9-Bit Constant Offset Indexed Addressing

This addressing mode calculates the effective address by adding a 9-bit signed offset in an extension byte to the indexing register (X, Y, SP, or PC). The value in the indexing register does not change. The sign bit of the offset is in the postbyte. The 9-bit offset gives a range of –256 through +255 from the value in the indexing register.

For these examples assume X contains $1000 and Y contains $2000:

```
LDAA          $FF,X
```

The value at address $10FF is loaded into A.

```
LDAB          -20,Y
```

The value at address $2000 – $14, or $1FEC, is loaded into B.

4.2.7.4 16-Bit Constant Offset Indexed Addressing

This addressing mode calculates the effective address by adding a 16-bit offset in two extension bytes to the indexing register (X, Y, SP, or PC). The value in the indexing register does not change. The 16-bit offset allows access to any address in the 64K byte address space. The address bus and the offset are both

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

16 bits, so it does not matter whether the offset is considered to be signed or unsigned ($FFFF may be thought of as +65,535 or as –1).

4.2.7.5 16-Bit Constant Indexed-Indirect Addressing

This addressing mode calculates the address of a pointer to the effective address. It adds a 16-bit offset in two extension bytes to the indexing register (X, Y, SP, or PC). The value in the indexing register does not change. The square brackets distinguish this addressing mode from 16-bit constant offset indexed addressng.

For this example, assume X contains $1000 and the value at address $100A is $2000:

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

The value 10 is added to the value in X to form the address $100A. The CPU fetches the effective address pointer, $2000, from address $100A and loads the value at address $2000 into A.

4.2.7.6 Autodecrement/Autoincrement Indexed Addressing

This addressing mode calculates the effective address by adding an integer value between –8 and –1 or between 1 and 8 to the indexing register (X, Y, or SP). The indexing register retains its changed value.

NOTE: *Autodecrementing and autoincrementing do not apply to the program counter.*

When predecremented or preincremented, the indexing register changes before indexing takes place. When postdecremented or postincremented, the indexing register changes after indexing takes place.

This addressing mode adjusts the indexing value without increasing execution time by using an additional instruction.

In this example, the instruction compares X with the value that X points to and then increments X by one:

```
CPX           1,X+
```

The next two examples are equivalent to common push instructions. In the first example, the instruction predecrements the stack pointer by one and then stores A to the address contained in the stack pointer:

```
STAA          1,–SP  ;equivalent to PSHA
STX           2,–SP  ;equivalent to PSHX
```

The next two examples are equivalent to common pull instructions. In the first example, the instruction loads X from the address in the stack pointer and then postincrements the stack pointer by two:

```
LDX           2,SP+  ;equivalent to PULX
LDAA          1,SP+  ;equivalent to PULA
```

The next 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. The instruction postincrements the source pointer after reading the data from memory and preincrements the destination pointer before accessing memory:

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

Using a predecrement/increment version of LEAS, LEAX, or LEAY when SP, X, or Y is the respective indexing register changes the value in the indexing register. Using a postdecrement/increment version of LEAS, LEAX, LEAY when SP, X, or Y is the respective indexing register has no effect.

4.2.7.7 Accumulator Offset Indexed Addressing

This addressing mode calculates the effective address by adding the value in the indexing register to an unsigned offset value in one of the accumulators. The value in the indexing register is not changed. The indexing register can be X, Y, SP, or PC, and the accumulator can be A, B, or D.

Example:

```
LDAA        B,X
```

This instruction 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 two-instruction combination in an M68HC11.

4.2.7.8 Accumulator D Indexed-Indirect Addressing

This addressing mode calculates address of a pointer to the effective address. It adds the value in D to the value in the indexing register (X, Y, SP, or PC) The value in the indexing register does not change. The square brackets distinguish this addressing mode from D accumulator offset indexing.

In this example, accumulator D indexed-indirect addressing is used in a computed GOTO:

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

The values beginning at GO1 are addresses of potential destinations of the jump 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.

4.2.8 Instructions Using Multiple Modes

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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

4.2.8.1 Move Instructions

Move instructions can use one addressing mode to access the source of the move and another addressing mode to access the destination. 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 instruction is data, not an address. For indexed moves, the indexing register can be X, Y, SP, or PC.

Move instructions do not have indirect modes, or 9-bit or 16-bit offset modes.

4.2.8.2 Bit Manipulation Instructions

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

A BCLR or BSET instruction has an 8-bit mask to clear or set bits in a memory byte. The mask is an immediate value supplied with the instruction. Direct, extended, or indexed addressing determines the location of the memory byte.

A BRCLR or BRSET instruction has an 8-bit mask to test the states of bits in a memory byte. The mask is an immediate value supplied with the instruction. Direct, extended, or indexed addressing determines the location of the memory byte. Relative addressing determines the branch address. A signed 8-bit offset must be supplied with the instruction.

4.3 Instruction Descriptions

A brief discussion of the CPU instructions group by type is given in the subsections below. For a detailed instruction-by-instruction description please consult **Appendix B** of this guide.

4.3.1 Load and Store Instructions

Load instructions copy a value in memory or an immediate value into a CPU register. The value in memory is not changed by the operation. Load instructions (except LEAS, LEAX, and LEAY) affect condition code bits so no separate test instructions are needed to check the loaded values for negative or zero conditions.

Store instructions copy the value in a CPU register to memory. The CPU register value 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.

A summary of the load and store instructions is given in **Table 4-3**.

Table 4-3 Load and Store Instructions

Mnemonic	Function	Operation
LDAA	Load A from memory Load A with immediate value	(M) ⇒ A imm ⇒ A
LDAB	Load B from memory Load B with immediate value	(M) ⇒ B imm ⇒ B
LDD	Load D from memory Load D with immediate value	(M) ⇒ A, (M + 1) ⇒ B imm_H ⇒ A, imm_L ⇒ B
LDS	Load SP from memory Load SP with immediate value	(M) ⇒ SP_H, (M + 1) ⇒ SP_L imm_H ⇒ SP_H, imm_L ⇒ SP_L
LDX	Load X from memory Load X with immediate value	(M) ⇒ X_H, (M + 1) ⇒ X_L imm_H ⇒ X_H, imm_L ⇒ X_L
LDY	Load Y from memory Load Y with immediate value	(M) ⇒ Y_H, (M + 1) ⇒ Y_L imm_H ⇒ Y_H, imm_L ⇒ Y_L
LEAS	Load effective address into SP	Effective address ⇒ SP
LEAX	Load effective address into X	Effective address ⇒ X
LEAY	Load effective address into Y	Effective address ⇒ Y
STAA	Store A in memory	(A) ⇒ M
STAB	Store B in memory	(B) ⇒ M
STD	Store D in memory	(A) ⇒ M, (B) ⇒ M + 1
STS	Store SP in memory	(SP_H) ⇒ M, (SP_L) ⇒ M + 1
STX	Store X in memory	(X_H) ⇒ M, (X_L) ⇒ M + 1
STY	Store Y in memory	(Y_H) ⇒ M, (Y_L) ⇒ M + 1

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.2 Transfer and Exchange Instructions

Transfer instructions copy the value in a CPU register into another CPU register. The source value is not changed by the operation. TFR is a universal transfer instruction, but other mnemonics are accepted for compatibility with the M68HC12. The TAB and TBA instructions affect the N, Z, and V condition code bits in the same way as M68HC12 instructions. The TFR instruction does not affect the condition code bits.

Exchange instructions exchange the values in pairs of CPU registers.

The sign-extend instruction, SEX, is a special case of the universal transfer instruction. It adds a sign extension to an 8-bit two's complement number so that the number can be used in 16-bit operations. The 8-bit number is copied from accumulator A, 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 MSB of the 8-bit number.

A summary of the transfer and exchange instructions is given in **Table 4-4**.

Table 4-4 Transfer and Exchange Instructions

Mnemonic	Function	Operation
TAB	Transfer A to B	(A) ⇒ B
TAP	Transfer A to CCR	(A) ⇒ CCR
TBA	Transfer B to A	(B) ⇒ A
TFR	Transfer register	(A, B, CCR, D, X, Y, or SP) ⇒ A, B, CCR, D, X, Y, or SP
TPA	Transfer CCR to A	(CCR) ⇒ A
TSX	Transfer SP to X	(SP) ⇒ X
TSY	Transfer SP to Y	(SP) ⇒ Y
TXS	Transfer X to SP	(X) ⇒ SP
TYS	Transfer Y to SP	(Y) ⇒ SP
EXG	Exchange registers	(A, B, CCR, D, X, Y, or SP) ⇔ (A, B, CCR, D, X, Y, or SP)
XGDX	Exchange D with X	(D) ⇔ (X)
XGDY	Exchange D with Y	(D) ⇔ (Y)
SEX	Sign-extend 8-bit operand	00:(A, B, or CCR) or FF:(A, B, or CCR) ⇒ D, X, Y, or SP

4.3.3 Move Instructions

These instructions move bytes or words from a source in memory, M_1 or M_1:M_1 + 1, to a destination in memory, M_2 or M_2:M_2 + 1. Six combinations of immediate, extended, and indexed addressing can specify source and destination addresses: IMM/EXT, IMM/IDX, EXT/EXT, EXT/IDX, IDX/EXT, and IDX/IDX. A summary of the move instructions is given in **Table 4-5**.

Table 4-5 Move Instructions

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

4.3.4 Add and Subtract Instructions

Signed and unsigned 8-bit and 16-bit addition and subtraction can be performed on CPU registers, on a CPU register and memory, or on a CPU register and an immediate value. Special instructions support index calculation. Instructions that add or subtract the carry bit, C, in the CCR facilitate multiple precision computation. A summary of the add and subtract instructions is given in **Table 4-6**.

Table 4-6 Add and Subtract Instructions

Mnemonic	Function	Operation
ABA	Add A to B	(A) + (B) ⇒ A
ABX	Add B to X	(B) + (X) ⇒ X
ABY	Add B to Y	(B) + (Y) ⇒ Y
ADCA	Add memory and carry to A Add immediate value and carry to A	(A) + (M) + C ⇒ A (A) + imm + C ⇒ A
ADCB	Add memory and carry to B Add immediate value and carry to B	(B) + (M) + C ⇒ B (B) + imm + C ⇒ B
ADDA	Add memory to A Add immediate value to A	(A) + (M) ⇒ A (A) + imm ⇒ A
ADDB	Add memory to B Add immediate value to B	(B) + (M) ⇒ B (B) + imm ⇒ B
ADDD	Add memory to D Add immediate value to D	(A):(B) + (M):(M + 1) ⇒ A:B (A):(B) + imm ⇒ A:B
SBA	Subtract B from A	(A) – (B) ⇒ A
SBCA	Subtract memory and carry from A Subtract immediate value and carry from A	(A) – (M) – C ⇒ A (A) – imm – C ⇒ A
SBCB	Subtract memory and carry from B Subtract immediate value and carry from B	(B) – (M) – C ⇒ B (B) – imm – C ⇒ B
SUBA	Subtract memory from A Subtract immediate value from A	(A) – (M) ⇒ A (A) – imm ⇒ A
SUBB	Subtract memory from B Subtract immediate value from B	(B) – (M) ⇒ B (B) – imm ⇒ B
SUBD	Subtract memory from D Subtract immediate value from D	(A):(B) – (M):(M + 1) ⇒ A:B (A):(B) – imm ⇒ A:B

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

4.3.5 Binary Coded Decimal Instructions

To add binary coded decimal (BCD) operands, use addition instructions that set the half-carry bit, H, in the CCR. Then adjust the result with the DAA instruction. A summary of the BCD instructions is given in **Table 4-7**.

Table 4-7 BCD Instructions

Mnemonic	Function	Operation
ABA	Add B to A	(A) + (B) ⇒ A
ADCA	Add memory and carry to A Add immediate value and carry to A	(A) + (M) + C ⇒ A (A) + imm + C ⇒ A
ADCB	Add memory and carry to B Add immediate value and carry to B	(B) + (M) + C ⇒ B (B) + imm + C ⇒ B
ADDA	Add memory to A Add immediate value to A	(A) + (M) ⇒ A (A) + imm ⇒ A
ADDB	Add memory to B Add immediate value to B	(B) + (M) ⇒ B (B) + imm ⇒ B
DAA	Decimal adjust A	$(A)_{10}$ ⇒ A

4.3.6 Decrement and Increment Instructions

These instructions are optimized 8-bit and 16-bit addition and subtraction operations. They are used to implement counters. Because they do not affect the carry bit, C, in the CCR, they are particularly well suited for loop counters in multiple-precision computation routines. See **4.3.17.4 Loop Primitive Instructions** for information concerning automatic counter branches. A summary of the decrement and increment instructions is given in **Table 4-8 Decrement and Increment Instructions**.

Table 4-8 Decrement and Increment Instructions

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

4.3.7 Compare and Test Instructions

Compare and test instructions perform subtraction on a pair of CPU registers, on a CPU register and memory, or on a CPU register and an immediate value. The result is not stored, but the operation can affect condition codes in the CCR. These instructions are used to establish conditions for branch instructions. However, most instructions update condition codes automatically, so it is often unnecessary to include separate compare or test instructions. A summary of the compare and test instructions is given in **Table 4-9**.

Table 4-9 Compare and Test Instructions

Mnemonic	Function	Operation
CBA	Compare A to B	(A) – (B)
CMPA	Compare A to memory Compare A to immediate value	(A) – (M) (A) – imm
CMPB	Compare B to memory Compare B to immediate value	(B) – (M) (B) – imm
CPD	Compare D to memory Compare D to immediate value	(A):(B) – (M):(M + 1) (A):(B) – imm
CPS	Compare SP to memory Compare SP to immediate value	(SP) – (M):(M + 1) (SP) – imm
CPX	Compare X to memory Compare X to immediate value	(X) – (M):(M + 1) (X) – imm
CPY	Compare Y to memory Compare Y to immediate value	(Y) – (M):(M + 1) (Y) – imm
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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.8 Boolean Logic Instructions

These instructions perform a logic operation on the A or B accumulator and a memory value or immediate value, or on the CCR and an immediate value. A summary of the boolean logic instructions is given in **Table 4-10**.

Table 4-10 Boolean Logic Instructions

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

4.3.9 Clear, Complement, and Negate Instructions

These instructions perform binary operations on values in an accumulator or in memory. Clear operations clear the value, complement operations replace the value with its one's complement, and negate operations replace the value with its two's complement. A summary of the clear, complement and negate instructions is given in **Table 4-11**.

Table 4-11 Clear, Complement, and Negate Instructions

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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.10 Multiply and Divide Instructions

The multiply instructions perform signed and unsigned, 8-bit and 16-bit multiplication. An 8-bit multiplication gives a 16-bit product. A 16-bit multiplication gives a 32-bit product.

An integer divide or fractional divide instruction has a 16-bit dividend, divisor, quotient, and remainder. Extended divide instructions use a 32-bit dividend and a 16-bit divisor to produce a 16-bit quotient and a 16-bit remainder.

A summary of the multiply and divide instructions is given in **Table 4-12**.

Table 4-12 Multiplication and Division Instructions

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

4.3.11 Bit Test and Bit Manipulation Instructions

These operations use a mask value to test or change the value of individual bits in an accumulator or in memory. BITA and BITB provide a convenient means of testing bits without altering the value of either operand. A summary of the bit test and bit manipulation instructions is given in **Table 4-13**.

Table 4-13 Bit Test and Bit Manipulation Instructions

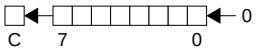
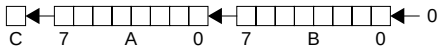
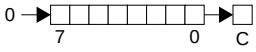
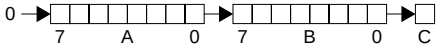
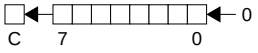
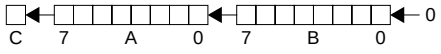
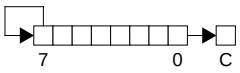
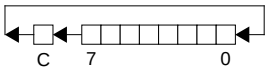
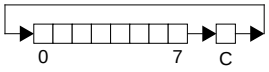
Mnemonic	Function	Operation
BCLR	Clear bit(s) in memory	(M) • mask byte ⇒ M
BITA	Bit test A	(A) • (M)
BITB	Bit test B	(B) • (M)
BSET	Set bits in memory	(M) + mask byte ⇒ M

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.12 Shift and Rotate Instructions

There are shifts and rotates for accumulators and memory bytes. For multiple-byte operations, all shifts and rotates pass the shifted-out bit through the carry bit, C. Because logical and arithmetic left shifts are identical, there are no separate logical left shift operations. LSL mnemonics are assembled as ASL operations. A summary of the shift and rotate instructions is given in **Table 4-14**.

Table 4-14 Shift and Rotate Instructions

Mnemonic	Function	Operation
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	
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	
ROL ROLA ROLB	Rotate left memory Rotate left A Rotate left B	
ROR RORA RORB	Rotate right memory Rotate right A Rotate right B	

4.3.13 Fuzzy Logic Instructions

The instruction set supports efficient processing of fuzzy logic operations. A summary of the fuzzy logic instructions is given in **Table 4-15**.

Table 4-15 Fuzzy Logic Instructions

Mnemonic	Function	Operation
MEM	Membership evaluation	μ (grade) $\Rightarrow M_{(Y)}$, (X) + 4 $\Rightarrow$ X, (Y) + 1 $\Rightarrow$ Y, A unchanged If (A) < P1 or (A) > P2, then μ = 0, else μ = MIN [((A) – P1) $\times$ S1, (P2 – (A)) $\times$ S2, $FF] A contains current crisp input value. X points to 4-byte data structure describing trapezoidal membership function as base intercept points and slopes (P1, P2, S1, S2). Y points to fuzzy input (RAM location).
REV	MIN-MAX rule evaluation	Find smallest rule input (MIN). Store to rule outputs unless fuzzy output is larger (MAX). Rules are unweighted. Each rule input is 8-bit offset from base address in Y. Each rule output is 8-bit offset from base address in Y. $FE separates rule inputs from rule outputs. $FF terminates rule list. REV can be interrupted.
REVW	Weighted MIN-MAX rule evaluation	Find smallest rule input (MIN). Multiply by rule-weighting factor (optional). Store to rule outputs unless fuzzy output is larger (MAX). Each rule input is 16-bit address of a fuzzy input. Each rule output is 16-bit address of fuzzy output. Address $FFFE separates rule inputs from rule outputs. $FFFF terminates rule list. Weights are 8-bit values in separate table. REVW can be interrupted.
WAV	Weighted average calculation	Calculate numerator (sum of products) and denominator (sum of weights). $\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ $\sum_{i=1} F_i \Rightarrow X$ Put results in correct CPU registers for EDIV immediately after WAV.
wavr	Return to interrupted WAV instruction	Recover intermediate results from stack rather than initializing to zero.

4.3.14 Maximum and Minimum Instructions

4.3.14.1 Fuzzy Logic Membership Instruction

The MEM instruction is used during the fuzzification process. During fuzzification, current system input values are compared to 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.14.2 Fuzzy Logic Rule Evaluation Instructions

The 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. REV and REVW 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.

4.3.14.3 Fuzzy Logic Averaging Instruction

The WAV instruction calculates weighted averages. In order 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 CPU typically uses singletons for output membership functions rather than the trapezoidal shapes used for inputs. As with inputs, the x-axis represents the range of possible values for a system output. Singleton membership functions consist of the x-axis position for a label of the system output. Fuzzy outputs correspond to the y-axis height of the corresponding output membership function. The WAV instruction calculates the numerator and denominator sums for a weighted average of the fuzzy outputs.

Because WAV requires a number of cycles to execute, it can be interrupted. The wavr pseudoinstruction causes execution to resume at the point where it was interrupted.

These instructions make comparisons between an accumulator and a memory location. They 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 or in the memory location. A summary of the minimum and maximum instructions is given in **Table 4-16**.

Table 4-16 Minimum and Maximum Instructions

Mnemonic	Function	Operation
EMIND	Put smaller of two unsigned 16-bit values in D	MIN [(D), (M):(M + 1)] ⇒ D
EMINM	Put smaller of two unsigned 16-bit values in memory	MIN [(D), (M):(M + 1)] ⇒ M:M + 1
MINA	Put smaller of two unsigned 8-bit values in A	MIN [(A), (M)] ⇒ A
MINM	Put smaller of two unsigned 8-bit values in memory	MIN [(A), (M)] ⇒ M
EMAXD	Put larger of two unsigned 16-bit values in D	MAX [(D), (M):(M + 1)] ⇒ D
EMAXM	Put larger of two unsigned 16-bit values in memory	MAX [(D), (M):(M + 1)] ⇒ M:M + 1
MAXA	Put larger of two unsigned 8-bit values in A	MAX [(A), (M)] ⇒ A
MAXM	Put larger of two unsigned 8-bit values in memory	MAX[(A), (M)] ⇒ M

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.15 Multiply and Accumulate Instruction

The 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-bit 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. A summary of the multiply and accumulate instructions is given in **Table 4-17**.

Table 4-17 Multiply and Accumulate Instruction

Mnemonic	Function	Operation
EMACS	Multiply and accumulate (signed) 16 × 16 bit ⇒ 32 bit	$(M_X):(M_{X+1}) \times (M_Y):M_{Y+1}) + (M):(M+1):(M+2):(M+3) \Rightarrow M:M+1:M+2:M+3$

4.3.16 Table Interpolation Instructions

The TBL and ETBL instructions 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. 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. Indexed addressing modes provide flexibility in structuring tables.

Consider each of the successive values stored in a table as 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 value in B to obtain an intermediate delta-y term. The result (stored in the A accumulator by TBL, in the D accumulator by ETBL) is the y-value of the beginning point plus the signed intermediate delta-y value.

A summary of the table interpolation instructions is given in **Table 4-18**.

Table 4-18 Table Interpolation Instructions

Mnemonic	Function	Operation
ETBL	16-bit table lookup and interpolate (indirect addressing not allowed)	$(M):(M+1) + [(B) \times [(M+2):(M+3) - (M):(M+1)]] \Rightarrow D$ Initialize B, and index before ETBL. Effective address points to the first 16-bit table entry (M):(M + 1) B is fractional part of lookup value
TBL	8-bit table lookup and interpolate (indirect addressing not allowed)	$(M) + [(B) \times [(M+1) - (M)]] \Rightarrow A$ Initialize B, and index before TBL. Effective address points to the first 8-bit table entry (M) B is fractional part of lookup value.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.17 Branch Instructions

A branch instruction causes a program sequence change when specific conditions exist. There are three types of branch instructions: short, long, and bit-conditional.

Branch instructions can also be classified by the type of condition that must be satisfied in order for a branch to be taken:

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

Some branch instructions belong to more than one type.

4.3.17.1 Short Branch Instructions

When a specified condition is met, a short branch instruction adds a signed 8-bit offset 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. A summary of the short branch instructions is given in **Table 4-19**.

Table 4-19 Short Branch Instructions

Mnemonic	Type	Function	Condition Equation
BRA	Unary	Branch always	1 = 1
BRN		Branch never	1 = 0
BCC	Simple	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
BHI	Unsigned	Branch if higher (R > M)	C + Z = 0
BHS		Branch if higher or same (R ≥ M)	C = 0
BLO		Branch if lower (R < M)	C = 1
BLS		Branch if lower or same (R ≤ M)	C + Z = 1
BGE	Signed	Branch if greater than or equal (R ≥ M)	N ⊕ V = 0
BGT		Branch if greater than (R > M)	Z + (N ⊕ V) = 0
BLE		Branch if less than or equal (R ≤ M)	Z + (N ⊕ V) = 1
BLT		Branch if less than (R < M)	N ⊕ V = 1

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.17.2 Long Branch Instructions

When a specified condition is met, a long branch instruction adds a signed 16-bit offset 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 64K byte address map to any other location in the map. A summary of the long branch instructions is given in **Table 4-20**.

Table 4-20 Long Branch Instructions

Mnemonic	Class	Function	Condition Equation
LBRA	Unary	Long branch always	1 = 1
LBRN		Long branch never	1 = 0
LBCC	Simple	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
LBHI	Unsigned	Long branch if higher (R > M)	C + Z = 0
LBHS		Long branch if higher or same (R ≥ M)	C = 0
LBLO		Long branch if lower (R < M)	Z = 1
LBLS		Long branch if lower or same (R ≤ M)	C + Z = 1
LBGE	Signed	Long branch if greater than or equal (R ≥ M)	N ⊕ V = 0
LBGT		Long branch if greater than (R > M)	Z + (N ⊕ V) = 0
LBLE		Long branch if less than or equal (R ≤ M)	Z + (N ⊕ V) = 1
LBLT		Long branch if less than (R < M)	N ⊕ V = 1

4.3.17.3 Bit Condition Branch Instructions

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. A summary of the bit condition branch instructions is given in **Table 4-21**.

Table 4-21 Bit Condition Branch Instructions

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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.17.4 Loop Primitive Instructions

Loop primitive instructions test a counter value in a CPU register (A, B, D, X, Y, or SP) for a zero or nonzero value as a branch condition. There are predecrement, preincrement and test-only versions of these instructions. The numeric range of 9-bit offset values is –256 to +255 from the address of the next memory location after the offset value. A summary of the loop primitive instructions is given in **Table 4-22**.

Table 4-22 Loop Primitive Instructions

Mnemonic	Function	Operation
DBEQ	Decrement counter and branch if zero	(counter) – 1 ⇒ counter If (counter) = 0, then branch, else continue to next instruction
DBNE	Decrement counter and branch if not zero	(counter) – 1 ⇒ counter If (counter) ≠ 0, then branch, else continue to next instruction
IBEQ	Increment counter and branch if zero	(counter) + 1 ⇒ counter If (counter) = 0, then branch, else continue to next instruction
IBNE	Increment counter and branch if not zero	(counter) + 1 ⇒ counter If (counter) ≠ 0, then branch, else continue to next instruction
TBEQ	Test counter and branch if zero	If (counter) = 0, then branch, else continue to next instruction
TBNE	Test counter and branch if not zero	If (counter) ≠ 0, then branch, else continue to next instruction

4.3.18 Jump and Subroutine Instructions

Jump instructions cause immediate changes in program sequence. The JMP instruction loads the PC with an address in the 64K byte 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 transfer control to a code segment that performs a particular task. A short branch to subroutine (BSR), a jump to subroutine (JSR), or an expanded-memory call (CALL) can be used to initiate subroutines. There is no long branch to subroutine instruction (LBSR), 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 64K byte address space are terminated with an RTS instruction. RTS unstacks the return address so that execution resumes with the instruction after BSR or JSR.

The 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 RTC instruction ends 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 operate correctly on devices that do not have expanded addressing capability.

A summary of the jump and subroutine instructions is given in **Table 4-23**.

Table 4-23 Jump and Subroutine Instructions

Mnemonic	Function	Operation
BSR	Branch to subroutine	(SP) – $0002 ⇒ SP, RTN_H:RTN_L ⇒ M_{SP}:M_{SP+1}, subroutine address ⇒ PC
CALL	Call subroutine in expanded memory	(SP) – $0002 ⇒ SP, RTN_H:RTN_L ⇒ M_{SP}:M_{SP+1},(SP) – $0001 ⇒ SP, (PPAGE) ⇒ M_{SP}, page ⇒ PPAGE, subroutine address ⇒ PC
JMP	Jump	Subroutine address ⇒ PC
JSR	Jump to subroutine	(SP) – $0002 ⇒ SP, RTN_H:RTN_L ⇒ M_{SP}:M_{SP+1}, subroutine address ⇒ PC
RTS	Return from subroutine	(M_{SP}) ⇒ PPAGE, (SP) + $0001 ⇒ SP, ($M_{SP}$):($M_{SP+1}$) ⇒ PC_H:PC_L, (SP) + $0002 ⇒ SP
RTC	Return from call	(M_{SP}):(M_{SP+1}) ⇒ PC_H:PC_L, (SP) + $0002 ⇒ SP

4.3.19 Interrupt Instructions

Interrupt instructions handle transfer of control to and from interrupt service routines.

The SWI instruction initiates a software interrupt. It stacks the return address and the values in the CPU registers. Then execution begins at the address pointed to by the SWI vector.

The SWI instruction causes an interrupt without an interrupt request. The global mask bits I and X in the CCR do not inhibit SWI. SWI sets the I bit, inhibiting maskable interrupts until the I bit is cleared.

The TRAP instruction The CPU 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. A summary of the interrupt instructions is given in **Table 4-24**.

Table 4-24 Interrupt Instructions

Mnemonic	Function	Operation
RTI	Return from interrupt	(M_{SP}) ⇒ CCR, (SP) + $0001 ⇒ SP ($M_{SP}$):($M_{SP+1}$) ⇒ B:A, (SP) + $0002 ⇒ SP ($M_{SP}$):($M_{SP+1}$) ⇒ X_H:X_L, (SP) + $0004 ⇒ SP ($M_{SP}$):($M_{SP+1}$) ⇒ PC_H:PC_L, (SP) + $0002 ⇒ SP ($M_{SP}$):($M_{SP+1}$) ⇒ Y_H:Y_L, (SP) + $0004 ⇒ SP
SWI	Software interrupt	(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}, 1 ⇒ I
TRAP		

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.20 Index Manipulation Instructions

Index manipulation instructions perform 8-bit and 16-bit operations on CPU registers or memory. A summary of the index manipulation instructions is given in **Table 4-25**.

Table 4-25 Index Manipulation Instructions

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

4.3.21 Stacking Instructions

There are two types of stacking instructions:

- Stack pointer manipulation
- Stack operation (saving and retrieving CPU register contents)

A summary of the stacking instructions is given in **Table 4-26**.

Table 4-26 Stacking Instructions

Mnemonic	Type	Function	Operation
CPS	Stack pointer manipulation	Compare SP to memory	(SP) – (M):(M + 1)
DES		Decrement SP	(SP) – $0001 ⇒ SP
INS		Increment SP	(SP) + $0001 ⇒ SP
LDS		Load SP	(M):(M + 1) ⇒ SP
LEAS		Load effective address into SP	Effective address ⇒ SP
STS		Store SP	(SP) ⇒ M:M + 1
TSX		Transfer SP to X	(SP) ⇒ X
TSY		Transfer SP to Y	(SP) ⇒ Y
TXS		Transfer X to SP	(X) ⇒ SP
TYS		Transfer Y to SP	(Y) ⇒ SP
PSHA	Stack operation	Push A	(SP) – $0001 ⇒ SP, (A) ⇒ M_{SP}
PSHB		Push B	(SP) – $0001 ⇒ SP, (B) ⇒ M_{SP}
PSHC		Push CCR	(SP) – $0001 ⇒ SP, (CCR) ⇒ M_{SP}
PSHD		Push D	(SP) – $0002 ⇒ SP, (A):(B) ⇒ M_{SP}:$M_{SP + 1}$
PSHX		Push X	(SP) – $0002 ⇒ SP, (X) ⇒ M_{SP}:$M_{SP + 1}$
PSHY		Push Y	(SP) – $0002 ⇒ SP, (Y) ⇒ M_{SP}:$M_{SP + 1}$
PULA		Pull A	(M_{SP}) ⇒ A, (SP) + 1 ⇒ SP
PULB		Pull B	(M_{SP}) ⇒ B, (SP) + 1 ⇒ SP
PULC		Pull CCR	(M_{SP}) ⇒ CCR, (SP) + 1 ⇒ SP
PULD		Pull D	(M_{SP}):($M_{SP + 1}$) ⇒ A:B, (SP) + 2 ⇒ SP
PULX		Pull X	(M_{SP}):($M_{SP + 1}$) ⇒ X, (SP) + 2 ⇒ SP
PULY		Pull Y	(M_{SP}):($M_{SP + 1}$) ⇒ Y, (SP) + 2 ⇒ SP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.22 Load Effective Address Instructions

Load effective address instructions add a constant or the value in an accumulator to the value in an index register, the stack pointer, or the program counter. The constant can be a 5-, 8-, or 16-bit value. The accumulator can be A, B, or D. A summary of the load effective address instructions is given in **Table 4-27**.

Table 4-27 Load Effective Address Instructions

Mnemonic	Function	Operation
LEAS	Load effective address into SP	(X), (Y), (SP), or (PC) ± constant ⇒ SP (X, (Y), (SP), or (PC) + (A, B, or D) ⇒ SP
LEAX	Load effective address into X	(X), (Y), (SP), or (PC) ± constant ⇒ X (X), (Y), (SP), or (PC) + (A, B, or D) ⇒ X
LEAY	Load effective address into Y	(X), (Y), (SP), or (PC) ± constant ⇒ Y (R) + (A), (B), or (D) ⇒ Y

4.3.23 Condition Code Instructions

A summary of the condition code instructions is given in **Table 4-28**.

Table 4-28 Condition Code Instructions

Mnemonic	Function	Operation
ANDCC	Logical AND CCR with immediate value	(CCR) • imm ⇒ CCR
CLC	Clear C bit	0 ⇒ C
CLI	Clear I bit	0 ⇒ I
CLV	Clear V bit	0 ⇒ V
ORCC	Logical OR CCR with immediate value	(CCR) + imm ⇒ CCR
PSHC	Push CCR onto stack	(SP) – $0001 ⇒ SP, (CCR) ⇒ M_{SP}
PULC	Pull CCR from stack	(M_{SP}) ⇒ CCR, (SP) + $0001 ⇒ SP
SEC	Set C bit	1 ⇒ C
SEI	Set I bit	1 ⇒ I
SEV	Set V bit	1 ⇒ V
TAP	Transfer A to CCR	(A) ⇒ CCR
TPA	Transfer CCR to A	(CCR) ⇒ A

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.3.24 STOP and WAI Instructions

The STOP and WAI instructions put the MCU in a standby state to reduce power consumption.

The STOP instruction stacks a return address and the values in the CPU registers, then stops all system clocks, halting program execution. A reset or an external interrupt request recovers the stacked values and restarts the system clocks, and program execution resumes.

The WAI instruction stacks a return address and the values in the CPU registers, then stops the CPU clocks, halting program execution. A reset or any enabled interrupt request recovers the stacked values and restarts the CPU clocks, and program execution resumes.

Although recovery from STOP or WAI takes the same number of clock cycles, restarting after STOP requires extra time for the oscillator to reach operating speed.

A summary of the STOP and WAI instructions is given in **Table 4-29**.

Table 4-29 STOP and WAI Instructions

Mnemonic	Function	Operation
STOP	Stop	(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
WAI	Wait for interrupt	(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 CPU clocks

4.3.25 Background Mode and Null Operation Instructions

Executing the BGND instruction when BDM is enabled puts the MCU in background debug mode for system development and debugging.

Null operations are often used to replace other instructions during software debugging. Replacing conditional branch instructions with BRN, for instance, permits testing a decision-making routine without actually taking the branches.

A summary of the background mode and null operation instructions is given in **Table 4-30**.

Table 4-30 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	Does nothing

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

4.4 High-Level Language Support

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. Historically, the most serious drawback to the use of HLL in microcontrollers has been the relatively large size of programs written in HLL. Larger program memory space size requirements translate into increased system costs.

Motorola solicited the cooperation of third-party software developers to assure that the STAR12 instruction set would meet the needs of a more efficient generation of compilers. Several features of the STAR12 were specifically designed to improve the efficiency of compiled HLL, and thus minimize cost.

This subsection identifies STAR12 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 STAR12 instruction set is a superset of the M68HC11 instruction set, some of the discussions use the M68HC11 as a basis for comparison.

4.4.1 Data Types

The STAR12 CPU supports the bit-sized data type with bit-manipulation instructions that 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 STAR12 CPU can easily handle 16-bit integer types and the set of conditional branches, including long branches, allows 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.

Special sign-extension instructions 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.

4.4.2 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, that stack pointer-based indexing should be allowed, and that direct arithmetic manipulation of the stack pointer value should be allowed. The STAR12 instruction set provides 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.

4.4.2.1 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 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.

4.4.2.2 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 the following 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. Postincrement 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.

4.4.2.3 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 midframe position between the locals and the parameters before returning to the calling program. The STAR12 CPU requires only the execution of TFR X,S to deallocate the local storage and return.

The concept of a frame pointer is supported in the STAR12 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.4.3 Increment and Decrement Operators

In C, the notation + + i or i – – is often used to form loop counters. Within limited constraints, the STAR12 loop primitives can speed up the loop-count-and-branch function.

The STAR12 includes a set of six basic loop-control instructions that decrement, increment, or test a loop-count register and then branch if the register 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 postincrement operation can be combined with an indexed instruction to eliminate the cost of the increment operation. This is typically done by postcompile optimization because the indexed instruction that could absorb the increment/decrement operation may not be apparent at compile time.

4.4.4 Higher Math Functions

In the STAR12 CPU, 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 CPU can use native instructions instead of relatively large, slow routines.

4.4.5 Conditional If Constructs

In the STAR12 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 STAR12 includes branch instructions that perform conditional branching based on the state of the indicators in the condition code 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 64K byte map.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.4.6 Case and Switch Statements

Case and switch statements (and computed GOTOs) can use PC-relative indexed-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 indexed-indirect addressing.

4.4.7 Pointers

The STAR12 supports pointers with direct arithmetic operations on the 16-bit index registers (LEAS, LEAX, and LEAY instructions) and with indexed-indirect addressing modes.

4.4.8 Function Calls

Bank switching is a common way of adapting a CPU with a 16-bit address bus to accommodate more than 64K bytes of program memory space. One of the most significant drawbacks of this technique is the requirement of masking interrupts while the bank page value is being changed. Another problem is that the physical location of the bank page register can change from one system 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 STAR12 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 STAR12 has dedicated signal lines that allow the CPU to access the bank page register without having to use an address in the normal 64K byte 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. A CALL/RTC pair can be used to access any function subroutine on any page. But when the called subroutine is on the current page or in an area of memory that is always visible, it is more efficient to access it with JSR/RTS instructions.

Push and pull instructions can be used to stack some or all the CPU registers during a function call. The STAR12 CPU can push and pull any of the CPU registers A, B, D, CCR, X, Y, or SP.

4.4.9 Instruction Set Orthogonality

One very helpful aspect of the STAR12 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, and would include both signed and unsigned versions of all mathematical instructions. Greater regularity of the instruction set makes it

possible to implement compilers more efficiently because operation is more consistent, and fewer special cases must be handled.

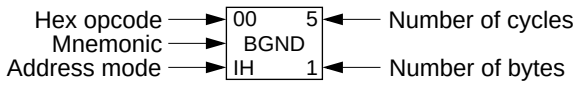
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

4.5 Opcode Map

00 5 BGND IH 1	10 1 ANDCC IM 2	20 3 BRA RL 2	30 3 PULX IH 1	40 1 NEGA IH 1	50 1 NEGB IH 1	60 3–6 NEG ID 2–4	70 4 NEG EX 3	80 1 SUBA IM 2	90 3 SUBA DI 2	A0 3/4/6 SUBA ID 2–4	B0 3 SUBA EX 3	C0 1 SUBB IM 2	D0 3 SUBB DI 2	E0 3/4/6 SUBB ID 2–4	F0 3 SUBB EX 3
01 5 MEM IH 1	11 11 EDIV IH 1	21 1 BRN RL 2	31 3 PULY IH 1	41 1 COMA IH 1	51 1 COMB IH 1	61 3–6 COM ID 2–4	71 4 COM EX 3	81 1 CMPA IM 2	91 3 CMPA DI 2	A1 3/4/6 CMPA ID 2–4	B1 3 CMPA EX 3	C1 1 CMPB IM 2	D1 3 CMPB DI 2	E1 3/4/6 CMPB ID 2–4	F1 3 CMPB EX 3
02 1 INY IH 1	12 1 MUL IH 1	22 3/1 BHI RL 2	32 3 PULA IH 1	42 1 INCA IH 1	52 1 INCB IH 1	62 3–6 INC ID 2–4	72 4 INC EX 3	82 1 SBCA IM 2	92 3 SBCA DI 2	A2 3/4/6 SBCA ID 2–4	B2 3 SBCA EX 3	C2 1 SBCB IM 2	D2 3 SBCB DI 2	E2 3/4/6 SBCB ID 2–4	F2 3 SBCB EX 3
03 1 DEY IH 1	13 3 EMUL IH 1	23 3/1 BLS RL 2	33 3 PULB IH 1	43 1 DECA IH 1	53 1 DECB IH 1	63 3–6 DEC ID 2–4	73 4 DEC EX 3	83 2 SUBD IM 3	93 3 SUBD DI 2	A3 3/4/6 SUBD ID 2–4	B3 3 SUBD EX 3	C3 2 ADDD IM 3	D3 3 ADDD DI 2	E3 3/4/6 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 3/4/6 ANDA ID 2–4	B4 3 ANDA EX 3	C4 1 ANDB IM 2	D4 3 ANDB DI 2	E4 3/4/6 ANDB ID 2–4	F4 3 ANDB EX 3
05 3/4/6 JMP ID 2–4	15 4/5/7 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 3/4/6 BITA ID 2–4	B5 3 BITA EX 3	C5 1 BITB IM 2	D5 3 BITB DI 2	E5 3/4/6 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 3/4/6 LDAA ID 2–4	B6 3 LDAA EX 3	C6 1 LDAB IM 2	D6 3 LDAB DI 2	E6 3/4/6 LDAB ID 2–4	F6 3 LDAB EX 3
07 4 BSR RL 2	17 4 JSR DI 2	27 3/1 BEQ RL 2	37 2 PSHB IH 1	47 1 ASRA IH 1	57 1 ASRB IH 1	67 3–6 ASR ID 2–4	77 4 ASR EX 3	87 1 CLRA IH 1	97 1 TSTA IH 1	A7 1 NOP IH 1	B7 1 TFR/EXG IH 2	C7 1 CLRB IH 1	D7 1 TSTB IH 1	E7 3/4/6 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 3/4/6 EORA ID 2–4	B8 3 EORA EX 3	C8 1 EORB IM 2	D8 3 EORB DI 2	E8 3/4/6 EORB ID 2–4	F8 3 EORB EX 3
09 1 DEX IH 1	19 2 LEAY ID 2–4	29 3/1 BVS RL 2	39 2 PSHC IH 1	49 1 LSRD IH 1	59 1 ASLD IH 1	69 2–4 CLR ID 2–4	79 3 CLR EX 3	89 1 ADCA IM 2	99 3 ADCA DI 2	A9 3/4/6 ADCA ID 2–4	B9 3 ADCA EX 3	C9 1 ADCB IM 2	D9 3 ADCB DI 2	E9 3/4/6 ADCB ID 2–4	F9 3 ADCB EX 3
0A 7 RTC IH 1	1A 2 LEAX ID 2–4	2A 3/1 BPL RL 2	3A 3 PULD IH 1	4A 7 CALL EX 4	5A 2 STAA DI 2	6A 2–4 STAA ID 2–4	7A 3 STAA EX 3	8A 1 ORAA IM 2	9A 3 ORAA DI 2	AA 3/4/6 ORAA ID 2–4	BA 3 ORAA EX 3	CA 1 ORAB IM 2	DA 3 ORAB DI 2	EA 3/4/6 ORAB ID 2–4	FA 3 ORAB EX 3
0B 8/11 RTI IH 1	1B 2 LEAS ID 2–4	2B 3/1 BMI RL 2	3B 2 PSHD IH 1	4B 7/8/10 CALL ID 2–5	5B 2 STAB DI 2	6B 2–4 STAB ID 2–4	7B 3 STAB EX 3	8B 1 ADDA IM 2	9B 3 ADDA DI 2	AB 3/4/6 ADDA ID 2–4	BB 3 ADDA EX 3	CB 1 ADDB IM 2	DB 3 ADDB DI 2	EB 3/4/6 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 2–4 STD ID 2–4	7C 3 STD EX 3	8C 2 CPD IM 3	9C 3 CPD DI 2	AC 3/4/6 CPD ID 2–4	BC 3 CPD EX 3	CC 2 LDD IM 3	DC 3 LDD DI 2	EC 3/4/6 LDD ID 2–4	FC 3 LDD EX 3
0D 4/6 BCLR ID 3–5	1D 4 BCLR EX 4	2D 3/1 BLT RL 2	3D 5 RTS IH 1	4D 4 BCLR DI 3	5D 2 STY DI 2	6D 2–4 STY ID 2–4	7D 3 STY EX 3	8D 2 CPY IM 3	9D 3 CPY DI 2	AD 3/4/6 CPY ID 2–4	BD 3 CPY EX 3	CD 2 LDY IM 3	DD 3 LDY DI 2	ED 3/4/6 LDY ID 2–4	FD 3 LDY EX 3
0E 4–6 BRSET ID 4–6	1E 5 BRSET EX 5	2E 3/1 BGT RL 2	3E 7+6 WAI IH 1	4E 4 BRSET DI 4	5E 2 STX DI 2	6E 2–4 STX ID 2–4	7E 3 STX EX 3	8E 2 CPX IM 3	9E 3 CPX DI 2	AE 3/4/6 CPX ID 2–4	BE 3 CPX EX 3	CE 2 LDX IM 3	DE 3 LDX DI 2	EE 3/4/6 LDX ID 2–4	FE 3 LDX EX 3
0F 4–6 BRCLR ID 4–6	1F 5 BRCLR EX 5	2F 3/1 BLE RL 2	3F 9 SWI IH 1	4F 4 BRCLR DI 4	5F 2 STS DI 2	6F 2–4 STS ID 2–4	7F 3 STS EX 3	8F 2 CPS IM 3	9F 3 CPS DI 2	AF 3/4/6 CPS ID 2–4	BF 3 CPS EX 3	CF 2 LDS IM 3	DF 3 LDS DI 2	EF 3/4/6 LDS ID 2–4	FF 3 LDS EX 3

Opcode $04 is for one of the loop primitive instructions DBEQ, DBNE, IBNE, TBEQ, or TBNE.

Address mode abbreviations: DI — direct IH — inherent SP — special
EX — extended IM — immediate
ID — indexed RL — relative

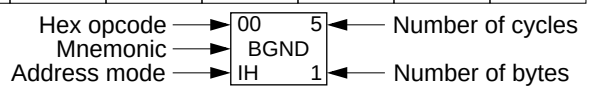


PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

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 4/5/7 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 4/5/7 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 4/5/7 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 4/5/7 EMIND ID 3–5	2B 4/3 LBMI RL 4	3B 5n/3n REVW SP 2	4B 10 TRAP IH 2	5B 10 TRAP IH 2	6B 10 TRAP IH 2	7B 10 TRAP IH 2	8B 10 TRAP IH 2	9B 10 TRAP IH 2	AB 10 TRAP IH 2	BB 10 TRAP IH 2	CB 10 TRAP IH 2	DB 10 TRAP IH 2	EB 10 TRAP IH 2	FB 10 TRAP IH 2
0C 6 MOVB EX–EX 6	1C 4–7 MAXM ID 3–5	2C 4/3 LBGE RL 4	3C 7n 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 6 TBL ID 3	4D 10 TRAP IH 2	5D 10 TRAP IH 2	6D 10 TRAP IH 2	7D 10 TRAP IH 2	8D 10 TRAP IH 2	9D 10 TRAP IH 2	AD 10 TRAP IH 2	BD 10 TRAP IH 2	CD 10 TRAP IH 2	DD 10 TRAP IH 2	ED 10 TRAP IH 2	FD 10 TRAP IH 2
0E 2 TAB IH 2	1E 4–7 EMAXM ID 3–5	2E 4/3 LBGT RL 4	3E 8+6 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 EX 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

Address mode abbreviations: DI — direct, EX — extended, ID — indexed, IH — inherent, IM — immediate, RL — relative, SP — special



4.6 Transfer and Exchange Postbyte Encoding

Transfers									
↓LS	MS→	0	1	2	3	4	5	6	7
0		A⇒A	B⇒A	CCR⇒A	$TMP3_L$⇒A	B⇒A	X_L⇒A	Y_L⇒A	SP_L⇒A
1		A⇒B	B⇒B	CCR⇒B	$TMP3_L$⇒B	B⇒B	X_L⇒B	Y_L⇒B	SP_L⇒B
2		A⇒CCR	B⇒CCR	CCR⇒CCR	$TMP3_L$⇒CCR	B⇒CCR	X_L⇒CCR	Y_L⇒CCR	SP_L⇒CCR
3		sex:A⇒TMP2	sex:B⇒TMP2	sex:CCR⇒TMP2	TMP3⇒TMP2	D⇒TMP2	X⇒TMP2	Y⇒TMP2	SP⇒TMP2
4		sex:A⇒D SEX A,D	sex:B⇒D SEX B,D	sex:CCR⇒D SEX CCR,D	TMP3⇒D	D⇒D	X⇒D	Y⇒D	SP⇒D
5		sex:A⇒X SEX A,X	sex:B⇒X SEX B,X	sex:CCR⇒X SEX CCR,X	TMP3⇒X	D⇒X	X⇒X	Y⇒X	SP⇒X
6		sex:A⇒Y SEX A,Y	sex:B⇒Y SEX B,Y	sex:CCR⇒Y SEX CCR,Y	TMP3⇒Y	D⇒Y	X⇒Y	Y⇒Y	SP⇒Y
7		sex:A⇒SP SEX A,SP	sex:B⇒SP SEX B,SP	sex:CCR⇒SP SEX CCR,SP	TMP3⇒SP	D⇒SP	X⇒SP	Y⇒SP	SP⇒SP
Exchanges									
↓LS	MS→	8	9	A	B	C	D	E	F
0		A⇔A	B⇔A	CCR⇔A	$TMP3_L$⇒A $00:A⇒TMP3	B⇒A A⇒B	X_L⇒A $00:A⇒X	Y_L⇒A $00:A⇒Y	SP_L⇒A $00:A⇒SP
1		A⇔B	B⇔B	CCR⇔B	$TMP3_L$⇒B $FF:B⇒TMP3	B⇒B $FF⇒A	X_L⇒B $FF:B⇒X	Y_L⇒B $FF:B⇒Y	SP_L⇒B $FF:B⇒SP
2		A⇔CCR	B⇔CCR	CCR⇔CCR	$TMP3_L$⇒CCR $FF:CCR⇒TMP3	B⇒CCR $FF:CCR⇒D	X_L⇒CCR $FF:CCR⇒X	Y_L⇒CCR $FF:CCR⇒Y	SP_L⇒CCR $FF:CCR⇒SP
3		$00:A⇒TMP2 $TMP2_L$⇒A	$00:B⇒TMP2 $TMP2_L$⇒B	$00:CCR⇒TMP2 $TMP2_L$⇒CCR	TMP3⇔TMP2	D⇔TMP2	X⇔TMP2	Y⇔TMP2	SP⇔TMP2
4		$00:A⇒D	$00:B⇒D	$00:CCR⇒D B⇒CCR	TMP3⇔D	D⇔D	X⇔D	Y⇔D	SP⇔D
5		$00:A⇒X X_L⇒A	$00:B⇒X X_L⇒B	$00:CCR⇒X X_L⇒CCR	TMP3⇔X	D⇔X	X⇔X	Y⇔X	SP⇔X
6		$00:A⇒Y Y_L⇒A	$00:B⇒Y Y_L⇒B	$00:CCR⇒Y Y_L⇒CCR	TMP3⇔Y	D⇔Y	X⇔Y	Y⇔Y	SP⇔Y
7		$00:A⇒SP SP_L⇒A	$00:B⇒SP SP_L⇒B	$00:CCR⇒SP SP_L⇒CCR	TMP3⇔SP	D⇔SP	X⇔SP	Y⇔SP	SP⇔SP

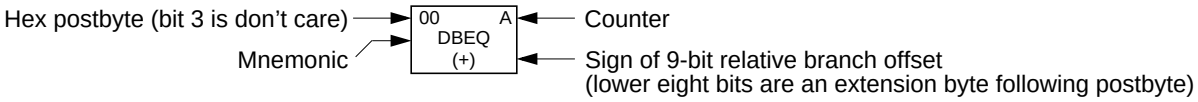
TMP2 and TMP3 registers are for factory use only.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

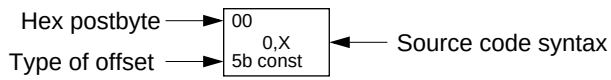
4.7 Loop Primitive Postbyte (lb) Encoding

00 A DBEQ (+)	10 A DBEQ (–)	20 A DBNE (+)	30 A DBNE (–)	40 A TBEQ (+)	50 A TBEQ (–)	60 A TBNE (+)	70 A TBNE (–)	80 A IBEQ (+)	90 A IBEQ (–)	A0 A IBNE (+)	B0 A IBNE (–)
01 B DBEQ (+)	11 B DBEQ (–)	21 B DBNE (+)	31 B DBNE (–)	41 B TBEQ (+)	51 B TBEQ (–)	61 B TBNE (+)	71 B TBNE (–)	81 B IBEQ (+)	91 B IBEQ (–)	A1 B IBNE (+)	B1 B IBNE (–)
02 —	12 —	22 —	32 —	42 —	52 —	62 —	72 —	82 —	92 —	A2 —	B2 —
03 —	13 —	23 —	33 —	43 —	53 —	63 —	73 —	83 —	93 —	A3 —	B3 —
04 D DBEQ (+)	14 D DBEQ (–)	24 D DBNE (+)	34 D DBNE (–)	44 D TBEQ (+)	54 D TBEQ (–)	64 D TBNE (+)	74 D TBNE (–)	84 D IBEQ (+)	94 D IBEQ (–)	A4 D IBNE (+)	B4 D IBNE (–)
05 X DBEQ (+)	15 X DBEQ (–)	25 X DBNE (+)	35 X DBNE (–)	45 X TBEQ (+)	55 X TBEQ (–)	65 X TBNE (+)	75 X TBNE (–)	85 X IBEQ (+)	95 X IBEQ (–)	A5 X IBNE (+)	B5 X IBNE (–)
06 Y DBEQ (+)	16 Y DBEQ (–)	26 Y DBNE (+)	36 Y DBNE (–)	46 Y TBEQ (+)	56 Y TBEQ (–)	66 Y TBNE (+)	76 Y TBNE (–)	86 Y IBEQ (+)	96 Y IBEQ (–)	A6 Y IBNE (+)	B6 Y IBNE (–)
07 SP DBEQ (+)	17 SP DBEQ (–)	27 SP DBNE (+)	37 SP DBNE (–)	47 SP TBEQ (+)	57 SP TBEQ (–)	67 SP TBNE (+)	77 SP TBNE (–)	87 SP IBEQ (+)	97 SP IBEQ (–)	A7 SP IBNE (+)	B7 SP IBNE (–)



4.8 Indexed Addressing Postbyte (xb) Encoding

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



PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 5 Instruction Execution

The CPU uses a three-stage instruction queue to facilitate instruction fetching and increase execution speed. This section provides a general description of the instruction queue during normal program execution and during changes in execution flow. Operation of the queue is automatic and generally transparent to the user.

5.1 Normal Instruction Execution

Queue logic prefetches program information and positions it for sequential execution, one instruction at a time. The relationship between bus cycles and execution cycles is straightforward and facilitates tracking and debugging.

There are three 16-bit stages in the instruction queue. Instructions enter the queue at stage1 and roll out after stage 3. Each byte in the queue is selectable. An opcode-prediction algorithm determines the location of the next opcode in the instruction queue.

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

Two external pins, IPIPE[1:0], provide time-multiplexed information about instruction execution and data movement in the queue. Decoding and using the IPIPE signals is discussed in.

5.2 Execution Sequence

All queue operations are defined by two basic queue movement cycles. Queue movement cycles are only one factor in instruction execution time and should not be confused with bus cycles.

5.2.1 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.

5.2.2 Advance and Load from Data Bus

The content of queue stage 1 advances to stage 2, stage 2 advances to stage 3, and stage 1 is loaded with a word of program information from the data bus.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.3 Changes of Flow

Most of the time, the instruction queue operates in a continuous sequence of queue movement cycles. When program flow changes because of an exception, subroutine call, branch, or jump, the queue automatically adjusts its movement sequence to accommodate the change in program flow.

5.3.1 Exceptions

Exceptions include three types of reset, an unimplemented opcode trap, a software interrupt instruction, X bit maskable interrupts, and I bit maskable interrupts.

To minimize the effect of queue operation on exception handling:

- The exception vector fetch is the first part of exception processing.
- Fetches to refill the queue from the new address are interleaved with the context-stacking operations, so that program access time does not delay the switch.

Please see **Section 6** of this guide for more detailed information on exception processing.

5.3.2 Subroutines

The CPU can branch to (BSR), jump to (JSR), or CALL subroutines. The BSR and JSR instructions are for accessing subroutines in the normal 64K byte address space. The CALL instruction is for accessing subroutines in expanded memory.

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

A subroutine in the normal 64K byte address space ends with a return from subroutine instruction (RTS). RTS unstacks the return address and does three program word fetches from that address to refill the queue.

CALL is similar to JSR. MCUs with expanded memory treat the 16K bytes of addresses from $8000 to $BFFF as an expanded memory window. An 8-bit PPAGE register switches the memory pages in the window. CALL calculates and stacks a return address along with the current PPAGE value and writes a new instruction-supplied value to PPAGE. Then it calculates the subroutine address and fetches three program words from that address to refill the queue.

A subroutine in expanded memory ends with a return from call instruction (RTC). RTC unstacks the PPAGE value and the return address and does three program word fetches from that address to refill the queue.

5.3.3 Branches

A branch instruction changes the execution flow when a specific condition exists. There are short conditional branches, long conditional branches, and bit-condition branches. All branch instructions affect the queue similarly, but there are differences in cycle counts between the various types. Loop primitive instructions are a special type of branch instruction for implementing counter-based loops.

A branch instruction has two execution cases. Either the branch condition is satisfied, and a change of flow takes place, or the condition is not satisfied, and no change of flow occurs.

5.3.3.1 Short Branches

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

The branch-taken case for a short branch requires that the queue be refilled so that execution can begin at a new address. First, the CPU calculates the effective address of the destination using the relative offset in the instruction. Then it loads the address into the program counter, and performs three program word fetches at the new address to refill the queue.

5.3.3.2 Long Branches

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

A long branch instruction begins with a $18 prebyte which indicates that the opcode is on page 2 of the opcode map. The CPU treats the prebyte as a special one-byte instruction. To maintain alignment in the two-byte queue, the first cycle of a long branch instruction is an optional cycle. If the prebyte is not aligned, the CPU does a program word access; if the prebyte is aligned, the first cycle is a free cycle.

Optional cycles align byte-sized and misaligned instructions with aligned word-length instructions. Program information is always fetched as aligned 16-bit words. When an instruction has an odd number of bytes, and the first byte is not aligned with an even byte boundary, the optional cycle makes an additional program word access that maintains queue order. In all other cases, the optional cycle is a free cycle. In the branch-not-taken case, the queue advances so that execution can continue with the next instruction. The CPU does one program fetch and one optional fetch to refill the queue.

In the branch-taken case, the CPU calculates the effective address of the branch using the 16-bit relative offset contained in the second word of the instruction. It loads the address into the program counter and then does three program word fetches at the new address to refill the queue.

5.3.3.3 Bit Condition Branches

A bit-condition branch instruction reads a location in memory and branches if the bits in that location are in a certain state. It can use direct, extended, or indexed addressing mode. Indexed operations require varying amounts of information to determine the effective address, so instruction length varies with the addressing mode. The amount of program information fetched also varies with instruction length. To shorten execution time, the CPU does one program word fetch in anticipation of the branch-taken case. The data from this fetch is ignored if the branch is not taken, and the CPU refills the queue according to the instruction length. If the branch is taken, the CPU refills the queue from the new address according to the instruction length.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.3.3.4 Loop Primitive Instructions

A loop primitive instruction tests a counter value in a register or accumulator. If the test condition is met, the CPU branches to an address specified by a 9-bit relative offset contained in the instruction. There are autoincrement and autodecrement 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, the CPU does one program word fetch in anticipation of the branch-taken case. The data from this fetch is ignored if the branch is not taken, and the CPU does one program fetch and one optional fetch to refill the queue. If the branch is taken, the CPU refills the queue with two additional program word fetches at the new address.

5.3.4 Jumps

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 with the addressing mode. The amount of program information fetched also varies with instruction length. In all forms of JMP, the CPU refills the queue with three program word fetches at the new address.

5.4 Instruction Timing

The **Access Detail** column of the summary in **Table 5-1** shows how many bytes of information the CPU accesses while executing an instruction. With this information and knowledge of the type and speed of memory in the system, you can determine the execution time for any instruction in any system. Simply count the 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.

A description of the notation used in each column of the table is given in the subsections that follow including that of the **Access Detail** column. This information as well as the summary in **Table 5-1** is repeated from **Section 1** of this guide for completeness.

Table 5-1 Instruction Set Summary

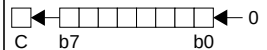
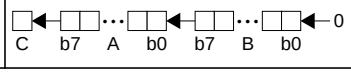
Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ABA	Add B to A; (A)+(B)⇒A	INH	18 06	OO	– – Δ – Δ Δ Δ Δ
ABXSame as LEAX B,X	Add B to X; (X)+(B)⇒X	IDX	1A E5	Pf	– – – – – – – –
ABYSame as LEAY B,Y	Add B to Y; (Y)+(B)⇒Y	IDX	19 ED	Pf	– – – – – – – –
ADCA #*opr8i* ADCA *opr8a* ADCA *opr16a* ADCA *oprx0_xysppc* ADCA *oprx9,xysppc* ADCA *oprx16,xysppc* ADCA [D,*xysppc*] ADCA [*oprx16*,*xysppc*]	Add with carry to A; (A)+(M)+C⇒A or (A)+imm+C⇒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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – Δ – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ADCB #*opr8i* ADCB *opr8a* ADCB *opr16a* ADCB *oprx0_xysppc* ADCB *oprx9,xysppc* ADCB *oprx16,xysppc* ADCB [D,*xysppc*] ADCB [*oprx16,xysppc*]	Add with carry to B; (B)+(M)+C⇒B or (B)+imm+C⇒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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – Δ – Δ Δ Δ Δ

ADDA #*opr8i*
ADDA *opr8a*
ADDA *opr16a*
ADDA *oprx0_xysppc*
ADDA *oprx9,xysppc*
ADDA *oprx16,xysppc*
ADDA [D,*xysppc*]
ADDA [*oprx16,xysppc*]
	Add to A; (A)+(M)⇒A
	or (A)+imm⇒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
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – Δ – Δ Δ Δ Δ

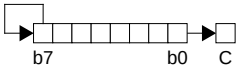
ADDB #*opr8i*
ADDB *opr8a*
ADDB *opr16a*
ADDB *oprx0_xysppc*
ADDB *oprx9,xysppc*
ADDB *oprx16,xysppc*
ADDB [D,*xysppc*]
ADDB [*oprx16,xysppc*]
	Add to B; (B)+(M)⇒B
	or (B)+imm⇒B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CB ii
			DB dd
			FB hh ll
			EB xb
			EB xb ff
			EB xb ee ff
			EB xb
			EB xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – Δ – Δ Δ Δ Δ

ADDD #*opr16i*
ADDD *opr8a*
ADDD *opr16a*
ADDD *oprx0_xysppc*
ADDD *oprx9,xysppc*
ADDD *oprx16,xysppc*
ADDD [D,*xysppc*]
ADDD [*oprx16,xysppc*]
	Add to D; (A:B)+(M:M+1)⇒A:B
	or (A:B)+imm⇒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
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ Δ Δ

ANDA #*opr8i* ANDA *opr8a* ANDA *opr16a* ANDA *oprx0_xysppc* ANDA *oprx9,xysppc* ANDA *oprx16,xysppc* ANDA [D,*xysppc*] ANDA [*oprx16,xysppc*]	AND with A; (A)•(M)⇒A or (A)•imm⇒A	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ANDB #*opr8i* ANDB *opr8a* ANDB *opr16a* ANDB *oprx0_xysppc* ANDB *oprx9,xysppc* ANDB *oprx16,xysppc* ANDB [D,*xysppc*] ANDB [*oprx16,xysppc*]	AND with B; (B)•(M)⇒B or (B)•imm⇒B	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 rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ANDCC #*opr8i*	AND with CCR; (CCR)•imm⇒CCR	IMM	10 ii	P	⇓ ⇓ ⇓ ⇓ ⇓ ⇓ ⇓ ⇓
ASL *opr16a*Same as LSL ASL *oprx0_xysp* ASL *oprx9,xysppc* ASL *oprx16,xysppc* ASL [D,*xysppc*] ASL [*oprx16,xysppc*] ASLASame as LSLA ASLBSame as LSLB	Arithmetic shift left M  Arithmetic shift left A Arithmetic shift left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ Δ Δ
ASLDSame as LSLD	Arithmetic shift left D  	INH	59	0	– – – – Δ Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
ASR *opr16a* ASR *oprx0_xysppc* ASR *oprx9,xysppc* ASR *oprx16,xysppc* ASR [D,*xysppc*] ASR [*oprx16,xysppc*] ASRA ASRB	Arithmetic shift right M  Arithmetic shift right A Arithmetic shift right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	77 hh ll 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
BCC *rel8*Same as BHS	Branch if C clear; if C=0, then (PC)+2+rel⇒PC	REL	24 rr	PPP (branch) P (no branch)	– – – – – – – –
BCLR *opr8a, msk8* BCLR *opr16a, msk8* BCLR *oprx0_xysppc, msk8* BCLR *oprx9,xysppc, msk8* BCLR *oprx16,xysppc, msk8*	Clear bit(s) in M; (M)•mask byte⇒M	DIR EXT IDX IDX1 IDX2	4D dd mm 1D hh ll mm 0D xb mm 0D xb ff mm 0D xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	– – – – Δ Δ 0 –
BCS *rel8*Same as BLO	Branch if C set; if C=1, then (PC)+2+rel⇒PC	REL	25 rr	PPP (branch) P (no branch)	– – – – – – – –
BEQ *rel8*	Branch if equal; if Z=1, then (PC)+2+rel⇒PC	REL	27 rr	PPP (branch) P (no branch)	– – – – – – – –
BGE *rel8*	Branch if ≥ 0, signed; if N⊕V=0, then (PC)+2+rel⇒PC	REL	2C rr	PPP (branch) P (no branch)	– – – – – – – –
BGND	Enter background debug mode	INH	00	VfPPP	– – – – – – – –
BGT *rel8*	Branch if > 0, signed; if Z | (N⊕V)=0, then (PC)+2+rel⇒PC	REL	2E rr	PPP (branch) P (no branch)	– – – – – – – –
BHI *rel8*	Branch if higher, unsigned; if C | Z=0, then (PC)+2+rel⇒PC	REL	22 rr	PPP (branch) P (no branch)	– – – – – – – –
BHS *rel8*Same as BCC	Branch if higher or same, unsigned; if C=0, then (PC)+2+rel⇒PC	REL	24 rr	PPP (branch) P (no branch)	– – – – – – – –
BITA #*opr8i* BITA *opr8a* BITA *opr16a* BITA *oprx0_xysppc* BITA *oprx9,xysppc* BITA *oprx16,xysppc* BITA [D,*xysppc*] BITA [*oprx16,xysppc*]	Bit test A; (A)•(M) or (A)•imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	85 ii 95 dd B5 hh ll A5 xb A5 xb ff A5 xb ee ff A5 xb A5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
BITB #*opr8i* BITB *opr8a* BITB *opr16a* BITB *oprx0_xysppc* BITB *oprx9,xysppc* BITB *oprx16,xysppc* BITB [D,*xysppc*] BITB [*oprx16,xysppc*]	Bit test B; (B)•(M) or (B)•imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C5 ii D5 dd F5 hh ll E5 xb E5 xb ff E5 xb ee ff E5 xb E5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
BLE *rel8*	Branch if ≤ 0, signed; if Z | (N⊕V)=1, then (PC)+2+rel⇒PC	REL	2F rr	PPP (branch) P (no branch)	– – – – – – – –
BLO *rel8*Same as BCS	Branch if lower, unsigned; if C=1, then (PC)+2+rel⇒PC	REL	25 rr	PPP (branch) P (no branch)	– – – – – – – –
BLS *rel8*	Branch if lower or same, unsigned; if C | Z=1, then (PC)+2+rel⇒PC	REL	23 rr	PPP (branch) P (no branch)	– – – – – – – –
BLT *rel8*	Branch if < 0, signed; if N⊕V=1, then (PC)+2+rel⇒PC	REL	2D rr	PPP (branch) P (no branch)	– – – – – – – –
BMI *rel8*	Branch if minus; if N=1, then (PC)+2+rel⇒PC	REL	2B rr	PPP (branch) P (no branch)	– – – – – – – –
BNE *rel8*	Branch if not equal to 0; if Z=0, then (PC)+2+rel⇒PC	REL	26 rr	PPP (branch) P (no branch)	– – – – – – – –
BPL *rel8*	Branch if plus; if N=0, then (PC)+2+rel⇒PC	REL	2A rr	PPP (branch) P (no branch)	– – – – – – – –
BRA *rel8*	Branch always	REL	20 rr	PPP	– – – – – – – –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
BRCLR *oprx0_xysppc*, *msk8*, *rel8* BRCLR *oprx9,xysppc*, *msk8*, *rel8* BRCLR *oprx16,xysppc*, *msk8*, *rel8*	Branch if bit(s) clear; if (M)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4F dd mm rr 1F hh ll mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BRN *rel8*	Branch never	REL	21 rr	P	– – – – – – – –
BRSET *oprx0_xysppc*, *msk8*, *rel8* BRSET *oprx9,xysppc*, *msk8*, *rel8* BRSET *oprx16,xysppc*, *msk8*, *rel8*	Branch if bit(s) set; if ($\overline{M}$)•(mask byte)=0, then (PC)+2+rel⇒PC	DIR EXT IDX IDX1 IDX2	4E dd mm rr 1E hh ll mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	– – – – – – – –
BSET *oprx0_xysppc*, *msk8* BSET *oprx9,xysppc*, *msk8* BSET *oprx16,xysppc*, *msk8*	Set bit(s) in M (M) | mask byte⇒M	DIR EXT IDX IDX1 IDX2	4C dd mm 1C hh ll mm 0C xb mm 0C xb ff mm 0C xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	– – – – Δ Δ 0 –
BSR *rel8*	Branch to subroutine; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (PC)+2+rel⇒PC	REL	07 rr	SPPP	– – – – – – – –
BVC *rel8*	Branch if V clear; if V=0, then (PC)+2+rel⇒PC	REL	28 rr	PPP (branch) P (no branch)	– – – – – – – –
BVS *rel8*	Branch if V set; if V=1, then (PC)+2+rel⇒PC	REL	29 rr	PPP (branch) P (no branch)	– – – – – – – –
CALL *oprx0_xysppc*, *page* CALL *oprx9,xysppc*, *page* CALL *oprx16,xysppc*, *page* CALL [D,*xysppc*] CALL [*oprx16*, *xysppc*]	Call subroutine in expanded memory (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (PPG)⇒M_{SP} pg⇒PPAGE register subroutine address⇒PC	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	4A hh ll pg 4B xb pg 4B xb ff pg 4B xb ee ff pg 4B xb 4B xb ee ff	gnSsPPP gnSsPPP gnSsPPP fgnSsPPP fIignSsPPP fIignSsPPP	– – – – – – – –
CBA	Compare A to B; (A)–(B)	INH	18 17	OO	– – – – Δ Δ Δ Δ
CLCSame as ANDCC #$FE	Clear C bit	IMM	10 FE	P	– – – – – – – 0
CLISame as ANDCC #$EF	Clear I bit	IMM	10 EF	P	– – – 0 – – – –
CLR *oprx0_xysppc* CLR *oprx9,xysppc* CLR *oprx16,xysppc* CLR [D,*xysppc*] CLR [*oprx16,xysppc*] CLRA CLRB	Clear M; $00⇒M Clear A; $00⇒A Clear B; $00⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	79 hh ll 69 xb 69 xb ff 69 xb ee ff 69 xb 69 xb ee ff 87 C7	PwO Pw PwO PwP PIfw PIPw O O	– – – – 0 1 0 0
CLVSame as ANDCC #$FD	Clear V	IMM	10 FD	P	– – – – – – 0 –

CMPA #*oprx0_xysppc*
CMPA *oprx9,xysppc*
CMPA *oprx16,xysppc*
CMPA [D,*xysppc*]
CMPA [*oprx16,xysppc*]
	Compare A
	(A)–(M) or (A)–imm
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			81 ii
			91 dd
			B1 hh ll
			A1 xb
			A1 xb ff
			A1 xb ee ff
			A1 xb
			A1 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

CMPB #*oprx0_xysppc*
CMPB *oprx9,xysppc*
CMPB *oprx16,xysppc*
CMPB [D,*xysppc*]
CMPB [*oprx16,xysppc*]
	Compare B
	(B)–(M) or (B)–imm
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C1 ii
			D1 dd
			F1 hh ll
			E1 xb
			E1 xb ff
			E1 xb ee ff
			E1 xb
			E1 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
COM *opr16a* COM *oprx0_xysppc* COM *oprx9,xysppc* COM *oprx16,xysppc* COM [D,*xysppc*] COM [*oprx16,xysppc*] COMA COMB	Complement M; ($\overline{M}$)=$FF–(M)⇒M Complement A; ($\overline{A}$)=$FF–(A)⇒A Complement B; ($\overline{B}$)=$FF–(B)⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	71 hh ll 61 xb 61 xb ff 61 xb ee ff 61 xb 61 xb ee ff 41 51	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ 0 1
CPD #*opr16i* CPD *opr8a* CPD *opr16a* CPD *oprx0_xysppc* CPD *oprx9,xysppc* CPD *oprx16,xysppc* CPD [D,*xysppc*] CPD [*oprx16,xysppc*]	Compare D (A:B)–(M:M+1) or (A:B)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8C jj kk 9C dd BC hh ll AC xb AC xb ff AC xb ee ff AC xb AC xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPS #*opr16i* CPS *opr8a* CPS *opr16a* CPS *oprx0_xysppc* CPS *oprx9,xysppc* CPS *oprx16,xysppc* CPS [D,*xysppc*] CPS [*oprx16,xysppc*]	Compare SP (SP)–(M:M+1) or (SP)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8F jj kk 9F dd BF hh ll AF xb AF xb ff AF xb ee ff AF xb AF xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPX #*opr16i* CPX *opr8a* CPX *opr16a* CPX *oprx0_xysppc* CPX *oprx9,xysppc* CPX *oprx16,xysppc* CPX [D,*xysppc*] CPX [*oprx16,xysppc*]	Compare X (X)–(M:M+1) or (X)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8E jj kk 9E dd BE hh ll AE xb AE xb ff AE xb ee ff AE xb AE xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
CPY #*opr16i* CPY *opr8a* CPY *opr16a* CPY *oprx0_xysppc* CPY *oprx9,xysppc* CPY *oprx16,xysppc* CPY [D,*xysppc*] CPY [*oprx16,xysppc*]	Compare Y (Y)–(M:M+1) or (Y)–imm	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8D jj kk 9D dd BD hh ll AD xb AD xb ff AD xb ee ff AD xb AD xb ee ff	P0 RPf RP0 RPf RP0 fRPP fIfRPf fIPRPf	– – – – Δ Δ Δ Δ
DAA	Decimal adjust A for BCD	INH	18 07	0f0	– – – – Δ Δ ? Δ
DBEQ *abdxysp*, *rel9*	Decrement and branch if equal to 0 (counter)–1⇒counter if (counter)=0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PP0 (no branch)	– – – – – – – –
DBNE *abdxysp*, *rel9*	Decrement and branch if not equal to 0; (counter)–1⇒counter; if (counter)≠0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PP0 (no branch)	– – – – – – – –
DEC *opr16a* DEC *oprx0_xysppc* DEC *oprx9,xysppc* DEC *oprx16,xysppc* DEC [D,*xysppc*] DEC [*oprx16,xysppc*] DECA DECB	Decrement M; (M)–1⇒M Decrement A; (A)–1⇒A Decrement B; (B)–1⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	73 hh ll 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff 43 53	rPw0 rPw rPw0 frPwP fIfrPw fIPrPw 0 0	– – – – Δ Δ Δ –
DESSame as LEAS –1,SP	Decrement SP; (SP)–1⇒SP	IDX	1B 9F	Pf	– – – – – – – –
DEX	Decrement X; (X)–1⇒X	INH	09	0	– – – – – Δ – –
DEY	Decrement Y; (Y)–1⇒Y	INH	03	0	– – – – – Δ – –
EDIV	Extended divide, unsigned; 32 by 16 to 16-bit; (Y:D)÷(X)⇒Y; remainder⇒D	INH	11	fffffffff0	– – – – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
EDIVS	Extended divide,signed; 32 by 16 to 16-bit; (Y:D)÷(X)⇒Y remainder⇒D	INH	18 14	Offfffffffff0	– – – – Δ Δ Δ Δ
EMACS *opr16a*	Extended multiply and accumulate, signed; $(M_X:M_{X+1})\times(M_Y:M_{Y+1})+$ (M~M+3)⇒M~M+3; 16 by 16 to 32-bit	Special	18 12 hh ll	ORROfffRRfWWP	– – – – Δ Δ Δ Δ
EMAXD *oprx0_xysppc* EMAXD *oprx9,xysppc* EMAXD *oprx16,xysppc* EMAXD [D,*xysppc*] EMAXD [*oprx16,xysppc*]	Extended maximum in D; put larger of 2 unsigned 16-bit values in D MAX[(D), (M:M+1)]⇒D N, Z, V, C bits reflect result of internal compare [(D)–(M:M+1)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1A xb 18 1A xb ff 18 1A xb ee ff 18 1A xb 18 1A xb ee ff	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – – Δ Δ Δ Δ
EMAXM *oprx0_xysppc* EMAXM *oprx9,xysppc* EMAXM *oprx16,xysppc* EMAXM [D,*xysppc*] EMAXM [*oprx16,xysppc*]	Extended maximum in M; put larger of 2 unsigned 16-bit values in M MAX[(D), (M:M+1)]⇒M:M+1 N, Z, V, C 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_xysppc* EMIND *oprx9,xysppc* EMIND *oprx16,xysppc* EMIND [D,*xysppc*] EMIND [*oprx16,xysppc*]	Extended minimum in D; put smaller of 2 unsigned 16-bit values in D MIN[(D), (M:M+1)]⇒D N, Z, V, C bits reflect result of internal compare [(D)–(M:M+1)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1B xb 18 1B xb ff 18 1B xb ee ff 18 1B xb 18 1B xb ee ff	ORPf ORPO OfRPP OfIfRPf OfIPRPf	– – – – Δ Δ Δ Δ
EMINM *oprx0_xysppc* EMINM *oprx9,xysppc* EMINM *oprx16,xysppc* EMINM [D,*xysppc*] EMINM [*oprx16,xysppc*]	Extended minimum in M; put smaller of 2 unsigned 16-bit values in M MIN[(D), (M:M+1)]⇒M:M+1 N, Z, V, C 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	Extended multiply, unsigned (D)×(Y)⇒Y:D; 16 by 16 to 32-bit	INH	13	ff0	– – – – Δ Δ – Δ
EMULS	Extended multiply, signed (D)×(Y)⇒Y:D; 16 by 16 to 32-bit	INH	18 13	Of0 Off0 (if followed by page 2 instruction)	– – – – Δ Δ – Δ
EORA #*opr8i* EORA *opr8a* EORA *opr16a* EORA *oprx0_xysppc* EORA *oprx9,xysppc* EORA *oprx16,xysppc* EORA [D,*xysppc*] EORA [*oprx16,xysppc*]	Exclusive OR A (A)⊕(M)⇒A or (A)⊕imm⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	88 ii 98 dd B8 hh ll A8 xb A8 xb ff A8 xb ee ff A8 xb A8 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
EORB #*opr8i* EORB *opr8a* EORB *opr16a* EORB *oprx0_xysppc* EORB *oprx9,xysppc* EORB *oprx16,xysppc* EORB [D,*xysppc*] EORB [*oprx16,xysppc*]	Exclusive OR B (B)⊕(M)⇒B or (B)⊕imm⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C8 ii D8 dd F8 hh ll E8 xb E8 xb ff E8 xb ee ff E8 xb E8 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ETBL *oprx0_xysppc*	Extended table lookup and interpolate, 16-bit; (M:M+1)+ [(B)×((M+2:M+3)–(M:M+1))]⇒D	IDX	18 3F xb	ORRffffffP	– – – – Δ Δ – Δ
Before executing ETBL, initialize B with fractional part of lookup value; initialize index register to point to first table entry (M:M+1). No extensions or indirect addressing allowed.					
EXG *abcdxysp,abcdxysp*	Exchange register contents (r1)⇔(r2) r1 and r2 same size $00:(r1)⇒r2r1=8-bit; r2=16-bit $(r1_L)⇔(r2)r1=16-bit; r2=8-bit	INH	B7 eb	P	– – – – – – – –
FDIV	Fractional divide; (D)÷(X)⇒X remainder⇒D; 16 by 16-bit	INH	18 11	Offffffffff0	– – – – Δ Δ Δ

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
IBEQ *abdxysp*, *rel9*	Increment and branch if equal to 0 (counter)+1⇒counter If (counter)=0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
IBNE *abdxysp*, *rel9*	Increment and branch if not equal to 0 (counter)+1⇒counter If (counter)≠0, then branch	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
IDIV	Integer divide, unsigned; (D)÷(X)⇒X Remainder⇒D; 16 by 16-bit	INH	18 10	OfffffffffO	– – – – – Δ 0 Δ
IDIVS	Integer divide, signed; (D)÷(X)⇒X Remainder⇒D; 16 by 16-bit	INH	18 15	OfffffffffO	– – – – Δ Δ Δ Δ
INC *opr16a* INC *oprx0_xysppc* INC *oprx9,xysppc* INC *oprx16,xysppc* INC [D,*xysppc*] INC [*oprx16,xysppc*] INCA INCB	Increment M; (M)+1⇒M Increment A; (A)+1⇒A Increment B; (B)+1⇒B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	72 hh ll 62 xb 62 xb ff 62 xb ee ff 62 xb 62 xb ee ff 42 52	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ –
INSSame as LEAS 1,SP	Increment SP; (SP)+1⇒SP	IDX	1B 81	Pf	– – – – – – – –
INX	Increment X; (X)+1⇒X	INH	08	O	– – – – – Δ – –
INY	Increment Y; (Y)+1⇒Y	INH	02	O	– – – – – Δ – –
JMP *opr16a* JMP *oprx0_xysppc* JMP *oprx9,xysppc* JMP *oprx16,xysppc* JMP [D,*xysppc*] JMP [*oprx16,xysppc*]	Jump Subroutine address⇒PC	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	06 hh ll 05 xb 05 xb ff 05 xb ee ff 05 xb 05 xb ee ff	PPP PPP PPP fPPP fIfPPP fIfPPP	– – – – – – – –
JSR *opr8a* JSR *opr16a* JSR *oprx0_xysppc* JSR *oprx9,xysppc* JSR *oprx16,xysppc* JSR [D,*xysppc*] JSR [*oprx16,xysppc*]	Jump to subroutine (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} Subroutine address⇒PC	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	17 dd 16 hh ll 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff	SPPP SPPP PPPS PPPS fPPPS fIfPPPS fIfPPPS	– – – – – – – –
LBCC *rel16*Same as LBHS	Long branch if C clear; if C=0, then (PC)+4+rel⇒PC	REL	18 24 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBCS *rel16*Same as LBLO	Long branch if C set; if C=1, then (PC)+4+rel⇒PC	REL	18 25 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBEQ *rel16*	Long branch if equal; if Z=1, then (PC)+4+rel⇒PC	REL	18 27 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBGE *rel16*	Long branch if ≥ 0, signed If N⊕V=0, then (PC)+4+rel⇒PC	REL	18 2C qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBGT *rel16*	Long branch if > 0, signed If Z | (N⊕V)=0, then (PC)+4+rel⇒PC	REL	18 2E qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBHI *rel16*	Long branch if higher, unsigned If C | Z=0, then (PC)+4+rel⇒PC	REL	18 22 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBHS *rel16*Same as LBCC	Long branch if higher or same, unsigned; If C=0, (PC)+4+rel⇒PC	REL	18 24 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLE *rel16*	Long branch if ≤ 0, signed; if Z | (N⊕V)=1, then (PC)+4+rel⇒PC	REL	18 2F qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLO *rel16*Same as LBCS	Long branch if lower, unsigned; if C=1, then (PC)+4+rel⇒PC	REL	18 25 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLS *rel16*	Long branch if lower or same, unsigned; If C | Z=1, then (PC)+4+rel⇒PC	REL	18 23 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBLT *rel16*	Long branch if < 0, signed If N⊕V=1, then (PC)+4+rel⇒PC	REL	18 2D qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
LBMI *rel16*	Long branch if minus If N=1, then (PC)+4+rel⇒PC	REL	18 2B qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBNE *rel16*	Long branch if not equal to 0 If Z=0, then (PC)+4+rel⇒PC	REL	18 26 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBPL *rel16*	Long branch if plus If N=0, then (PC)+4+rel⇒PC	REL	18 2A qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBRA *rel16*	Long branch always	REL	18 20 qq rr	OPPP	– – – – – – – –
LBRN *rel16*	Long branch never	REL	18 21 qq rr	OPO	– – – – – – – –
LBVC *rel16*	Long branch if V clear If V=0,then (PC)+4+rel⇒PC	REL	18 28 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –
LBVS *rel16*	Long branch if V set If V=1,then (PC)+4+rel⇒PC	REL	18 29 qq rr	OPPP (branch) OPO (no branch)	– – – – – – – –

LDAA #*opr8i*
LDAA *opr8a*
LDAA *opr16a*
LDAA *oprx0_xysppc*
LDAA *oprx9,xysppc*
LDAA *oprx16,xysppc*
LDAA [D,*xysppc*]
LDAA [*oprx16,xysppc*]
	Load A
	(M)⇒A
	or imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			86 ii
			96 dd
			B6 hh ll
			A6 xb
			A6 xb ff
			A6 xb ee ff
			A6 xb
			A6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ 0 –

LDAB #*opr8i*
LDAB *opr8a*
LDAB *opr16a*
LDAB *oprx0_xysppc*
LDAB *oprx9,xysppc*
LDAB *oprx16,xysppc*
LDAB [D,*xysppc*]
LDAB [*oprx16,xysppc*]
	Load B
	(M)⇒B
	or imm⇒B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C6 ii
			D6 dd
			F6 hh ll
			E6 xb
			E6 xb ff
			E6 xb ee ff
			E6 xb
			E6 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ 0 –

LDD #*opr16i*
LDD *opr8a*
LDD *opr16a*
LDD *oprx0_xysppc*
LDD *oprx9,xysppc*
LDD *oprx16,xysppc*
LDD [D,*xysppc*]
LDD [*oprx16,xysppc*]
	Load D
	(M:M+1)⇒A:B
	or imm⇒A:B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CC jj kk
			DC dd
			FC hh ll
			EC xb
			EC xb ff
			EC xb ee ff
			EC xb
			EC xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

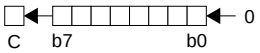
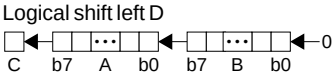
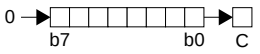
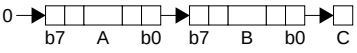
LDS #*opr16i*
LDS *opr8a*
LDS *opr16a*
LDS *oprx0_xysppc*
LDS *oprx9,xysppc*
LDS *oprx16,xysppc*
LDS [D,*xysppc*]
LDS [*oprx16,xysppc*]
	Load SP
	(M:M+1)⇒SP
	or imm⇒SP
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CF jj kk
			DF dd
			FF hh ll
			EF xb
			EF xb ff
			EF xb ee ff
			EF xb
			EF xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

LDX #*opr16i*
LDX *opr8a*
LDX *opr16a*
LDX *oprx0_xysppc*
LDX *oprx9,xysppc*
LDX *oprx16,xysppc*
LDX [D,*xysppc*]
LDX [*oprx16,xysppc*]
	Load X
	(M:M+1)⇒X
	or imm⇒X
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CE jj kk
			DE dd
			FE hh ll
			EE xb
			EE xb ff
			EE xb ee ff
			EE xb
			EE xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

LDY #*opr16i*
LDY *opr8a*
LDY *opr16a*
LDY *oprx0_xysppc*
LDY *oprx9,xysppc*
LDY *oprx16,xysppc*
LDY [D,*xysppc*]
LDY [*oprx16,xysppc*]
	Load Y
	(M:M+1)⇒Y
	or imm⇒Y
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			CD jj kk
			DD dd
			FD hh ll
			ED xb
			ED xb ff
			ED xb ee ff
			ED xb
			ED xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ 0 –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
LEAS *oprx0_xysppc* LEAS *oprx9,xysppc* LEAS *oprx16,xysppc*	Load effective address into SP EA⇒SP	IDX IDX1 IDX2	1B xb 1B xb ff 1B xb ee ff	Pf PO PP	– – – – – – – –
LEAX *oprx0_xysppc* LEAX *oprx9,xysppc* LEAX *oprx16,xysppc*	Load effective address into X EA⇒X	IDX IDX1 IDX2	1A xb 1A xb ff 1A xb ee ff	Pf PO PP	– – – – – – – –
LEAY *oprx0_xysppc* LEAY *oprx9,xysppc* LEAY *oprx16,xysppc*	Load effective address into Y EA⇒Y	IDX IDX1 IDX2	19 xb 19 xb ff 19 xb ee ff	Pf PO PP	– – – – – – – –
LSL *opr16a*Same as ASL LSL *oprx0_xysppc* LSL *oprx9,xysppc* LSL *oprx16,xysppc* LSL [D,*xysppc*] LSL [*oprx16,xysppc*] LSLASame as ASLA LSLBSame as ASLB	Logical shift left M  Logical shift left A Logical shift left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rOPw rPw rPOw frPPw fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
LSLDSame as ASLD	Logical shift left D  	INH	59	O	– – – – Δ Δ Δ Δ
LSR *oprx16a* LSR *oprx0_xysppc* LSR *oprx9,xysppc* LSR *oprx16,xysppc* LSR [D,*xysppc*] LSR [*oprx16,xysppc*] LSRA LSRB	Logical shift right M  Logical shift right A Logical shift right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	74 hh ll 64 xb 64 xb ff 64 xb ee ff 64 xb 64 xb ee ff 44 54	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – 0 Δ Δ Δ
LSRD	Logical shift right D  	INH	49	O	– – – – 0 Δ Δ Δ
MAXA *oprx0_xysppc* MAXA *oprx9,xysppc* MAXA *oprx16,xysppc* MAXA [D,*xysppc*] MAXA [*oprx16,xysppc*]	Maximum in A; put larger of 2 unsigned 8-bit values in A MAX[(A), (M)]⇒A N, Z, V, C 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	OrPf OrPO OfrPP OfIfrPf OfIPrPf	– – – – Δ Δ Δ Δ
MAXM *oprx0_xysppc* MAXM *oprx9,xysppc* MAXM *oprx16,xysppc* MAXM [D,*xysppc*] MAXM [*oprx16,xysppc*]	Maximum in M; put larger of 2 unsigned 8-bit values in M MAX[(A), (M)]⇒M N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1C xb 18 1C xb ff 18 1C xb ee ff 18 1C xb 18 1C xb ee ff	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	– – – – Δ Δ Δ Δ
MEM	Determine grade of membership; μ (grade)⇒M_Y; (X)+4⇒X; (Y)+1⇒Y If (A)<P1 or (A)>P2, then μ=0; else μ= MIN[((A)–P1)×S1, (P2–(A))×S2, $FF] (A)=current crisp input value; X points at 4 data bytes (P1, P2, S1, S2) of a trapezoidal membership function; Y points at fuzzy input (RAM location)	Special	01	RRfOw	– – ? – ? ? ? ?
MINA *oprx0_xysppc* MINA *oprx9,xysppc* MINA *oprx16,xysppc* MINA [D,*xysppc*] MINA [*oprx16,xysppc*]	Minimum in A; put smaller of 2 unsigned 8-bit values in A MIN[(A), (M)]⇒A N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 19 xb 18 19 xb ff 18 19 xb ee ff 18 19 xb 18 19 xb ee ff	OrPf OrPO OfrPP OfIfrPf OfIPrPf	– – – – Δ Δ Δ Δ
MINM *oprx0_xysppc* MINM *oprx9,xysppc* MINM *oprx16,xysppc* MINM [D,*xysppc*] MINM [*oprx16,xysppc*]	Minimum in N; put smaller of two unsigned 8-bit values in M MIN[(A), (M)]⇒M N, Z, V, C bits reflect result of internal compare [(A)–(M)]	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1D xb 18 1D xb ff 18 1D xb ee ff 18 1D xb 18 1D xb ee ff	OrPw OrPwO OfrPwP OfIfrPw OfIPrPw	– – – – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C

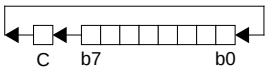
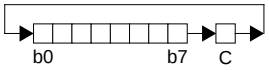
MOVB #*opr8*, *opr16a*
MOVB #*opr8i*, *oprx0_xysppc*
MOVB *opr16a*, *opr16a*
MOVB *oprx0_xysppc*, *oprx0_xysppc*
MOVB *oprx0_xysppc*, *opr16a*
MOVB *oprx0_xysppc*, *oprx0_xysppc*
	Move byte
	Memory-to-memory 8-bit byte-move
	$(M_1)\Rightarrow M_2$
	First operand specifies byte to move
		IMM-EXT
		IMM-IDX
		EXT-EXT
		EXT-IDX
		IDX-EXT
		IDX-IDX
			18 0B ii hh ll
			18 08 xb ii
			18 0C hh ll hh ll
			18 09 xb hh ll
			18 0D xb hh ll
			18 0A xb xb
				OPwP
				OPwO
				OrPwPO
				OPrPw
				OrPwP
				OrPwO
					– – – – – – – –

MOVW #*oprx16*, *opr16a*
MOVW #*opr16i*, *oprx0_xysppc*
MOVW *opr16a*, *opr16a*
MOVW *opr16a*, *oprx0_xysppc*
MOVW *oprx0_xysppc*, *opr16a*
MOVW *oprx0_xysppc*, *oprx0_xysppc*
	Move word
	Memory-to-memory 16-bit word-move
	$(M_1{:}M_1{+}1)\Rightarrow M_2{:}M_2{+}1$
	First operand specifies word to move
		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	Multiply, unsigned (A)×(B)⇒A:B; 8 by 8-bit	INH	12	0	– – – – – – – Δ
NEG *opr16a* NEG *oprx0_xysppc* NEG *oprx9*,*xysppc* NEG *oprx16*,*xysppc* NEG [D,*xysppc*] NEG [*oprx16*,*xysppc*] NEGA NEGB	Negate M; 0–(M)⇒M or $(\overline{M})+1\Rightarrow M$ Negate A; 0–(A)⇒A or $(\overline{A})+1\Rightarrow A$ Negate B; 0–(B)⇒B or $(\overline{B})+1\Rightarrow B$	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	70 hh ll 60 xb 60 xb ff 60 xb ee ff 60 xb 60 xb ee ff 40 50	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
NOP	No operation	INH	A7	0	– – – – – – – –
ORAA #*opr8i* ORAA *opr8a* ORAA *opr16a* ORAA *oprx0_xysppc* ORAA *oprx9*,*xysppc* ORAA *oprx16*,*xysppc* ORAA [D,*xysppc*] ORAA [*oprx16*,*xysppc*]	OR accumulator A (A) | (M)⇒A or (A) | imm⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8A ii 9A dd BA hh ll AA xb AA xb ff AA xb ee ff AA xb AA xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ORAB #*opr8i* ORAB *opr8a* ORAB *opr16a* ORAB *oprx0_xysppc* ORAB *oprx9*,*xysppc* ORAB *oprx16*,*xysppc* ORAB [D,*xysppc*] ORAB [*oprx16*,*xysppc*]	OR accumulator B (B) | (M)⇒B or (B) | imm⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CA ii DA dd FA hh ll EA xb EA xb ff EA xb ee ff EA xb EA xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ 0 –
ORCC #*opr8i*	OR CCR; (CCR) | imm⇒CCR	IMM	14 ii	P	⇑ – ⇑ ⇑ ⇑ ⇑ ⇑ ⇑
PSHA	Push A; (SP)–1⇒SP; (A)⇒M_{SP}	INH	36	Os	– – – – – – – –
PSHB	Push B; (SP)–1⇒SP; (B)⇒M_{SP}	INH	37	Os	– – – – – – – –
PSHC	Push CCR; (SP)–1⇒SP; (CCR)⇒M_{SP}	INH	39	Os	– – – – – – – –
PSHD	Push D (SP)–2⇒SP; (A:B)⇒$M_{SP}{:}M_{SP+1}$	INH	3B	0S	– – – – – – – –
PSHX	Push X (SP)–2⇒SP; $(X_H{:}X_L)\Rightarrow M_{SP}{:}M_{SP+1}$	INH	34	0S	– – – – – – – –
PSHY	Push Y (SP)–2⇒SP; $(Y_H{:}Y_L)\Rightarrow M_{SP}{:}M_{SP+1}$	INH	35	0S	– – – – – – – –
PULA	Pull A $(M_{SP})\Rightarrow$A; (SP)+1⇒SP	INH	32	uf0	– – – – – – – –
PULB	Pull B $(M_{SP})\Rightarrow$B; (SP)+1⇒SP	INH	33	uf0	– – – – – – – –
PULC	Pull CCR $(M_{SP})\Rightarrow$CCR; (SP)+1⇒SP	INH	38	uf0	Δ ⇓ Δ Δ Δ Δ Δ
PULD	Pull D $(M_{SP}{:}M_{SP+1})\Rightarrow$A:B; (SP)+2⇒SP	INH	3A	Uf0	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
PULX	Pull X $(M_{SP}:M_{SP+1}) \Rightarrow X_H:X_L$; (SP)+2⇒SP	INH	30	UfO	– – – – – – – –
PULY	Pull Y $(M_{SP}:M_{SP+1}) \Rightarrow Y_H:Y_L$; (SP)+2⇒SP	INH	31	UfO	– – – – – – – –
REV	Rule evaluation, unweighted; find smallest rule input; store to rule outputs unless fuzzy output is larger	Special	18 3A	Orf(t^tx)O* ff+Orft^**	– – ? – ? ? Δ ?
*The t^tx loop is executed once for each element in the rule list. The ^ denotes a check for pending interrupt requests. **These are additional cycles caused by an interrupt: ff is the exit sequence and Orft^ is the re-entry sequence.					
REVW	Rule evaluation, weighted; rule weights optional; find smallest rule input; store to rule outputs unless fuzzy output is larger	Special	18 3B	ORf(t^Tx)O* or ORf(r^ffRf)O** ffff+ORft^*** ffff+ORfr^****	– – ? – ? ? Δ !
*With weighting not enabled, the t^Tx loop is executed once for each element in the rule list. The ^ denotes a check for pending interrupt requests. **With weighting enabled, the t^Tx loop is replaced by r^ffRf. ***Additional cycles caused by an interrupt when weighting is not enabled: ffff is the exit sequence and ORft^ is the re-entry sequence. **** Additional cycles caused by an interrupt when weighting is enabled: ffff is the exit sequence and ORfr^ is the re-entry sequence.					
ROL *opr16a* ROL *oprx0_xysppc* ROL *oprx9,xysppc* ROL *oprx16,xysppc* ROL [D,*xysppc*] ROL [*oprx16,xysppc*] ROLA ROLB	Rotate left M  Rotate left A Rotate left B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	75 hh ll 65 xb 65 xb ff 65 xb ee ff 65 xb 65 xb ee ff 45 55	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
ROR *opr16a* ROR *oprx0_xysppc* ROR *oprx9,xysppc* ROR *oprx16,xysppc* ROR [D,*xysppc*] ROR [*oprx16,xysppc*] RORA RORB	Rotate right M  Rotate right A Rotate right B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	76 hh ll 66 xb 66 xb ff 66 xb ee ff 66 xb 66 xb ee ff 46 56	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	– – – – Δ Δ Δ Δ
RTC	Return from call; $(M_{SP}) \Rightarrow$ PPAGE (SP)+1⇒SP; $(M_{SP}:M_{SP+1}) \Rightarrow PC_H:PC_L$ (SP)+2⇒SP	INH	0A	uUnfPPP	– – – – – – – –
RTI	Return from interrupt $(M_{SP}) \Rightarrow$ CCR; (SP)+1⇒SP $(M_{SP}:M_{SP+1}) \Rightarrow$ B:A;(SP)+2⇒SP $(M_{SP}:M_{SP+1}) \Rightarrow X_H:X_L$;(SP)+4⇒SP $(M_{SP}:M_{SP+1}) \Rightarrow PC_H:PC_L$;(SP)+2⇒SP $(M_{SP}:M_{SP+1}) \Rightarrow Y_H:Y_L$;(SP)+4⇒SP	INH	0B	uUUUUPPP or uUUUUfVfPPP*	Δ ⇓ Δ Δ Δ Δ Δ Δ
*RTI takes 11 cycles if an interrupt is pending.					
RTS	Return from subroutine $(M_{SP}:M_{SP+1}) \Rightarrow PC_H:PC_L$; (SP)+2⇒SP	INH	3D	UfPPP	– – – – – – – –
SBA	Subtract B from A; (A)–(B)⇒A	INH	18 16	OO	– – – – Δ Δ Δ Δ
SBCA #*opr8i* SBCA *opr8a* SBCA *opr16a* SBCA *oprx0_xysppc* SBCA *oprx9,xysppc* SBCA *oprx16,xysppc* SBCA [D,*xysppc*] SBCA [*oprx16,xysppc*]	Subtract with carry from A (A)–(M)–C⇒A or (A)–imm–C⇒A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	82 ii 92 dd B2 hh ll A2 xb A2 xb ff A2 xb ee ff A2 xb A2 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
SBCB #*opr8i* SBCB *opr8a* SBCB *opr16a* SBCB *oprx0_xysppc* SBCB *oprx9,xysppc* SBCB *oprx16,xysppc* SBCB [D,*xysppc*] SBCB [*oprx16,xysppc*]	Subtract with carry from B (B)–(M)–C⇒B or (B)–imm–C⇒B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C2 ii D2 dd F2 hh ll E2 xb E2 xb ff E2 xb ee ff E2 xb E2 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	– – – – Δ Δ Δ Δ
SECSame as ORCC #$01	Set C bit	IMM	14 01	P	– – – – – – – 1
SEISame as ORCC #$10	Set I bit	IMM	14 10	P	– – – 1 – – – –
SEVSame as ORCC #$02	Set V bit	IMM	14 02	P	– – – – – – 1 –
SEX *abc,dxysp*Same as TFR r1, r2	Sign extend; 8-bit r1 to 16-bit r2 $00:(r1)⇒r2 if bit 7 of r1 is 0 $FF:(r1)⇒r2 if bit 7 of r1 is 1	INH	B7 eb	P	– – – – – – – –
STAA *opr8a* STAA *opr16a* STAA *oprx0_xysppc* STAA *oprx9,xysppc* STAA *oprx16,xysppc* STAA [D,*xysppc*] STAA [*oprx16,xysppc*]	Store accumulator A (A)⇒M	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5A dd 7A hh ll 6A xb 6A xb ff 6A xb ee ff 6A xb 6A xb ee ff	Pw PwO Pw PwO PwP PIfw PIPw	– – – – Δ Δ 0 –
STAB *opr8a* STAB *opr16a* STAB *oprx0_xysppc* STAB *oprx9,xysppc* STAB *oprx16,xysppc* STAB [D,*xysppc*] STAB [*oprx16,xysppc*]	Store accumulator B (B)⇒M	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5B dd 7B hh ll 6B xb 6B xb ff 6B xb ee ff 6B xb 6B xb ee ff	Pw PwO Pw PwO PwP PIfw PIPw	– – – – Δ Δ 0 –
STD *opr8a* STD *opr16a* STD *oprx0_xysppc* STD *oprx9,xysppc* STD *oprx16,xysppc* STD [D,*xysppc*] STD [*oprx16,xysppc*]	Store D (A:B)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5C dd 7C hh ll 6C xb 6C xb ff 6C xb ee ff 6C xb 6C xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –
STOP	Stop processing; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP} Stop all clocks	INH	18 3E	OOSSSSsf (enter stop mode) fVfPPP (exit stop mode) ff (continue stop mode) 00 (if stop mode disabled by S=1)	– – – – – – – –
STS *opr8a* STS *opr16a* STS *oprx0_xysppc* STS *oprx9,xysppc* STS *oprx16,xysppc* STS [D,*xysppc*] STS [*oprx16,xysppc*]	Store SP (SP_H:SP_L)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5F dd 7F hh ll 6F xb 6F xb ff 6F xb ee ff 6F xb 6F xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –
STX *opr8a* STX *opr16a* STX *oprx0_xysppc* STX *oprx9,xysppc* STX *oprx16,xysppc* STX [D,*xysppc*] STX [*oprx16,xysppc*]	Store X (X_H:X_L)⇒M:M+1	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5E dd 7E hh ll 6E xb 6E xb ff 6E xb ee ff 6E xb 6E xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
STY *opr8a* STY *opr16a* STY *oprx0_xysppc* STY *oprx9,xysppc* STY *oprx16,xysppc* STY [D,*xysppc*] STY [*oprx16,xysppc*]	Store Y $(Y_H:Y_L) \Rightarrow M:M+1$	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5D dd 7D hh ll 6D xb 6D xb ff 6D xb ee ff 6D xb 6D xb ee ff	PW PWO PW PWO PWP PIfW PIPW	– – – – Δ Δ 0 –

SUBA #*opr8i*
SUBA *opr8a*
SUBA *opr16a*
SUBA *oprx0_xysppc*
SUBA *oprx9,xysppc*
SUBA *oprx16,xysppc*
SUBA [D,*xysppc*]
SUBA [*oprx16,xysppc*]
	Subtract from A
	(A)–(M)⇒A
	or (A)–imm⇒A
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			80 ii
			90 dd
			B0 hh ll
			A0 xb
			A0 xb ff
			A0 xb ee ff
			A0 xb
			A0 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

SUBB #*opr8i*
SUBB *opr8a*
SUBB *opr16a*
SUBB *oprx0_xysppc*
SUBB *oprx9,xysppc*
SUBB *oprx16,xysppc*
SUBB [D,*xysppc*]
SUBB [*oprx16,xysppc*]
	Subtract from B
	(B)–(M)⇒B
	or (B)–imm⇒B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			C0 ii
			D0 dd
			F0 hh ll
			E0 xb
			E0 xb ff
			E0 xb ee ff
			E0 xb
			E0 xb ee ff
				P
				rPf
				rPO
				rPf
				rPO
				frPP
				fIfrPf
				fIPrPf
					– – – – Δ Δ Δ Δ

SUBD #*opr16i*
SUBD *opr8a*
SUBD *opr16a*
SUBD *oprx0_xysppc*
SUBD *oprx9,xysppc*
SUBD *oprx16,xysppc*
SUBD [D,*xysppc*]
SUBD [*oprx16,xysppc*]
	Subtract from D
	(A:B)–(M:M+1)⇒A:B
	or (A:B)–imm⇒A:B
		IMM
		DIR
		EXT
		IDX
		IDX1
		IDX2
		[D,IDX]
		[IDX2]
			83 jj kk
			93 dd
			B3 hh ll
			A3 xb
			A3 xb ff
			A3 xb ee ff
			A3 xb
			A3 xb ee ff
				PO
				RPf
				RPO
				RPf
				RPO
				fRPP
				fIfRPf
				fIPRPf
					– – – – Δ Δ Δ Δ

SWI	Software interrupt; (SP)–2⇒SP $RTN_H:RTN_L \Rightarrow M_{SP}:M_{SP+1}$ (SP)–2⇒SP; $(Y_H:Y_L) \Rightarrow M_{SP}:M_{SP+1}$ (SP)–2⇒SP; $(X_H:X_L) \Rightarrow M_{SP}:M_{SP+1}$ (SP)–2⇒SP; (B:A)⇒$M_{SP}:M_{SP+1}$ (SP)–1⇒SP; (CCR)⇒M_{SP};1⇒I (SWI vector)⇒PC	INH	3F	VSPSSPSsP*	– – – 1 – – – –
*The CPU also uses VSPSSPSsP for hardware interrupts and unimplemented opcode traps.					
TAB	Transfer A to B; (A)⇒B	INH	18 0E	OO	– – – – Δ Δ 0 –
TAP	Transfer A to CCR; (A)⇒CCR Assembled as TFR A, CCR	INH	B7 02	P	Δ ⇓ Δ Δ Δ Δ Δ Δ
TBA	Transfer B to A; (B)⇒A	INH	18 0F	OO	– – – – Δ Δ 0 –
TBEQ *abdxysp,rel9*	Test and branch if equal to 0 If (counter)=0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TBL *oprx0_xysppc*	Table lookup and interpolate, 8-bit (M)+[(B)×((M+1)–(M))]⇒A	IDX	18 3D xb	ORfffP	– – – – Δ Δ – Δ
TBNE *abdxysp,rel9*	Test and branch if not equal to 0 If (counter)≠0, then (PC)+2+rel⇒PC	REL (9-bit)	04 lb rr	PPP (branch) PPO (no branch)	– – – – – – – –
TFR *abcdxysp,abcdxysp*	Transfer from register to register (r1)⇒r2r1 and r2 same size $00:(r1) \Rightarrow r2r1$=8-bit; r2=16-bit $(r1_L) \Rightarrow r2r1$=16-bit; r2=8-bit	INH	B7 eb	P	– – – – – – – – or Δ ⇓ Δ Δ Δ Δ Δ Δ
TPASame as TFR CCR ,A	Transfer CCR to A; (CCR)⇒A	INH	B7 20	P	– – – – – – – –

Source Form	Operation	Address Mode	Machine Coding (Hex)	Access Detail	S X H I N Z V C
TRAP *trapnum*	Trap unimplemented opcode; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP} 1⇒I; (trap vector)⇒PC	INH	18 tn tn = $30–$39 or tn = $40–$FF	OVSPSSPSsP	– – – 1 – – – –
TST *opr16a* TST *oprx0_xysppc* TST *oprx9,xysppc* TST *oprx16,xysppc* TST [D,*xysppc*] TST [*oprx16,xysppc*] TSTA TSTB	Test M; (M)–0 Test A; (A)–0 Test B; (B)–0	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	F7 hh ll E7 xb E7 xb ff E7 xb ee ff E7 xb E7 xb ee ff 97 D7	rPO rPf rPO frPP fIfrPf fIPrPf O O	– – – – Δ Δ 0 0
TSXSame as TFR SP,X	Transfer SP to X; (SP)⇒X	INH	B7 75	P	– – – – – – – –
TSYSame as TFR SP,Y	Transfer SP to Y; (SP)⇒Y	INH	B7 76	P	– – – – – – – –
TXSSame as TFR X,SP	Transfer X to SP; (X)⇒SP	INH	B7 57	P	– – – – – – – –
TYSSame as TFR Y,SP	Transfer Y to SP; (Y)⇒SP	INH	B7 67	P	– – – – – – – –
WAI	Wait for interrupt; (SP)–2⇒SP RTN_H:RTN_L⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (Y_H:Y_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (X_H:X_L)⇒M_{SP}:M_{SP+1} (SP)–2⇒SP; (B:A)⇒M_{SP}:M_{SP+1} (SP)–1⇒SP; (CCR)⇒M_{SP}	INH	3E	OSSSSsf (before interrupt) fVfPPP (after interrupt)	– – – – – – – – or – – – 1 – – – – or – 1 – 1 – – – –
WAV	$\sum_{i=1}^{B} S_i F_i \Rightarrow Y{:}D$ $\sum_{i=1}^{B} F_i \Rightarrow X$ Calculate weighted average; sum of products (SOP) and sum of weights (SOW)*	Special	18 3C	Of(frr^ffff)O** SSS+UUUrr^***	– – ? – ? Δ ? ?
*Initialize B, X, and Y: B=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-bit values. **The frr^ffff sequence is the loop for one iteration of SOP and SOW accumulation. The ^ denotes a check for pending interrupt requests. ***Additional cycles caused by an interrupt: SSS is the exit sequence and UUUrr^ is the re-entry sequence. Intermediate values use six stack bytes.					
wavr*	Resume executing interrupted WAV	Special	3C	UUUrr^ffff(frr^ ffff)O** SSS+UUUrr^***	– – ? – ? Δ ? ?
*wavr is a pseudoinstruction that recovers intermediate results from the stack rather than initializing them to 0. **The frr^ffff sequence is the loop for one iteration of SOP and SOW recovery. The ^ denotes a check for pending interrupt requests. ***These are additional cycles caused by an interrupt: SSS is the exit sequence and UUUrr^ is the re-entry sequence.					
XGDXSame as EXG D, X	Exchange D with X; (D)⇔(X)	INH	B7 C5	P	– – – – – – – –
XGDYSame as EXG D, Y	Exchange D with Y; (D)⇔(Y)	INH	B7 C6	P	– – – – – – – –

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.4.1 Register and Memory Notation

Table 5-2 Register and Memory Notation

A or *a*	Accumulator A
An	Bit n of accumulator A
B or *b*	Accumulator B
Bn	Bit n of accumulator B
D or *d*	Accumulator D
Dn	Bit n of accumulator D
X or *x*	Index register X
X_H	High byte of index register X
X_L	Low byte of index register X
Xn	Bit n of index register X
Y or *y*	Index register Y
Y_H	High byte of index register Y
Y_L	Low byte of index register Y
Yn	Bit n of index register Y
SP or *sp*	Stack pointer
SPn	Bit n of stack pointer
PC or *pc*	Program counter
PC_H	High byte of program counter
PC_L	Low byte of program counter
CCR or *c*	Condition code register
M	Address of 8-bit memory location
Mn	Bit n of byte at memory location M
Rn	Bit n of the result of an arithmetic or logical operation
In	Bit n of the intermediate result of an arithmetic or logical operation
RTN_H	High byte of return address
RTN_L	Low byte of return address
( )	Contents of

5.4.2 Source Form Notation

The **Source Form** column of the summary in **Table 5-1** gives essential information about assembler source forms. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Everything in the **Source Form** column, *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 (A, B, D), are literal characters.

The groups of italic characters shown in **Table 5-3** 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 *xysppc* and *oprx0_xysppc* are both valid, but the two groups *oprx0 xysppc* are not valid because there is a space between them.

Table 5-3 Source Form Notation

abc	Register designator for A, B, or CCR
abcdxysp	Register designator for A, B, CCR, D, X, Y, or SP
abd	Register designator for A, B, or D
abdxysp	Register designator for A, B, D, X, Y, or SP
dxysp	Register designator for D, X, Y, or SP
msk8	8-bit mask value Some assemblers require the # symbol before the mask value.
opr8i	8-bit immediate value
opr16i	16-bit immediate value
opr8a	8-bit address value used with direct address mode
opr16a	16-bit address value
oprx0_xysp	Indexed addressing postbyte code: *oprx3,–xysp* — Predecrement X , Y, or SP by 1–8 *oprx3,+xysp* — Preincrement X , Y, or SP by 1–8 *oprx3,xysp–* — Postdecrement X, Y, or SP by 1–8 *oprx3,xysp+* — Postincrement X, Y, or SP by 1–8 *oprx5,xysppc* — 5-bit constant offset from X, Y, SP, or PC *abd,xysppc* — Accumulator A, B, or D offset from X, Y, SP, or PC
oprx3	Any positive integer from 1 to 8 for pre/post increment/decrement
oprx5	Any integer from –16 to +15
oprx9	Any integer from –256 to +255
oprx16	Any integer from –32,768 to +65,535
page	8-bit value for PPAGE register Some assemblers require the # 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 the 64K byte memory space
trapnum	Any 8-bit integer from $30 to $39 or from $40 to $FF
xysp	Register designator for X or Y or SP
xysppc	Register designator for X or Y or SP or PC

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.4.3 Operation Notation

Table 5-4 Operation Notation

+	Add
–	Subtract
•	AND
|	OR
⊕	Exclusive OR
×	Multiply
÷	Divide
:	Concatenate
⇒	Transfer
⇔	Exchange

5.4.4 Address Mode Notation

Table 5-5 Address Mode Notation

INH	Inherent; no operands in instruction stream
IMM	Immediate; operand immediate value in instruction stream
DIR	Direct; operand is lower byte of address from $0000 to $00FF
EXT	Operand is a 16-bit address
REL	Two’s complement relative offset; for branch instructions
IDX	Indexed (no extension bytes); includes: 5-bit constant offset from X, Y, SP or PC Pre/post increment/decrement by 1–8 Accumulator A, B, or D offset
IDX1	9-bit signed offset from X, Y, SP, or PC; 1 extension byte
IDX2	16-bit signed offset from X, Y, SP, or PC; 2 extension bytes
[IDX2]	Indexed-indirect; 16-bit offset from X, Y, SP, or PC
[D, IDX]	Indexed-indirect; accumulator D offset from X, Y, SP, or PC

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

5.4.5 Machine Code Notation

In the **Machine Code (Hex)** column of the summary in **Table 5-1**, digits 0–9 and upper case letters A–F represent hexadecimal values. Pairs of lower-case letters represent 8-bit values as shown in **Table 5-6**.

Table 5-6 Machine Code Notation

dd	8-bit direct address from $0000 to $00FF; high byte is $00
ee	High byte of a 16-bit constant offset for indexed addressing
eb	Exchange/transfer postbyte
ff	Low eight bits of a 9-bit signed constant offset in indexed addressing, or low byte of a 16-bit constant offset in indexed addressing
hh	High byte of a 16-bit extended address
ii	8-bit immediate data value
jj	High byte of a 16-bit immediate data value
kk	Low byte of a 16-bit immediate data value
lb	Loop primitive (DBNE) postbyte
ll	Low byte of a 16-bit extended address
mm	8-bit immediate mask value for bit manipulation instructions; bits that are set indicate bits to be affected
pg	Program page or bank number used in CALL instruction
qq	High byte of a 16-bit relative offset for long branches
tn	Trap number from $30 to $39 or from $40 to $FF
rr	Signed relative offset $80 (–128) to $7F (+127) relative to the byte following the relative offset byte, or low byte of a 16-bit relative offset for long branches
xb	Indexed addressing postbyte

5.4.6 Access Detail Notation

A single-letter code in the **Access Detail** column of **Table 5-1** represents a single CPU access cycle. An upper-case letter indicates a 16-bit access.

Table 5-7 Access Detail Notation

f	Free cycle. During an f cycle, the CPU does not use the bus. An f cycle is always one cycle of the system bus clock. An f cycle can be used by a queue controller or the background debug system to perform a single-cycle access without disturbing the CPU.
g	Read PPAGE register. A g cycle is used only in CALL instructions and is not visible on the external bus. Since PPAGE is an internal 8-bit register, a g cycle is never stretched.
I	Read indirect pointer. Indexed-indirect instructions use the 16-bit indirect pointer from memory to address the instruction operand. An I cycle is a 16-bit read that can be aligned or misaligned. An I cycle is 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. An I cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 5-7 Access Detail Notation (Continued)

i	Read indirect PPAGE value. An i cycle is used only in indexed-indirect CALL instructions. The 8-bit PPAGE value for the CALL destination is fetched from an indirect memory location. An i cycle is stretched only when controlled by a chip-select circuit that is programmed for slow memory.
n	Write PPAGE register. An n cycle is used only in CALL and RTC instructions to write the destination value of the PPAGE register and is not visible on the external bus. Since the PPAGE register is an internal 8-bit register, an n cycle is never stretched.
O	Optional cycle. An O cycle adjusts instruction alignment in the instruction queue. An O cycle can be a free cycle (f) or a program word access cycle (P). When the first byte of an instruction with an odd number of bytes is misaligned, the O cycle becomes a P cycle to maintain queue order. If the first byte is aligned, the O cycle is an f cycle. The $18 prebyte for a page-two opcode is treated as a special one-byte instruction. If the prebyte is misaligned, the O cycle at the beginning of the instruction becomes a P cycle to maintain queue order. If the prebyte is aligned, the O cycle is an f cycle. If the instruction has an odd number of bytes, it has a second O cycle at the end. If the first O cycle is a P cycle (prebyte misaligned), the second O cycle is an f cycle. If the first O cycle is an f cycle (prebyte aligned), the second O cycle is a P cycle. An O cycle that becomes a P cycle 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. An O cycle that becomes an f cycle is never stretched.
P	Program word access. Program information is fetched as aligned 16-bit words. A P cycle is 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. An r cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
R	16-bit data read. An R cycle is 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. An R cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
s	Stack 8-bit data. An s cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
S	Stack 16-bit data. An S cycle is 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. An S cycle is also stretched if it corresponds to a misaligned access 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. A w cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
W	16-bit data write. A W cycle is 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. A W cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
u	Unstack 8-bit data. A W cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.

Table 5-7 Access Detail Notation (Continued)

U	Unstack 16-bit data. A U cycle is 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. A U cycle is also stretched if it corresponds to a misaligned access 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	16-bit vector fetch. Vectors are always aligned 16-bit words. A V cycle is 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. A t cycle is either a data read cycle or a free cycle, depending on the data and flow of the REVW instruction. A t cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
T	16-bit conditional read. A T cycle is either a data read cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. A T cycle is 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. A T cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
x	8-bit conditional write. An x cycle is either a data write cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. An x cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
Special Notation for Branch Taken/Not Taken	
PPP/P	A short branch requires 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	A long branch requires four cycles if taken, three cycles if not taken. An O cycle is required because all long branches are page two opcodes and thus include the $18 prebyte. The prebyte is treated as a one-byte instruction. If the prebyte is misaligned, the O cycle is a P cycle; if the prebyte is aligned, the O cycle is an f cycle. As a result, both the taken and not-taken cases use one O cycle for the prebyte. In the not-taken case, the queue must advance so that execution can continue with the next instruction, and another O 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

5.4.7 Condition Code State Notation

Table 5-8 Condition Code State Notation

–	Not changed by operation
0	Cleared by operation
1	Set by operation
Δ	Set or cleared by operation
⇓	May be cleared or remain set, but not set by operation
⇑	May be set or remain cleared, but not cleared by operation
?	May be changed by operation but final state not defined
!	Used for a special purpose

5.5 External Visibility Of Instruction Queue

The instruction queue buffers program information and increases instruction throughput. The queue consists of three 16-bit stages. Program information is always fetched in aligned 16-bit words. Normally, at least three bytes of program information are available to the CPU when instruction execution begins.

Program information is fetched and queued a few cycles before it is used by the CPU. 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, IPIPE[1:0], 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 CPU at a specific instruction. By the time an operation is visible outside the system, 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 **14.4.8 Instruction Tagging**.

5.5.1 Instruction Queue Status Signals

The IPIPE[1:0] signals carry time-multiplexed information about data movement and instruction execution during normal operation. The signals are available on two multifunctional device pins. During reset, the pins are mode-select inputs MODA and MODB. After reset, information on the pins does not become valid until an instruction reaches stage two of the queue.

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 two-bit values, with the low bit on IPIPE0 and the high bit on IPIPE1. Data-movement information is available when E clock is high or on falling edges of the E clock; execution-start information is available when E clock is low or on rising edges of the E clock, as shown in **Figure 5-1**. Data-movement information refers to data on the

bus. Execution-start information is delayed one bus cycle to guarantee the indicated opcode is in stage three. **Table 5-9** summarizes the information encoded on the IPIPE[1:0] pins.

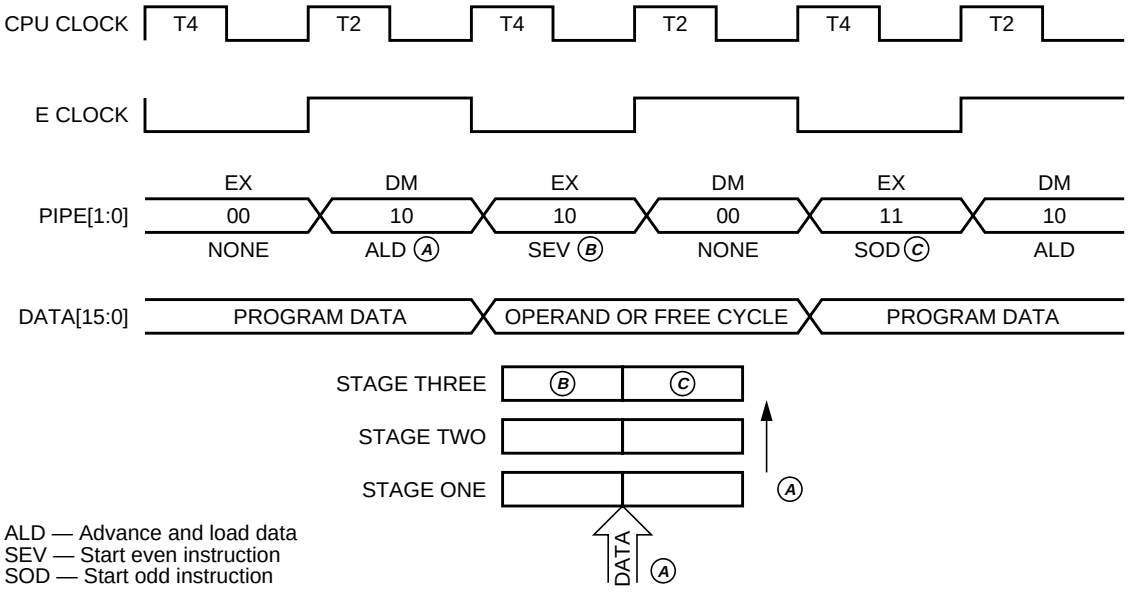


Figure 5-1 Queue Status Signal Timing

Data movement status is valid when the E clock is high and is represented by two states:

- No movement — There is no data shifting in the queue.
- Advance and load from data bus — The queue shifts up one stage with stage one being filled with the data on the read data bus.

Execution start status is valid when the E clock is low and is represented by four states:

- No start — Execution of the current instruction continues.
- Start interrupt — An interrupt sequence has begun.

NOTE: *The start-interrupt state is indicated when an interrupt request or tagged instruction alters program flow. SWI and TRAP instructions are part of normal program flow and are indicated as start even or start odd depending on their alignment. Since they are present in the queue, they can be tracked in an external queue rebuild. An external event that interrupts program flow is indeterministic. Program data is not present in the queue until after the vector jump.*

- Start even instruction — The current opcode is in the high byte of stage three of the queue.
- Start odd instruction — The current opcode is in the low byte of stage three of the queue.

Table 5-9 IPIPE[1:0] Decoding when E Clock is High

Data Movement (capture at E fall)	Mnemonic	Meaning
0:0	—	No movement

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Table 5-9 IPIPE[1:0] Decoding when E Clock is High

Data Movement (capture at E fall)	Mnemonic	Meaning
0:1	—	Reserved
1:0	ALD	Advance queue and load from bus
1:1	—	Reserved

Table 5-10 IPIPE[1:0] Decoding when E Clock is Low

Execution Start (capture at E rise)	Mnemonic	Meaning
0:0	—	No start
0:1	INT	Start interrupt sequence
1:0	SEV	Start even instruction
1:1	SOD	Start odd instruction

The execution-start status signals are delayed by one E clock cycle to allow a lagging program fetch and queue advance. Therefore the execution-start status always refers to the data in stage three of the queue.

The advance and load from bus signal can be used as a load-enable to capture the instruction word on the data bus. This signal is effectively the queue advance signal inside the CPU. Program data is registered into stage one on the rising edge of t4 when queue advance is asserted.

5.5.2 No Movement (0:0)

The 0:0 state at the falling edge of E indicates that there is no data movement in the instruction queue during the current cycle. The 0:0 state at the rising edge of E indicates continuation of an instruction or interrupt sequence during the previous cycle.

5.5.3 ALD — Advance and Load from Data Bus (1:0)

The three-stage instruction queue is advanced by one word and stage one is refilled with a word of program information from the data bus. The CPU requested the information two bus cycles earlier but, due to access delays, the information was not available until the E cycle immediately prior to the ALD.

5.5.4 INT — Start Interrupt (0:1)

This state indicates program flow has changed to 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 lower byte of the 16-bit interrupt vector.

5.5.5 SEV — Start Even Instruction (1:0)

This state indicates that the instruction is in the even (high) half of the word in stage three of the instruction queue. The queue treats the $18 prebyte of an instruction on page two of the opcode map as a special

one-byte, one-cycle instruction. However, interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

5.5.6 SOD — Start Odd Instruction (1:1)

This state indicates that the instruction in the odd (low) half of the word in stage three of the instruction queue. The queue treats the $18 prebyte of an instruction on page two of the opcode map as a special one-byte, one-cycle instruction. However, interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 6 Exception Processing

Exceptions are events that require a change in the sequence of instruction execution. This section describes the exceptions supported by the Core and their functionality.

6.1 Exception Processing Overview

The Core supports two basic types of exceptions; those from resets and those from interrupt requests. Regardless of the source, the first cycle in exception processing is a vector fetch cycle. The exception processing flow is shown in **Figure 6-1** below. Relevant points within the flow are detailed in the paragraphs that follow.

1.0-V

During the vector fetch cycle, the CPU indicates to the system that it is requesting that the vector address of the pending exception having the highest priority be driven onto the address bus. The CPU does not provide this address.

The vector points to the address where the exception service routine begins. Exception vectors are stored in a table at the top of the memory map ($FFB6–$FFFF). The CPU begins using the vector to fetch instructions in the third cycle of the exception processing sequence.

After the vector fetch, the CPU selects one of the three processing paths based on the source of the exception:

- Reset
- X bit maskable and I bit maskable interrupt request
- SWI and TRAP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

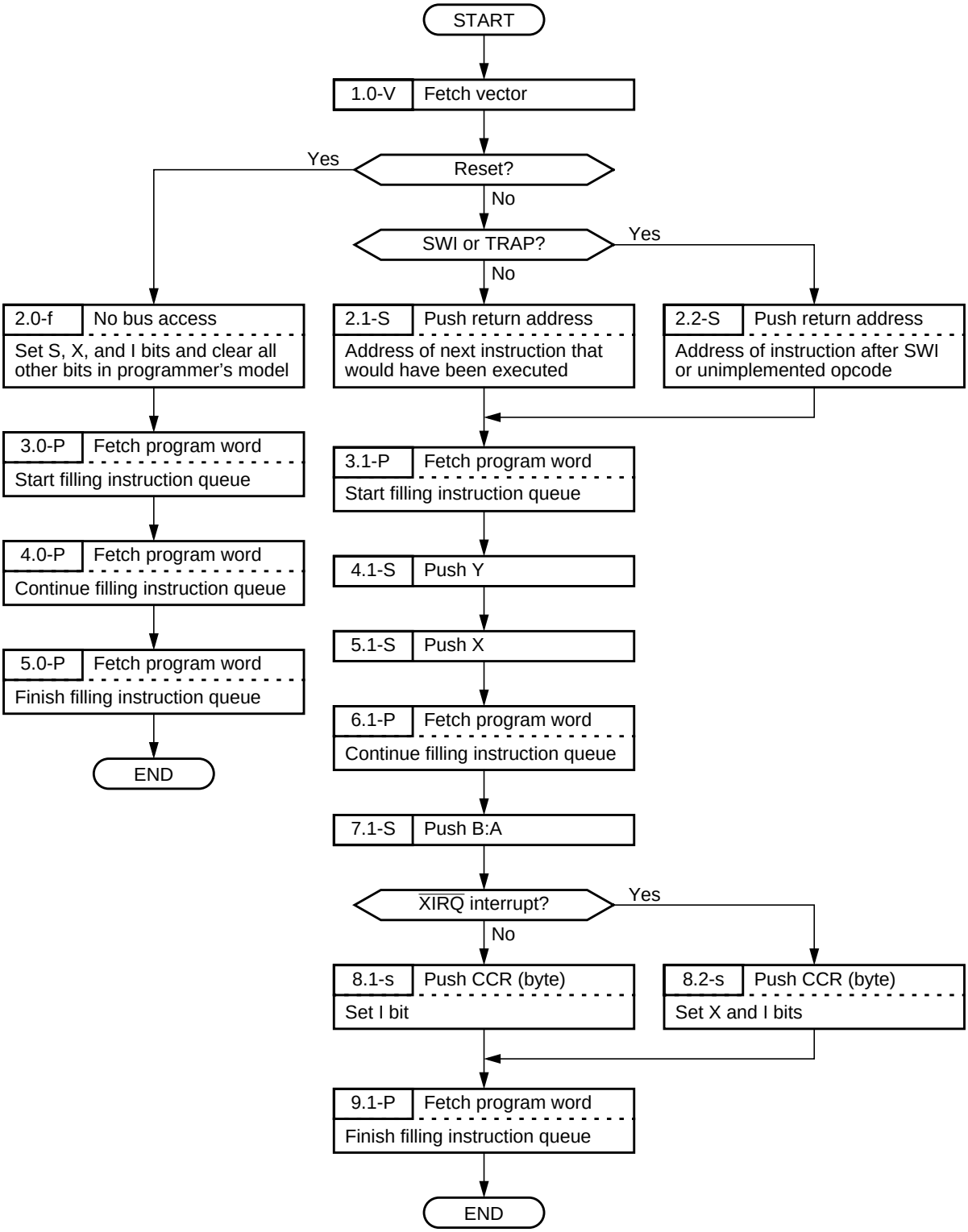


Figure 6-1 Exception Processing Flow

6.1.1 Reset Processing

2.0-f

This cycle sets the S, X and I bits.

3.0-P through 5.0-P

These cycles 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, reset processing ends, and the CPU starts executing the instruction at the head of the instruction queue.

6.1.2 Interrupt Processing

The SWI and TRAP interrupts have no mask or interrupt request and are always recognized. An $\overline{\text{XIRQ}}$ interrupt request is recognized any time after the X bit is cleared. An enabled I bit maskable interrupt request is recognized any time after the I bit is cleared. The CPU responds to an interrupt after it completes the execution of its current instruction. Interrupt latency depends on the number of cycles required to complete the instruction.

After the vector fetch, the CPU calculates a return address. The return address depends on the type of exception:

- When an X bit maskable or I bit maskable interrupt causes the exception, the return address points to the next instruction that would have been executed had processing not been interrupted.
- When an SWI opcode or TRAP causes the exception, the return address points to the next address after the SWI opcode or to the next address after the unimplemented opcode.

2.1-S and 2.2-S

These are both S cycles (16-bit writes) that push the return address onto the stack.

3.1-P

This cycle is the first of three program word fetches to refill the instruction queue. Instructions are fetched from the address pointed to by the vector.

4.1-S

This cycle pushes Y onto the stack.

5.1-S

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

This cycle pushes X onto the stack.

6.1-P

This cycle is the second of three program word fetches to refill the instruction queue. During this cycle, the contents of the A and B accumulators are concatenated in the order B:A, making register order in the stack frame the same as that of the M68HC11, M6801, and the M6800.

7.1-S

This cycle pushes the 16-bit word containing B:A onto the stack.

8.1-s and 8.2-s

These are both s cycles (8-bit writes) that push the 8-bit CCR onto the stack and then update the X and I mask bits:

- When an $\overline{\text{XIRQ}}$ interrupt causes the exception, both X and I are set to inhibit further interrupts during exception processing.
- When any other interrupt causes the exception, the I bit is set to inhibit further I bit maskable interrupts during exception processing, but the X bit is not changed.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

9.1-P

This cycle is the third of three program word fetches to refill the instruction queue. It is the last cycle of exception processing. After this cycle the CPU begins the interrupt service routine by executing the instruction at the head of the instruction queue.

At the end of the interrupt service routine, an RTI instruction restores the stacked registers, and the CPU returns to the return address. RTI is an 8-cycle instruction when no other interrupt is pending, and an 11-cycle instruction when another interrupt is pending. In either case, the first five cycles are used to 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 at the head of the queue.

6.2 Exception Vectors

Each exception has a 16-bit vector that points to the memory location where the routine that handles the exception is located. Vectors are stored in the upper 128 bytes of the standard 64K byte address map and are prioritized as shown in **Table 6-1** below from highest (system reset) to lowest (lowest priority I maskable interrupt).

Table 6-1 Exception Vector Map and Priority

Vector Address	Source
$FFFE–$FFFF	System reset
$FFFC–$FFFD	Crystal Monitor reset
$FFFA–$FFFB	COP reset
$FFF8–$FFF9	Unimplemented opcode trap
$FFF6–$FFF7	Software interrupt instruction (SWI) or BDM vector request
$FFF4–$FFF5	$\overline{\text{XIRQ}}$ signal
$FFF2–$FFF3	$\overline{\text{IRQ}}$ signal
$FFF0–$FF00	Device-specific I bit maskable interrupt sources (priority in descending order)

The six highest vector addresses are used for resets and nonmaskable 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

6.3 Exception Types

As stated previously, the Core supports exceptions from resets within the system as well as interrupt requests. Each of these exception types are discussed in the subsections that follow.

6.3.1 Resets

A block (or blocks) within the SoC design must evaluate any/all reset sources and request the proper reset vector from the Core. The CPU then fetches a vector determined by the source of the reset, configures the CPU registers to their reset states and fills the instruction queue from the address pointed to by the vector.

There are three reset sources supported by the Core:

- System reset
- Crystal Monitor reset
- COP Watchdog reset

The priority and vector addresses assigned to these reset sources are shown in **Table 6-2** below. Please note that the inclusion of Crystal Monitor and COP reset requests is based upon the two most common and predominately used requests historically implemented in HC12 based systems. (It is assumed that all systems will have a system reset). Each SoC integration of the Core will determine whether the system contains both requests, one or the other or neither request. Each source is described in the subsections that follow.

Table 6-2 Reset Sources

Reset Source	Exception Priority	Vector Address
System reset	1	$FFFE–$FFFF
Crystal Monitor block	2	$FFFC–$FFFD
Computer Operating Properly (COP) block	3	$FFFA–$FFFB

6.3.1.1 System reset

All systems generally have a block or sub-block within the system that determines the validity and priority of all possible sources of a system reset request. When a valid system reset request becomes active, the block or sub-block will request the appropriate reset vector from the Core. The Core will then acknowledge the request and provide the vector.

6.3.1.2 Crystal Monitor Reset

A Crystal Monitor sub-block typically contains a mechanism to determine whether or not the system clock frequency is above a predetermined limit. If the clock frequency falls below the limit when the Crystal Monitor is enabled, the sub-block will typically request the reset vector that is associated with this function from the Core.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

6.3.1.3 COP Reset

A Computer Operating Properly (COP) sub-block helps protect against software failures. When the COP is enabled, software might, for example, write a particular code sequence to a specific address in order to keep a watchdog timer from timing out. If software fails to execute the sequence properly, the sub-block will typically then request a reset vector from the Core.

6.3.2 Interrupts

The Core supports the following types of interrupt sources:

- nonmaskable interrupt requests
  - Unimplemented Opcode Trap
  - Software Interrupt instruction
  - $\overline{\text{XIRQ}}$ pin interrupt request
- Maskable interrupt requests
  - Optional highest priority maskable interrupt (defaults to $\overline{\text{IRQ}}$ pin)
  - $\overline{\text{IRQ}}$ pin interrupt request
  - System peripheral block I bit maskable interrupt requests

A block (or blocks) within the SoC design must evaluate the system peripheral block I bit maskable interrupt sources and request the proper interrupt vector from the Core. All other interrupt requests are handled within the Core. Once the CPU receives the request it then fetches the vector to the proper interrupt service routine. The CPU will then calculate and stack a return address and the contents of the CPU registers. Finally, it will set the I bit (and the X bit if $\overline{\text{XIRQ}}$ is the source) and fill the instruction queue from the address pointed to by the vector. The vector mapping for all interrupt sources is shown in **Table 6-3** below with detailed descriptions given in the sub-sections that follow.

Table 6-3 Interrupt Sources

Interrupt Source	Exception Priority	Mask	Vector Address
Unimplemented opcode trap (TRAP)	4	None	$FFF8–$FFF9
Software interrupt instruction (SWI)	4	None	$FFF6–$FFF7
Nonmaskable external interrupt pin ($\overline{\text{XIRQ}}$ pin)	5	X bit	$FFF4–$FFF5
Highest priority I-Maskable interrupt (defaults to $\overline{\text{IRQ}}$ pin)	6	I bit	$FFxx-$FFxx+1
Maskable external interrupt pin ($\overline{\text{IRQ}}$ pin)	6 or 7	I bit	$FFF2–$FFF3
System peripheral block interrupt requests	≥ 8	I bit	$FFF0–$FF00

Interrupts can be classified according to their maskability. TRAP and SWI are nonmaskable. The $\overline{\text{XIRQ}}$ pin is masked at reset by the X bit, but once software clears the X bit, the $\overline{\text{XIRQ}}$ pin is nonmaskable until another reset occurs. The remaining interrupt sources can be masked by the I bit. I bit maskable interrupt

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

requests come from the $\overline{\text{IRQ}}$ pin and peripheral blocks within the system such as timers and serial ports. These I bit 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 request. The $\overline{\text{IRQ}}$ pin is initially assigned the highest I bit maskable interrupt priority. The system can give one I bit maskable source priority over other I bit maskable sources configured at integration of the Core into the SoC design. The documentation for each system should provide more information.

6.3.2.1 Unimplemented Opcode Trap (TRAP)

Only 54 of the 256 positions on page 2 of the opcode map are used. Attempting to execute one of the 202 unused opcodes on page 2 causes a nonmaskable interrupt without an interrupt request. All 202 unused opcodes share the same interrupt vector, $FFF8:$FFF9.

TRAP processing stacks the CCR and then sets the I bit to prevent other interrupts during the TRAP service routine. An RTI instruction at the end of the service routine restores the I bit to its preinterrupt state.

The CPU 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. The stacked return address can be used to calculate the address of the unimplemented opcode for software-controlled traps.

6.3.2.2 Software Interrupt Instruction (SWI)

Execution of the SWI instruction causes a nonmaskable interrupt without an interrupt request.

SWI processing stacks the CCR and then sets the I bit to prevent other interrupts during the SWI service routine. An RTI instruction at the end of the service routine restores the I bit to its preinterrupt state.

NOTE: *CPU processing of a TRAP or SWI cannot be interrupted. Also, TRAP and SWI are mutually exclusive instructions with no relative priority.*

6.3.2.3 Nonmaskable External Interrupt Request Pin ($\overline{\text{XIRQ}}$)

Driving the $\overline{\text{XIRQ}}$ pin low generates an external interrupt request, subject initially to masking by the X bit. Reset sets the X bit, masking $\overline{\text{XIRQ}}$ interrupt requests. Software can unmask $\overline{\text{XIRQ}}$ interrupt requests once after reset by clearing the X bit with an instruction such as ANDCC #$BF. After the X bit has been cleared, it cannot be set and $\overline{\text{XIRQ}}$ interrupt requests are nonmaskable until another reset occurs.

$\overline{\text{XIRQ}}$ interrupt request processing stacks the CCR and then sets both the X and I bits to prevent other interrupts during the $\overline{\text{XIRQ}}$ service routine. An RTI instruction at the end of the service routine restores the X and I bits to their preinterrupt states.

6.3.2.4 Maskable External Interrupt Request Pin ($\overline{\text{IRQ}}$)

Driving the $\overline{\text{IRQ}}$ pin low generates an external interrupt request, subject to masking by the I bit. $\overline{\text{IRQ}}$ interrupt request processing stacks the CCR and then sets the I bit to prevent other interrupts during the $\overline{\text{IRQ}}$ service routine. An RTI instruction at the end of the service routine restores the I bit to its preinterrupt state.

The Interrupt sub-block of the Core (INT) also has a control bit to disconnect the $\overline{IRQ}$ input. Please see **Section 10** of this guide for a more detailed description.

6.3.2.5 System Peripheral Block Interrupt Requests

Some system peripheral blocks can generate interrupt requests that are subject to masking by the I bit. Processing of an interrupt request from one of these sources stacks the CCR and then sets the I bit to prevent other interrupts during the service routine. An RTI instruction at the end of the service routine restores the I bit to its preinterrupt state.

Interrupt requests from a system peripheral block may also be subject to masking by interrupt enable bits in control registers. In addition, there may be interrupt flags with register read-write sequences required for flag clearing. The documentation for the system peripheral block should provide a detailed functional description.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 7 Core Interface

This section provides a brief description of the Core interface to the rest of the SoC design. Detailed information on the Core interface, such as more complete descriptions of all signals and timing information, is provided in the **STAR12 V1.5 Core Integration Guide**.

7.1 Core Interface Overview

The Core is designed to be integrated into a SoC design as a fully synthesizable block. The Core interface is shown in **Figure 7-1** below with the interface signals grouped by function. All signals related to the internal and I.P. bus interfacing appear on the right side of the Core block in the diagram. In addition to bus interfacing, the Core receives reset and clock inputs from the system and provides signals for interacting with the CPU for vector request and acknowledge and for functional operation of the stop and wait modes. The Core interacts with the external blocks of the overall system through the port/pad logic for Ports A, B, E (which include the physical $\overline{\text{IRQ}}$ and $\overline{\text{XIRQ}}$ pins) and K and the BDM BKGD pin interfaces. The memory configuration switches shown in the diagram are inputs to the Core block that are tied to a constant logic state at the time of integration into the SoC design to correctly define the on-chip memory configuration for proper Core operation within the system.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

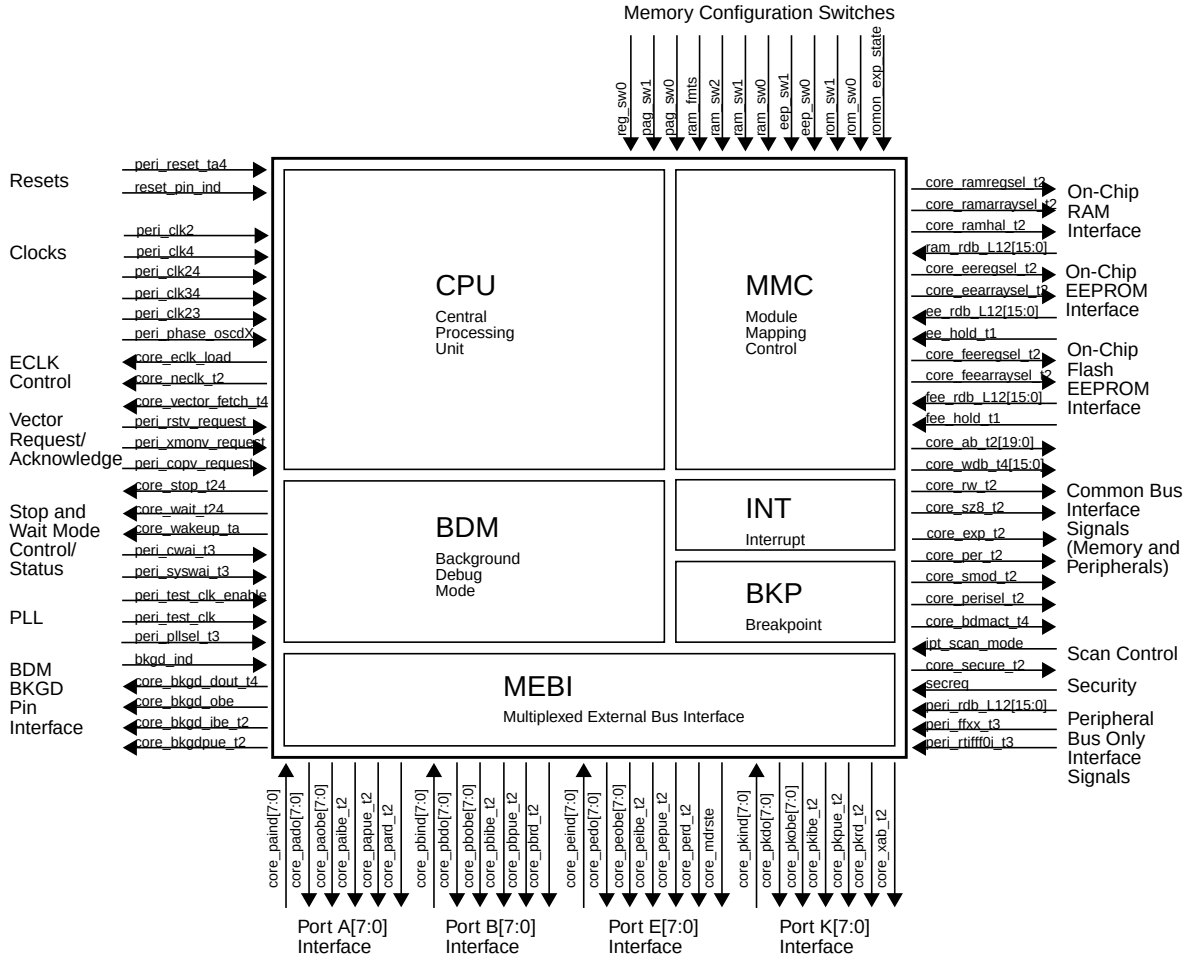


Figure 7-1 Core Interface Signals

7.1.1 Signal Summary

A brief summary of the Core interface signals is given in **Table 7-1** below. For detailed descriptions and timing information please consult the **STAR12 V1.5 Core Integration Guide**.

Table 7-1 Core Interface Signal Definitions

Signal Name	Type	Functional Description
Internal Bus Interface Signals		
core_ab_t2[19:0]	O	Core 16-bit Address Bus [19:0]
peri_rdb_L12[15:0]	I	16-bit Read Data Bus data from Peripheral block
ram_rdb_L12[15:0]	I	16-bit Read Data Bus data from on-chip RAM array
ee_rdb_L12[15:0]	I	16-bit Read Data Bus data from on-chip EEPROM array
fee_rdb_L12[15:0]	I	16-bit Read Data Bus data from on-chip Flash EEPROM or ROM array
core_wdb_t4[15:0]	O	Core 16-bit Write Data Bus [15:0]
core_rw_t2	O	Core Read/$\overline{\text{Write}}$ signal - active low Write

Table 7-1 Core Interface Signal Definitions

Signal Name	Type	Functional Description
core_sz8_t2	O	Core bus data size requested signal 0 - 16-bit access 1 - 8-bit access
core_exp_t2	O	Expanded Mode selected signal
core_per_t2	O	Peripheral Test Mode selected signal
core_smod_t2	O	Special Mode selected signal
core_secure_t2	O	Core secure mode signal
core_perisel_t2	O	Core peripheral select to I.P. Bus Interface
core_ramregsel_t2	O	On-chip RAM Register select from Core to memory and/or bus
core_ramarraysel_t2	O	On-chip RAM Array select from Core to memory and /or bus
core_ramhal_t2	O	On-chip RAM Array align signal from Core to memory and/or bus
core_eeregsel_t2	O	On-chip EEPROM Register select from Core to memory and/or bus
core_eearraysel_t2	O	On-chip EEPROM Array select from Core to memory and/or bus
core_feeregsel_t2	O	On-chip Flash EEPROM Register select from Core to memory and/or bus
core_feearraysel_t2	O	On-chip Flash EEPROM Array select from Core to memory and/or bus
ee_hold_t1	I	On-chip EEPROM signal to Core to suspend CPU operation
fee_hold_t1	I	On-chip Flash EEPROM signal to Core to suspend CPU operation
secreq	I	Security mode request from applicable memory
peri_ffxx_t3	I	Interrupt Bus from I.P. Bus Interface
peri_rtifff0i_t3	I	Real Time Interrupt signal
core_bdmact_t4	O	Core BDM active signal for I.P. Bus Interface (freeze signal)
External Bus Interface Signals		
core_paind[7:0]	I	Port A input data [7:0]
core_pado[7:0]	O	Port A data output [7:0]
core_paobe[7:0]	O	Port A output buffer enable [7:0]
core_paibe_t2	O	Port A input buffer enable
core_papue_t2	O	Port A pullup enable
core_padse_t2	O	Port A drive strength enable
core_pbind[7:0]	I	Port B input data [7:0]
core_pbdo[7:0]	O	Port B data output [7:0]
core_pbobe[7:0]	O	Port B output buffer enable [7:0]
core_pbibe_t2	O	Port B input buffer enable
core_pbpue_t2	O	Port B pullup enable
core_pbdse_t2	O	Port B drive strength enable
core_peind[7:0]	I	Port E input data [7:0] *NOTE:* PE1 is $\overline{\text{IRQ}}$ pin input; PE0 is $\overline{\text{XIRQ}}$ pin input.
core_pedo[7:0]	O	Port E data output [7:0]
core_peobe[7:0]	O	Port E output buffer enable [7:0]
core_peibe_t2	O	Port E input buffer enable
core_pepue_t2	O	Port E pullup enable
core_mdrste	O	Enable signal for EBI Mode pin pullups at the pad
core_pedse_t2	O	Port E drive strength enable

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 7-1 Core Interface Signal Definitions

Signal Name	Type	Functional Description
core_pkind[7:0]	I	Port K input data [7:0]
core_pkdo[7:0]	O	Port K data output [7:0]
core_pkobe[7:0]	O	Port K output buffer enable [7:0]
core_pkibe_t2	O	Port K input buffer enable
core_pkpue_t2	O	Port K pullup enable
core_pkdse_t2	O	Port K drive strength enable
Clock and Reset Signals		
See **Section 8** of this guide.		
Vector Request/Acknowledge Signals		
core_vector_fetch_t4	O	Core CPU vector request
peri_rstv_request	I	System level reset vector request
peri_xmonv_request	I	System level Crystal Monitor reset vector request
peri_copv_request	I	System level COP Watchdog reset vector request
Stop and Wait Mode Control/Status Signals		
See **Section 8** of this guide.		
Background Debug Mode (BDM) Interface Signals		
bkgd_ind	I	BDM BKGD pin input data
core_bkgd_dout_t4	O	Data output for BDM BKGD pin
core_bkgd_obe	O	BDM BKGD pin output buffer enable
core_bkgd_ibe_t2	O	BDM BKGD pin input buffer enable
core_bkgdpue_t2	O	BDM BKGD pin pullup enable
Memory Configuration Signals		
reg_sw0	I	Register space size select switch to be tied to the appropriate logic level at system integration: 0 - 1K byte register space aligned to lower address 1 - 2K byte register space.
pag_sw1	I	On-chip memory size select switch bit 1 to be tied to the appropriate logic level at system integration.
pag_sw0	I	On-chip memory size select switch bit 0 to be tied to the appropriate logic level at system integration.
ram_fmts	I	On-chip RAM fast memory transfer select to be tied to the appropriate logic level at system integration.
ram_sw2	I	On-chip RAM size select switch bit 2 to be tied to the appropriate logic level at system integration.
ram_sw1	I	On-chip RAM size select switch bit 1 to be tied to the appropriate logic level at system integration.
ram_sw0	I	On-chip RAM size select switch bit 0 to be tied to the appropriate logic level at system integration.
eep_sw1	I	On-chip EEPROM size select switch bit 1 to be tied to the appropriate logic level at system integration.
eep_sw0	I	On-chip EEPROM size select switch bit 0 to be tied to the appropriate logic level at system integration.
rom_sw1	I	On-chip Flash EEPROM or ROM size select switch bit 1 to be tied to the appropriate logic level at system integration.
rom_sw0	I	On-chip Flash EEPROM or ROM size select switch bit 0 to be tied to the appropriate logic level at system integration.

Table 7-1 Core Interface Signal Definitions

Signal Name	Type	Functional Description
romon_exp_state		Reset state of the ROMON bit in the MISC Register to be tied to the appropriate literal logic level at system integration (i.e. tied level is the state out of reset and not inverted).
Scan Control Interface Signals		
ipt_scan_mode	I	Scan mode select signal

7.2 Signal Descriptions

General descriptions of the Core interface signals are given in the subsections below. The clock, reset and wait and stop mode signals are discussed in **Section 8** of this guide. For detailed descriptions of these signals including timing information please consult the **STAR12 V1.5 Core Integration Guide**.

7.2.1 Internal Bus Interface Signals

These descriptions apply to the Core signals that interface with the on-chip memories either directly or through the Core bus and with the system peripheral blocks through the I.P. Bus Interface.

7.2.1.1 Core 20-bit Address Bus (core_ab_t2[19:0])

This 20-bit wide Core output provides the Core Address Bus to the system memory and peripheral blocks.

7.2.1.2 16-bit Read Data Bus from system peripheral blocks (peri_rdb_L12[15:0])

16-bit wide Read Data Bus input to the Core from the system peripherals via the I.P. Bus Interface block.

7.2.1.3 16-bit Read Data Bus from on-chip RAM (ram_rdb_L12[15:0])

16-bit wide Read Data Bus input to the Core from the on-chip RAM memory block.

7.2.1.4 16-bit Read Data Bus from on-chip EEPROM (ee_rdb_L12[15:0])

16-bit wide Read Data Bus input to the Core from the on-chip EEPROM memory block.

7.2.1.5 16-bit Read Data Bus from on-chip Flash EEPROM or ROM (fee_rdb_L12[15:0])

16-bit wide Read Data Bus input to the Core from the on-chip Flash EEPROM or ROM memory block.

7.2.1.6 Core 16-bit Write Data Bus (core_wdb_t4[15:0])

This 16-bit wide Core output provides the Core Write Data Bus to the system memory and peripheral blocks.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

7.2.1.7 Core Read/$\overline{\text{Write}}$ signal (core_rw_t2)

This single bit Core output indicates the direction of bus access (read or write with write being active low) by the Core.

7.2.1.8 Core bus data size request indicator (core_sz8_t2)

This single bit Core output indicates the size of data (8-bit or 16-bit when high or low, respectively) being read/written by a Core bus access.

7.2.1.9 Core Expanded Mode indicator (core_exp_t2)

This single bit Core output indicates that the Core is in Expanded Mode (i.e. the Core has been configured in one of the expanded modes via the MODE pins)

7.2.1.10 Core Peripheral Test Mode indicator (core_per_t2)

This single bit Core output indicates that the Core is in Peripheral Test Mode. In this mode, the cpu is disabled and the direction of the bus interface is switched such that the on-chip peripherals can be addressed directly. This mode is used for factory test only.

7.2.1.11 Core Special Mode indicator (core_smod_t2)

This single bit Core output indicates that the Core is in Special Mode (i.e. the Core has been configured in Special Mode via the MODE pins)

7.2.1.12 Core Secure Mode indicator (core_secure_t2)

This single bit Core output indicates that the Core is operating in secured mode. Please see **Section 15** of this guide for functional details.

7.2.1.13 Peripheral select signal (core_perisel_t2)

This single bit Core output indicates that the Core is accessing an address within the peripheral space of the system memory map.

7.2.1.14 On-Chip RAM register space select signal (core_ramregsel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip RAM register space of the system memory map.

7.2.1.15 On-Chip RAM array select signal (core_ramarraysel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip RAM array space of the system memory map.

7.2.1.16 On-Chip RAM array align signal (core_ramhal_t2)

This single bit Core output reflects the state of the RAMHAL bit in the INITRAM register within the Module Mapping Control (MMC) sub-block of the Core. Please see **Section 11** of this guide for further functional details.

7.2.1.17 On-Chip EEPROM register select signal (core_eeregsel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip EEPROM register space of the system memory map.

7.2.1.18 On-Chip EEPROM array select signal (core_eearraysel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip EEPROM array space of the system memory map.

7.2.1.19 On-Chip Flash EEPROM register select signal (core_feeregsel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip Flash EEPROM register space of the system memory map.

7.2.1.20 On-Chip Flash EEPROM array select signal (core_feearraysel_t2)

This single bit Core output indicates that the Core is accessing an address within the on-chip Flash EEPROM array space of the system memory map.

7.2.1.21 On-Chip EEPROM hold signal to Core (ee_hold_t1)

This single bit input to the Core is used to suspend operation of the CPU when needed for functions of the on-chip EEPROM memory block.

7.2.1.22 On-Chip Flash EEPROM hold signal to Core (fee_hold_t1)

This single bit input to the Core is used to suspend operation of the CPU when needed for functions of the on-chip Flash EEPROM memory block.

7.2.1.23 Core Security Request (secreq)

This single bit input indicates to the Core that the system memory is in a secured state and that the Core should operate in secured mode. Please see **Section 15** for functional details.

7.2.1.24 56-bit Interrupt request signals from peripheral block to Core (peri_ffxx_t3)

This 56-bit wide input to the Core provides the Core with the Interrupt request signals from all the system interrupt sources via the I.P. Bus Interface.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

7.2.1.25 System Real Time Interrupt request (peri_rtifff0i_t3)

This input signal indicates to the Core that the system is requesting the interrupt vector for a Real Time Interrupt (RTI) from the Core.

7.2.1.26 Background Debug Mode active indicator (core_bdmact_t4)

This single bit output from the Core indicates that the Background Debug Mode (BDM) is active.

7.2.2 External Bus Interface Signals

These descriptions apply to the interface signals between the Core and the system External Bus Interface pad logic. Please see **Section 12** of this guide for further functional details of the External Bus Interface.

7.2.2.1 Port A Input Data to Core (core_paind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port A.

7.2.2.2 Port A Output Data from Core (core_pado[7:0])

This 8-bit wide output from the Core provides the Port A data output to the system port/pad logic for Port A.

7.2.2.3 Port A output buffer enable from Core (core_paobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port A.

7.2.2.4 Port A input buffer enable from Core (core_paibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port A.

7.2.2.5 Port A pullup enable from Core (core_papue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port A should be enabled for all Port A pins.

7.2.2.6 Port A drive strength enable from Core (core_padse_t2)

This single bit output from the Core indicates whether all Port A pins will operate with full or reduced drive strength.

7.2.2.7 Port B Input Data to Core (core_pbind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port B.

7.2.2.8 Port B Output Data from Core (core_pbdo[7:0])

This 8-bit wide output from the Core provides the Port B data output to the system port/pad logic for Port B.

7.2.2.9 Port B output buffer enable from Core (core_pbobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port B.

7.2.2.10 Port B input buffer enable from Core (core_pbibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port B.

7.2.2.11 Port B pullup enable from Core (core_pbpue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port B should be enabled for all Port B pins.

7.2.2.12 Port B drive strength enable from Core (core_pbdse_t2)

This single bit output from the Core indicates whether all Port B pins will operate with full or reduced drive strength.

7.2.2.13 Port E Input Data to Core (core_peind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port E. When the system has an external $\overline{\text{IRQ}}$ pin implemented, the input signal from the $\overline{\text{IRQ}}$ pin pad logic must be tied to Port E Input Data Bit 1. Likewise, when the system has an external $\overline{\text{XIRQ}}$ pin implemented, the input signal from the $\overline{\text{XIRQ}}$ pin pad logic must be tied to Port E Input Data Bit 0. Both the $\overline{\text{IRQ}}$ and $\overline{\text{XIRQ}}$ signals are active low (i.e. their asserted state is logic 0).

7.2.2.14 Port E Output Data from Core (core_pedo[7:0])

This 8-bit wide output from the Core provides the Port E data output to the system port/pad logic for Port E.

7.2.2.15 Port E output buffer enable from Core (core_peobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port E.

7.2.2.16 Port E input buffer enable from Core (core_peibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port E.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

7.2.2.17 Port E pullup enable from Core (core_pepue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port E should be enabled for all Port E pins except the MODA (PE5) and MODB (PE6) pins.

7.2.2.18 Port E MODE pin pullup enable from Core (core_mdrste)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for the MODA (PE5) and MODB (PE6) pins within Port E should be enabled.

7.2.2.19 Port E drive strength enable from Core (core_pedse_t2)

This single bit output from the Core indicates whether all Port E pins will operate with full or reduced drive strength.

7.2.2.20 Port K Input Data to Core (core_pkind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port K.

7.2.2.21 Port K Output Data from Core (core_pkdo[7:0])

This 8-bit wide output from the Core provides the Port K data output to the system port/pad logic for Port K.

7.2.2.22 Port K output buffer enable from Core (core_pkobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port K.

7.2.2.23 Port K input buffer enable from Core (core_pkibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port K.

7.2.2.24 Port K pullup enable from Core (core_pkpue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port K should be enabled for all Port K pins.

7.2.2.25 Port K drive strength enable from Core (core_pkdse_t2)

This single bit output from the Core indicates whether all Port K pins will operate with full or reduced drive strength.

7.2.3 Clock and Reset Signals

Please see **Section 8** of this guide.

7.2.4 Vector Request/Acknowledge Signals

These descriptions apply to signals that provide for vector requesting to and corresponding acknowledgment from the Core.

7.2.4.1 CPU vector fetch (core_vector_fetch_t4)

This Core output signal indicates that the CPU is executing a vector fetch as a result of a reset or interrupt sequence.

7.2.4.2 System level reset vector request (peri_rstv_request)

This input signal indicates to the Core that the system is requesting the external reset vector from the Core.

7.2.4.3 System level Crystal Monitor reset vector request (peri_xmonv_request)

This input signal indicates to the Core that the system is requesting the Crystal Monitor reset vector from the Core.

7.2.4.4 System level COP Watchdog reset vector request (peri_copv_request)

This input signal indicates to the Core that the system is requesting the COP Watchdog reset vector from the Core.

7.2.5 Stop and Wait Mode Control/Status Signals

Please see **Section 8** of this guide.

7.2.6 Background Debug Mode (BDM) Interface Signals

These descriptions apply to the Core BDM sub-block interface with the system BKGD pad logic. Please see **Section 14** of this guide for further functional details of the BDM.

7.2.6.1 BKGD pin Input Data to Core (bkgd_ind)

This single bit input to the Core provides the Core with the input data from the system port/pad logic for BDM BKGD pin.

7.2.6.2 BKGD pin Output Data from Core (core_bkgd_dout_t4)

This single bit output from the Core provides the BKGD pin data output to the system port/pad logic for the BDM BKGD pin.

7.2.6.3 BKGD pin output buffer enable from Core (core_bkgd_obe)

This single bit output from the Core provides the output buffer enable signal to the system port/pad logic for the BDM BKGD pin.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

7.2.6.4 BKGD pin input buffer enable from Core (core_bkgd_ibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for the BDM BKGD pin.

7.2.6.5 BKGD pin pullup enable from Core (core_bkgdpue_t2)

This single bit output from the Core indicates that the pullup device within the system port/pad logic for the BKGD pin should be enabled for the BKGD pin.

7.2.7 Memory Configuration Signals

These input signals to the Core establish the system memory configuration. Each of these signals is to be tied off to the appropriate logic state at integration of the Core into the SoC design in order to configure the Core memory partitioning according to the needs of the system. Please consult the **STAR12 V1.5 Core Integration Guide** for details on defining the states of these signals.

7.2.8 Scan Control Interface Signals

These descriptions apply to the Core Scan test control signals.

7.2.8.1 Scan mode enable(ipt_scan_mode)

This single bit input indicates to the Core that the system is in Scan test mode and all logic within the Core that needs special conditions for Scan test mode will be handled appropriately.

7.3 Interface Operation

The subsections below give general descriptions of basic read and write operations of the Core. These operations include interfacing with system peripheral registers, on-chip memory registers and array elements, internal Core registers and external bus interface. For more detailed descriptions and timing information please consult the **STAR12 V1.5 Core Integration Guide**.

7.3.1 Read Operations

All read data coming into the Core is implemented by multiplexing the various input read data buses (*peri_rdb_L12[15:0], ram_rdb_L12[15:0], ee_rdb_L12[15:0]* and *fee_rdb_L12[15:0]*) onto the main internal Core read data bus. The active input read data bus is defined by the select signal that is active during the Core read cycle. The subsections below briefly discuss each of peripheral, on-chip memory register and array element and internal core register reads. In each of the figures used in these subsections, the read sequences are separated by write sequences to better illustrate the timing edges.

7.3.1.1 Peripheral Reads

The Core supports both 8-bit and 16-bit reads of peripheral registers. The timing relationship for a basic 8-bit read of a peripheral register is shown in **Figure 7-2** and that of a basic 16-bit read in **Figure 7-3**.

The Core clock (*peri_clk24*) provides the timing reference within the Core for all data transfers with the peripherals. The peripheral clock (*peri_clk34*) is the timing reference for all peripherals within the system tied to the I.P. Bus.

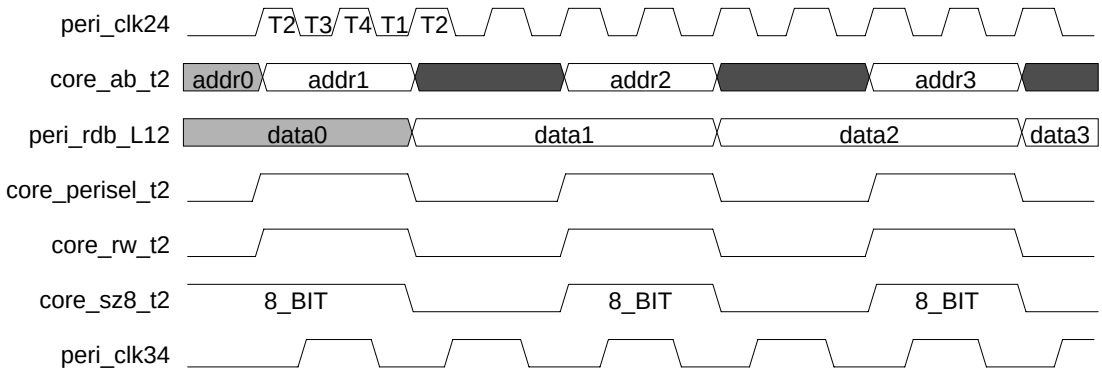


Figure 7-2 Basic 8-bit Peripheral Read Timing

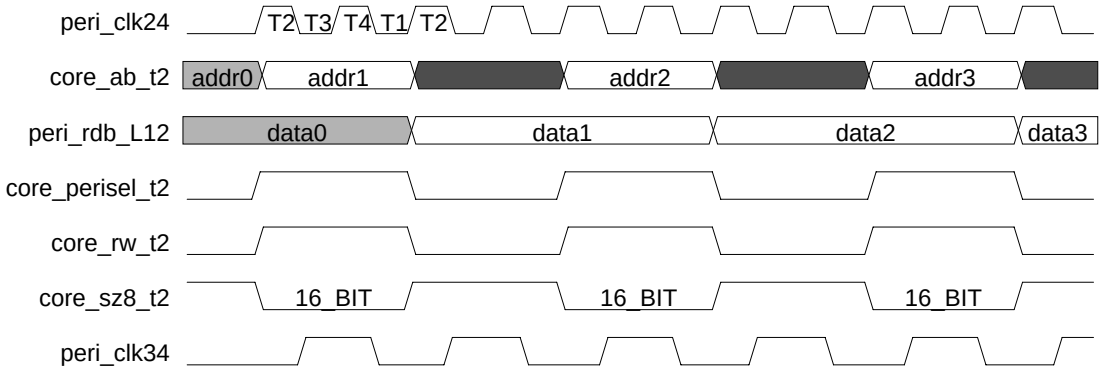


Figure 7-3 Basic 16-bit Peripheral Read Timing

7.3.1.2 Memory Reads

The timing relationship for a basic 8-bit read of a on-chip memory register or array byte by the Core is shown in below in **Figure 7-4** and that of a basic 16-bit read in **Figure 7-5**. In the diagrams, the *MEM_rdb_L12* signal represents any of the on-chip memory read data bus signals (*ram_rdb_L12*, *ee_rdb_L12* or *fee_rdb_L12*) and *core_MSEL_t2* represents any of the on-chip memory register or array selects (such as *core_ramregsel_t2* or *core_ramarraysel_t2* for the RAM and likewise for the EEPROM and Flash EEPROM).

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

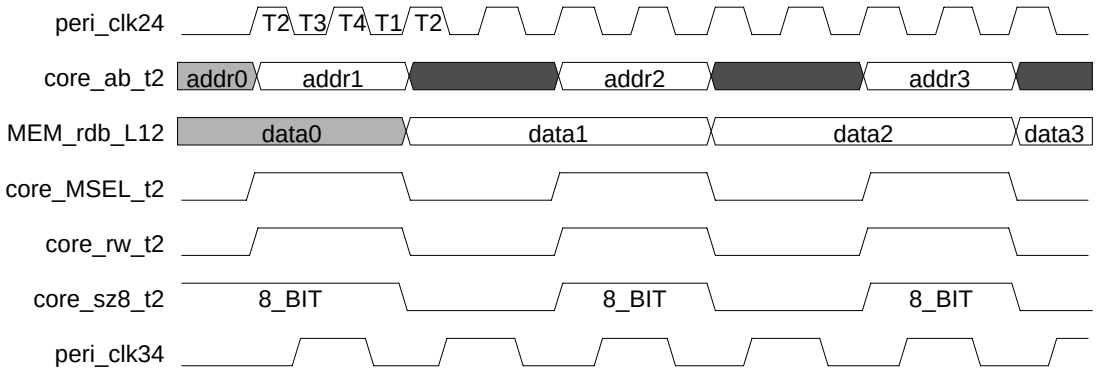


Figure 7-4 Basic 8-bit Memory Read Timing

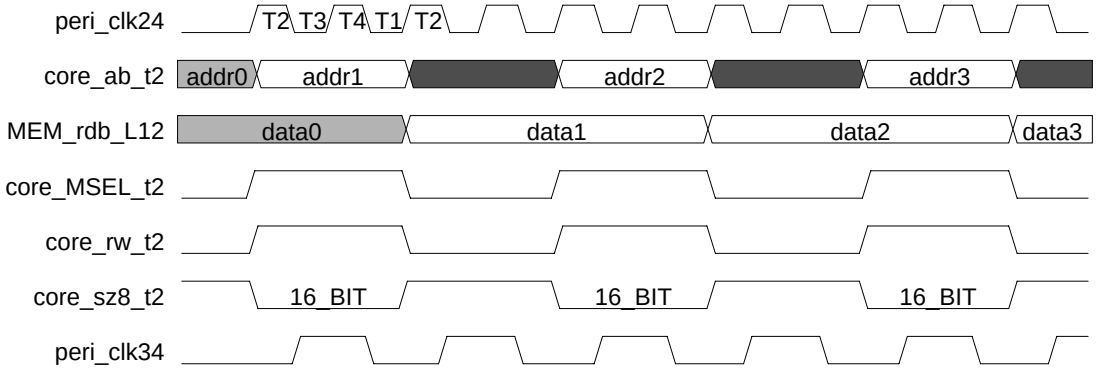


Figure 7-5 Basic 16-bit Memory Read Timing

7.3.1.3 Internal Core Register Reads

The timing for basic 8-bit and 16-bit reads of internal Core registers are shown in **Figure 7-6** and **Figure 7-7**, respectively.

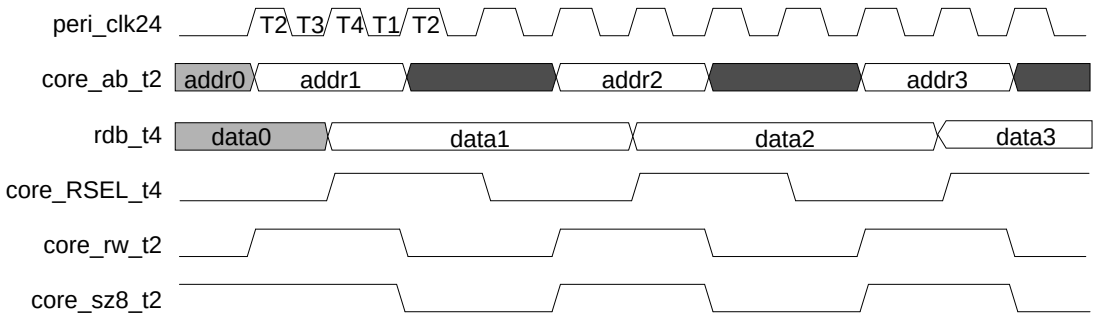


Figure 7-6 Basic 8-bit Core Register Read Timing

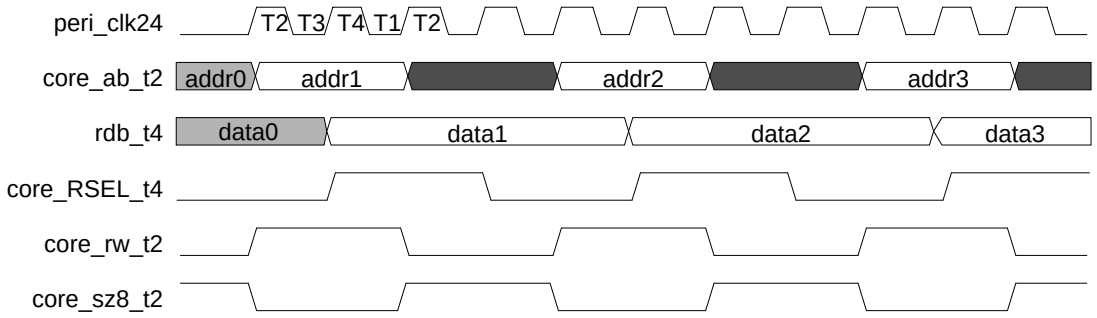


Figure 7-7 Basic 16-bit Core Register Read Timing

7.3.2 Write Operations

All write data exits the Core via the Core write data bus (*core_wdb_t4[15:0]*). The subsections below briefly discuss each of peripheral, on-chip memory register and array element and internal core register writes. In each of the figures used in these subsections, the write sequences are separated by read sequences to better illustrate the timing edges.

7.3.2.1 Peripheral Writes

The Core supports both 8-bit and 16-bit writes of peripheral registers. The timing relationship for a basic 8-bit write of a peripheral register is shown in **Figure 7-8** and that of a basic 16-bit write in **Figure 7-9**. An example of the I.P. Bus read data bus timing is provided in the figures for further illustration purposes.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

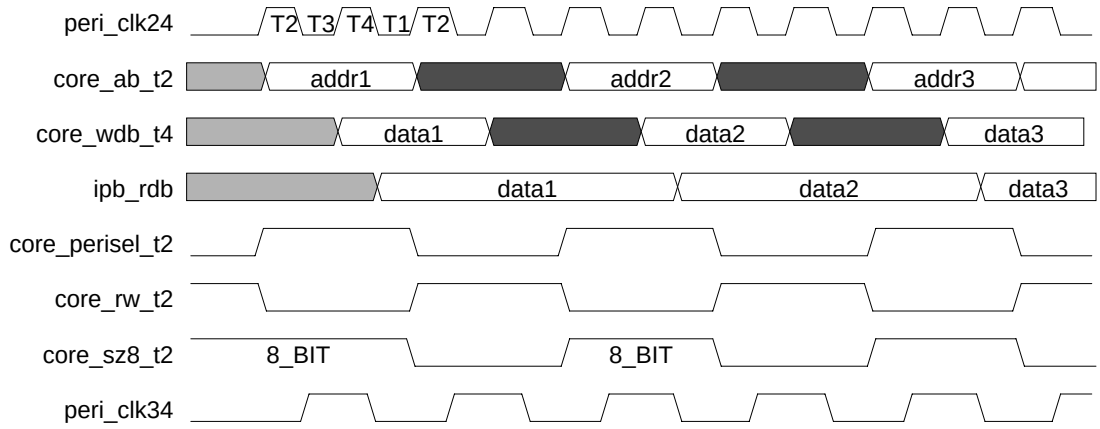


Figure 7-8 Basic 8-bit Peripheral Write Timing

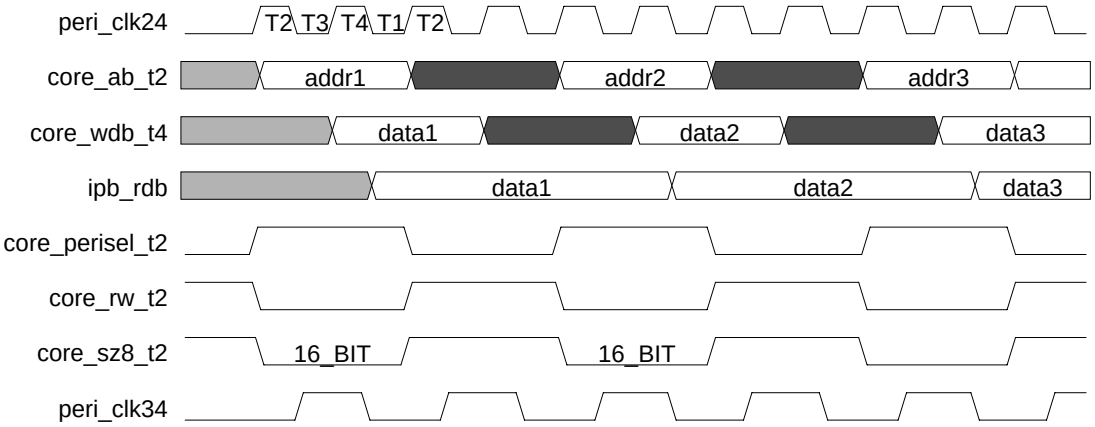


Figure 7-9 Basic 16-bit Peripheral Write Timing

7.3.2.2 Memory Writes

The timing relationship for a basic 8-bit write of a on-chip memory register or array byte by the Core is shown in below in **Figure 7-10** and that of a basic 16-bit write in **Figure 7-11**. As before, the *MEM_rdb_L12* signal represents any of the on-chip memory read data bus signals (*ram_rdb_L12*, *ee_rdb_L12* or *fee_rdb_L12*) and *core_MSEL_t2* represents any of the on-chip memory register or array selects (such as *core_ramregsel_t2* or *core_ramarraysel_t2* for the RAM and likewise for the EEPROM and Flash EEPROM).

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

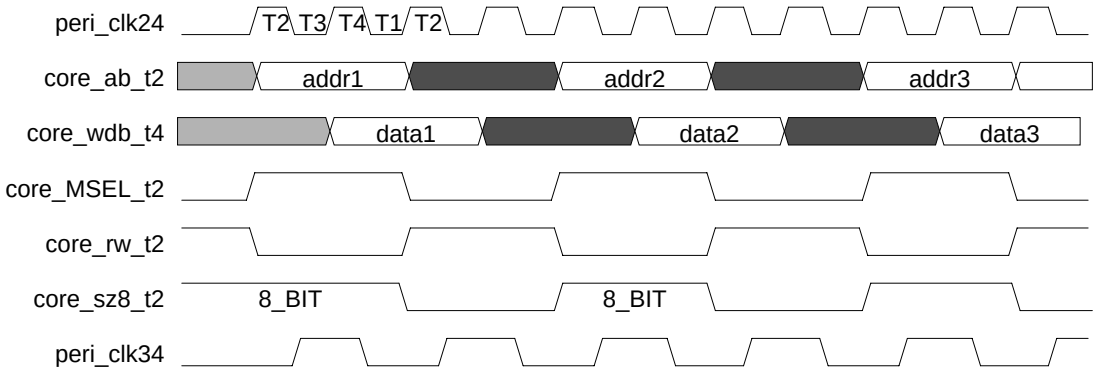


Figure 7-10 Basic 8-bit Memory Write Timing

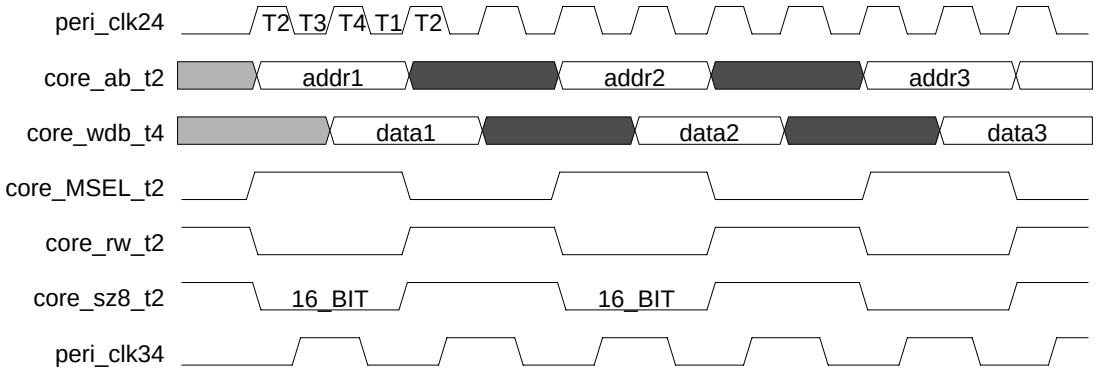


Figure 7-11 Basic 16-bit Memory Write Timing

7.3.2.3 Internal Core Register Writes

The timing for basic 8-bit and 16-bit writes of internal Core registers are shown in **Figure 7-12** and **Figure 7-13**, respectively.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

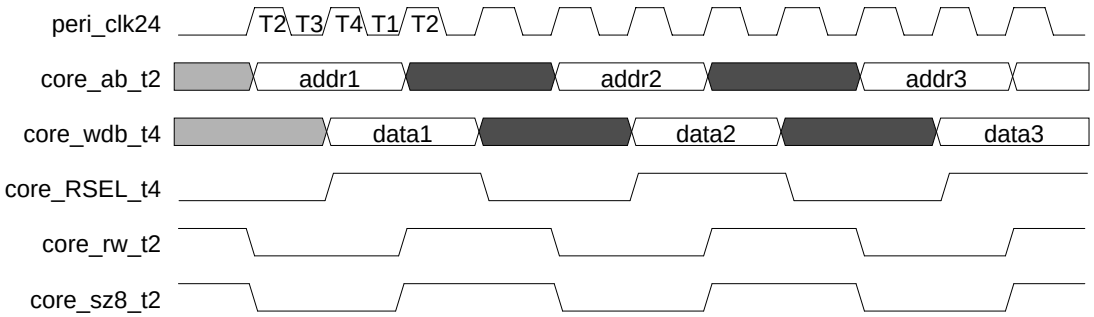


Figure 7-12 Basic 8-bit Core Register Write Timing

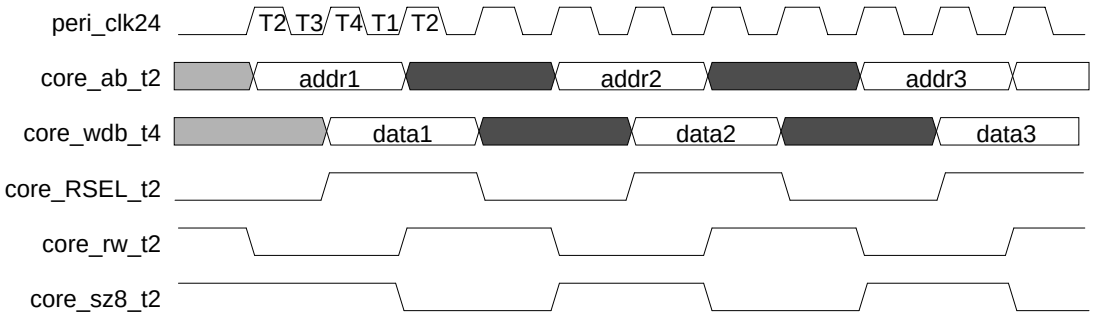


Figure 7-13 Basic 16-bit Core Register Write Timing

7.3.3 Multiplexed External Bus Interface

A timing diagram of the multiplexed external bus is shown in . Major bus signals are included in the diagram. While both a data write and data read cycle are shown, only one would occur on a particular bus cycle. **Table 7-2** gives the preliminary timing characteristics for the signals illustrated in .

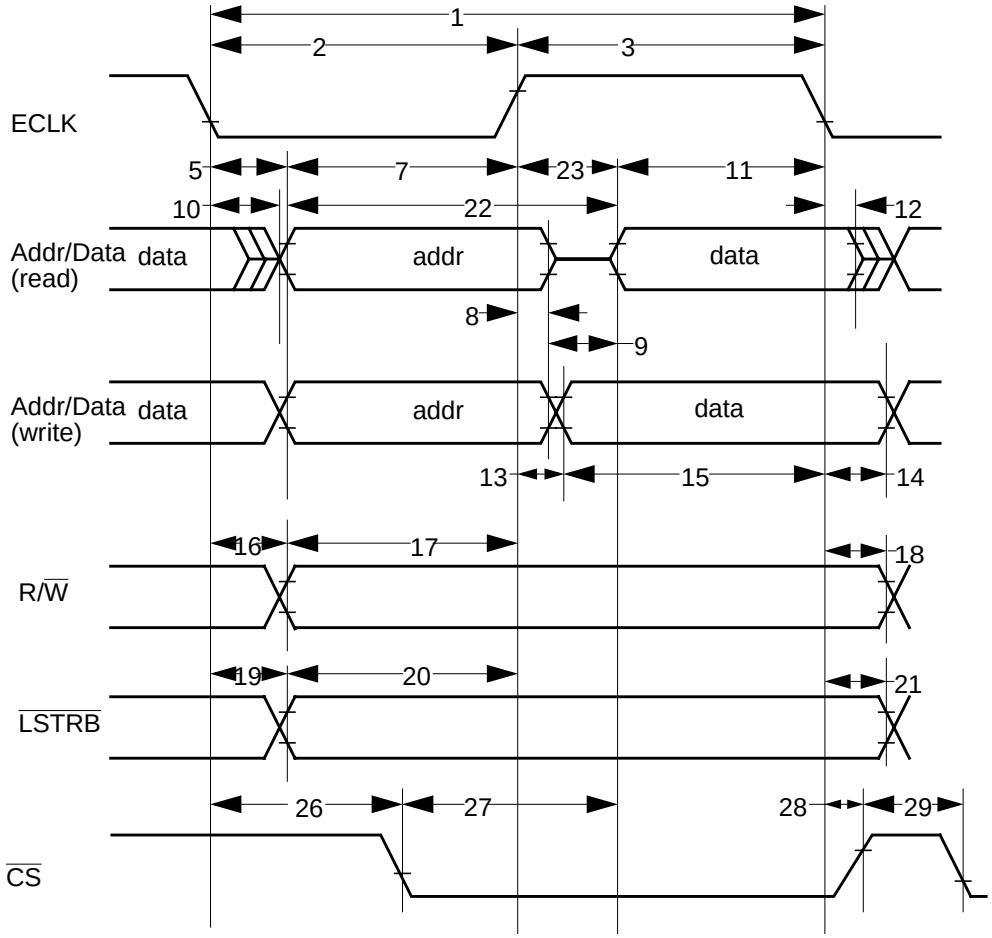


Figure 7-14 General External Bus Timing

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 7-2 Multiplexed Expansion Bus Timing - Preliminary Targets

Num	Characteristic [1 2 3]	Symbol	16 MHz		20 MHz		25 MHz		Unit
			Min	Max	Min	Max	Min	Max	
	Frequency of operation (E-clock)	f_o	D.C.	16.0	D.C.	20.0	D.C.	25.0	MHz
1	Cycle time	t_{cyc}	62		50		40		ns
2	Pulse width, E low	PW_{EL}	28		22		18		ns
3	Pulse width, E high[4]	PW_{EH}	28		22		18		ns
5	Address delay time	t_{AD}		12		10		8	ns
6	n/a	n/a							ns
7	Address valid time to E rise (PW_{EL}-T_{AD})	t_{AV}	16		12		10		ns
8	Muxed address hold time	t_{MAH}	2		2		1		ns
9	Address hold to data valid	t_{AHDS}	4		3		2		ns
10	Data hold to address	t_{DHA}		5		4		3	ns
11	Read data setup time	t_{DSR}	14		10		8		ns
12	Read data hold time	t_{DHR}	0		0		0		ns
13	Write data delay time	t_{DDW}		12		10		8	ns
14	Write data hold time	t_{DHW}	2		2		1		ns
15	Write data setup time[4] (PW_{EH}-t_{DDW})	t_{DSW}	16		12		10		ns
16	Read/write delay time	t_{RWD}		12		10		8	ns
17	Read/write valid time to E rise (PW_{EL}-t_{RWD})	t_{RWV}	16		12		10		ns
18	Read/write hold time	t_{RWH}	2		2		1		ns
19	Low strobe delay time	t_{LSD}		12		10		8	ns
20	Low strobe valid time to E rise (PW_{EL}-t_{LSD})	t_{LSV}	16		12		10		ns
21	Low strobe hold time	t_{LSH}	2		2		1		ns
22	Address access time[4] (t_{cyc}-t_{AD}-t_{DSR})	t_{ACCA}	36		30		24		ns
23	E high access time[4] (PW_{EH}-t_{DSR})	t_{ACCE}	14		12		10		ns
26	Chip select delay time	t_{CSD}		22		18		15	ns
27	Chip select access time[4] (t_{cyc}-t_{CSD}-t_{DSR})	t_{ACCS}	26		22		17		ns
28	Chip select hold time	t_{CSH}		1		1		1	ns
29	Chip select negated time	t_{CSN}	12		10		8		ns

NOTES:
1. Crystal input is required to be within 45% to 55% duty.
2. Reduced drive must be off to meet these timings.
3. Unequal loading of pins will affect relative timing numbers.
4. Affected by clock stretch: add N x t_{cyc} where N=0,1,2 or 3, depending on the number of clock stretches.

7.3.4 General Internal Read Visibility Timing

Internal writes have the same timing as external writes. Internal read visibility is shown in **Figure 7-15** and **Table 7-3** shows the associated timing numbers.

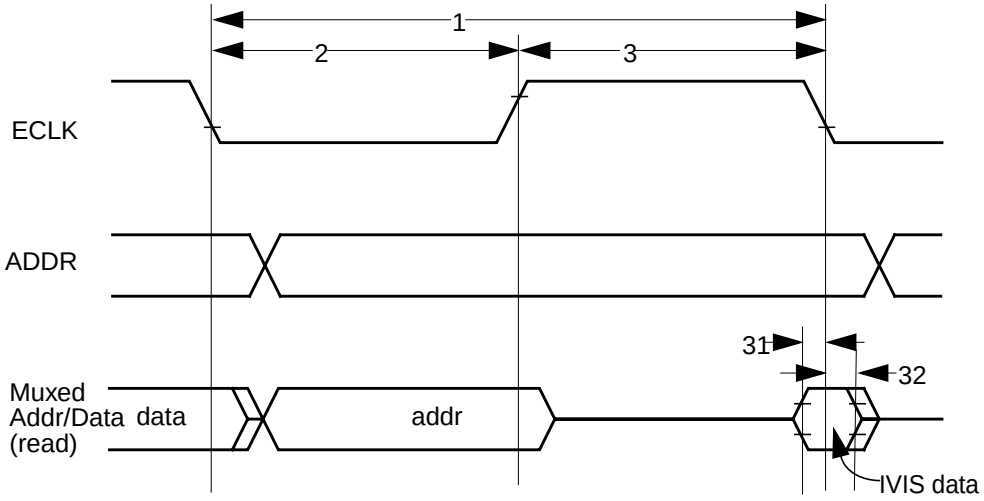


Figure 7-15 General Internal Read Visibility Timing

Table 7-3 Expansion Bus Timing - Preliminary Targets

Num	Characteristic [1 2 3]	Symbol	16 MHz		20 MHz		25 MHz		Unit
			Min	Max	Min	Max	Min	Max	
	Frequency of operation (E-clock)	f_o	D.C.	16.0	D.C.	20.0	D.C.	25.0	MHz
1	Cycle time	t_{cyc}	62		50		40		ns
2	Pulse width, E low	PW_{EL}	28		22		18		ns
3	Pulse width, E high[4]	PW_{EH}	28		22		18		ns
31RG	IVIS read data set-up time - Registers		11		5		2		ns
31RM	IVIS read data set-up time - RAM		11		5		2		ns
31EE	IVIS read data set-up time - EEPROM		11		5		2		ns
31FL	IVIS read data set-up time - FLASH[5]		6		0		0		ns

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 7-3 Expansion Bus Timing - Preliminary Targets

Num	Characteristic [1 2 3]	Symbol	16 MHz		20 MHz		25 MHz		Unit
32	IVIS read data hold time (all)		2		2		1		ns

NOTES:
1. Crystal input is required to be within 45% to 55% duty.
2. Reduced drive must be off to meet these timings.
3. Unequal loading of pins will affect relative timing numbers.
4. Affected by clock stretch: add N x t_{cyc} where N=0,1,2 or 3, depending on the number of clock stretches.
5. Timing is tighter than other memories due to larger array size.

7.3.5 Detecting Access Type from External Signals

The external signals $\overline{\text{LSTRB}}$, R/$\overline{\text{W}}$, and A0 indicate the type of bus access that is taking place. Accesses to the internal RAM are the only type of access that would produce $\overline{\text{LSTRB}}$=A0=1 because the internal RAM is specifically designed to allow misaligned 16-bit accesses in a single cycle. In these cases, the data for the address that was accessed is on the low half of the data bus and the data for address+1 is on the high half of the data bus. This operation only occurs when internal visibility is on.

Table 7-4 shows the relationship between these signals and the type of access.

Table 7-4 Access Type vs. Bus Control Pins

$\overline{\text{LSTRB}}$	A0	R/$\overline{\text{W}}$	Type of Access
1	0	1	8-bit read of an even address
0	1	1	8-bit read of an odd address
1	0	0	8-bit write of an even address
0	1	0	8-bit write of an odd address
0	0	1	16-bit read of an even address
1	1	1	16-bit read of an odd address (low/high data swapped)
0	0	0	16-bit write to an even address
1	1	0	16-bit write to an odd address (low/high data swapped)

Section 8 Core Clock and Reset Connections

This section details the STAR12 V1.5 Core external clock connections. In addition, this section will discuss the reset timing needs of the Core since this is associated very closely with the external clocking requirements.

8.1 Clocking Overview

The STAR12 V1.5 Core is implemented as a single clock source design with complete Mux-D scan test implementation. Since the Core is compatible with the feature set of the MHC12 microcontroller product family, many signal and timing requirements exist for the system clock and reset generation block(s) to support these features. Many of these requirements are driven by the interaction of the Core with the clock and reset generation block(s) in the system due to CPU wait and stop mode functionality and the various time based reset and interrupt functions (such as Crystal Monitor and COP Watchdog resets and Real Time Interrupt functions) available on the STAR12 family of products. A diagram of the Core interface signals is given in **Figure 8-1** below.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

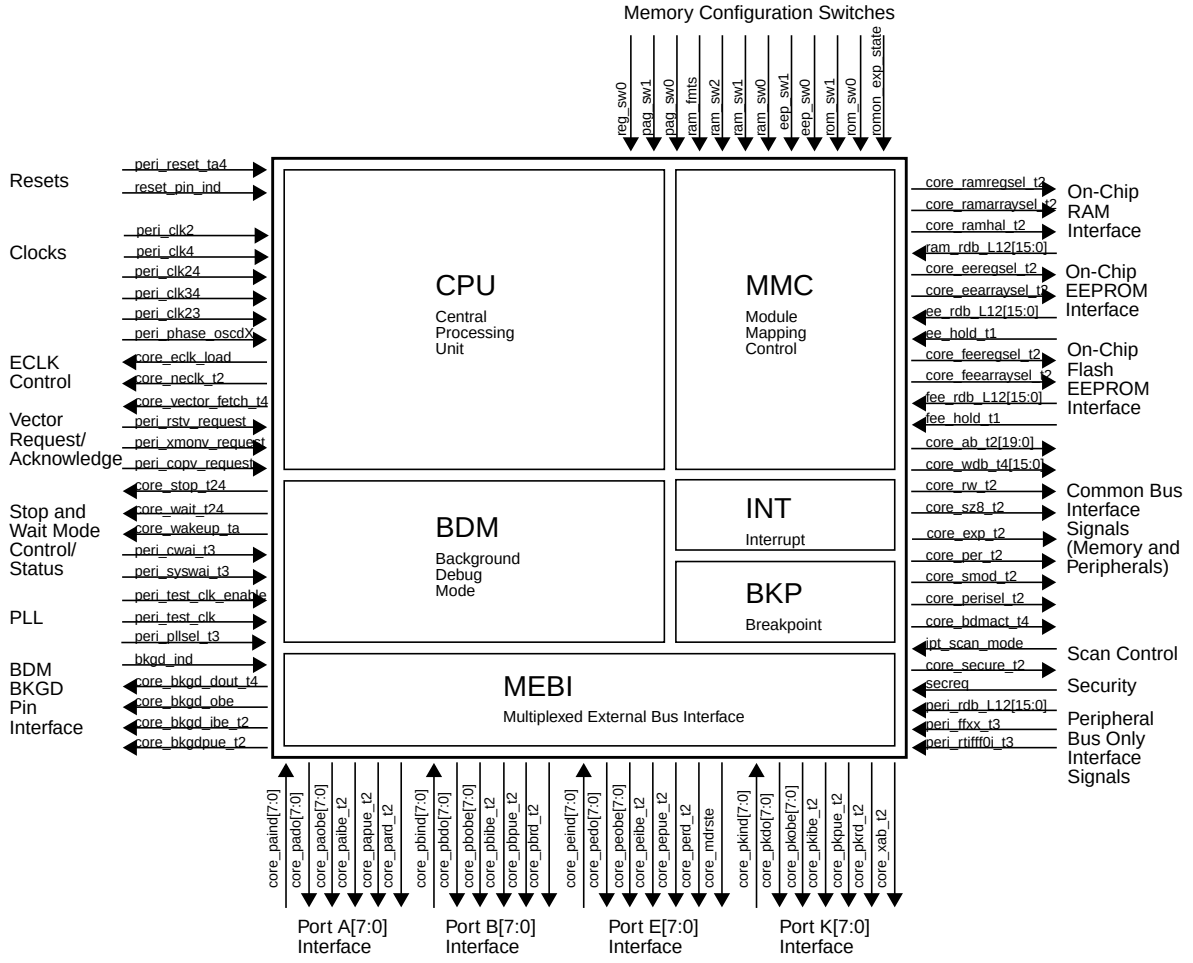


Figure 8-1 Core Interface Signals

The Core interfaces with the system clock and reset generation block(s) in order to synchronize the actions of the STAR12 CPU with the rest of the system. Through the interface signals, the Core supports the use of a system Phase-Locked Loop (PLL), Crystal Monitor, COP Watchdog and Real Time Interrupt as well as clocking options during CPU wait and stop modes. Each of these aspects are discussed in the subsections that follow.

8.1.1 Basic Clock Relationship

The basic system clock timing in shown in **Figure 8-2** below. The system clock generation block is required to provide the main Core clocks (*peri_clk24*, *peri_clk2*, *and peri_clk4*), the main peripheral clock (*peri_clk34*) and the system clk23 (*peri_clk23*) to the Core (the Core uses *peri_clk23* to generate the ECLK signal). The method of clock generation (i.e. crystal, PLL, etc.) is left up to the system integrator as long as the clocks provided meet the phase relationship shown in the figure.

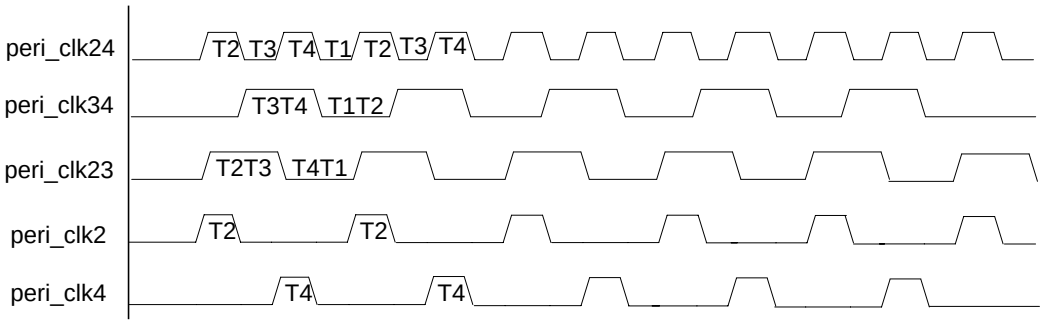


Figure 8-2 System Clock Timing Diagram

The remaining clock input to the Core, *peri_phase_oscdX*, is the same frequency as the *peri_clk34* as derived directly from the oscillator. When using the PLL for the system clocks, the BDM sub-block must maintain a constant rate clock and cannot depend upon the use of the PLL generated clock. Because of this, this signal operates at the same frequency as *peri_clk34* prior to engaging the PLL (or as derived directly from the oscillator). Once the PLL is engaged, this clock must maintain the pre-PLL frequency in order to keep the BDM synchronized.

8.1.2 Reset Relationship

The Core depends upon the use of two input signals, *reset_pin_ind* and *peri_reset_ta4*, for controlling the reset conditions of all logic within the Core. The active low *reset_pin_ind* signal timing follows that of the physical system reset pin indicating immediately when a system reset is requested (for example when the $\overline{\text{RESET}}$ pin is pulled low externally). This signal is used as a load enable on the MODE pins of the MEBI sub-block to ensure that the Core mode of operation is known and set up immediately upon a system reset request. The *peri_reset_ta4* signal will generally be asserted (logic 1) asynchronously by the reset generation block at the time that a system reset is requested. Further, the assumption is that this signal will stay asserted until such time that the clock generation block has determined that the clocks to the Core are stable and that the Core should proceed with a system reset sequence.

8.1.3 Phase-Locked Loop Interface

The Core allows for the implementation of a on-chip Phase-Locked Loop (PLL) and interacts with it through the *peri_pllsel_t3*, *peri_test_clk_enable* and *peri_test_clk* input signals. If a PLL is implemented, the Core assumes it will operate on the peripheral clock (*peri_clk34*) and thus the *peri_pllsel_t3* signal must be asserted (logic 1) on the phase three rising edge of this clock when the PLL is first engaged and to be negated (logic 0) when the PLL is disabled. The *peri_test_clk* and *peri_test_clk_enable* signals are provided in order to facilitate test features for the PLL. When the *peri_test_clk_enable* signal is asserted (logic 1), the Core will register the signal on the phase four rising edge of *peri_clk24* and will then output the clock signal being input on *peri_test_clk* directly on Port E Bit 6 of the system. This test feature is only valid in Special modes and setting of the PIPOE bit in the PEAR register overrides the clock output.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

8.1.4 STAR12 CPU Wait and Stop Modes

The Core inputs *peri_cwai_t3* and *peri_syswai_t3* indicate to the Core what the state of the system clocks will be during CPU wait mode with the former reflecting the Core clock (*peri_clk24*) state and the latter the state of all system clocks. These inputs typically come from the clock and reset generation block(s) and could either be hard-wired to a given logic value or reflect the state of software bits controlling the clock functionality. The Core assumes that the asserted (logic 1) state indicates that the clock(s) will cease during wait mode and that the negated (logic 0) state indicates that the clock(s) will run during wait mode.

The Core will reflect the CPU mode through the state of the *core_wait_t24* and *core_stop_t24* signals. The *core_wait_t24* or *core_stop_t24* signal will assert when the CPU executes a WAI or STOP instruction, respectively, and both will remain negated (logic 0) during normal operation. In the case of exit from either wait or stop mode due to a valid interrupt, the *core_wakeup_ta* signal will assert (logic 1) asynchronously upon receiving the valid interrupt request. This signal will then negate (logic 0) asynchronously once the interrupt source is negated (indicating that the interrupt has been serviced and is no longer being requested).

8.2 Signal Summary

Each of the Core I/O signals that interface with the system clock and reset generation block(s) are listed in **Table 8-1** below with the signal type and a brief functional description for completeness.

Table 8-1 Core Clock and Reset Interface Signals

Signal Name	Type	Functional Description
Clock and Reset Signals		
peri_reset_ta4	I	System reset signal
reset_pin_ind	I	System level reset pin input data
peri_clk2	I	System clock clk2 for Core
peri_clk4	I	System clock clk4 for Core
peri_clk24	I	System clock clk24 for Core
peri_clk34	I	System clock clk34 for peripherals on I.P. Bus Interface
peri_clk23	I	System clock clk23 used by Core to generate ECLK
peri_phase_oscdX	I	Oscillator Clock divided by 'X'
peri_test_clk_enable	I	PLL test feature clock enable signal
peri_test_clk	I	PLL test feature clock signal
peri_pllsel_t3	I	PLL selected signal
core_eclk_load	O	External clock load enable signal
core_neclk_t2	O	External clock disable signal
Stop and Wait Mode Control/Status Signals		
core_stop_t24	O	Core CPU stop mode signal
core_wait_t24	O	Core CPU wait mode signal
core_wakeup_ta	O	Core wakeup from stop or wait mode due to interrupt
peri_cwai_t3	I	Core wait signal: controls whether clk24 runs during CPU wait mode. 0 - clk24 runs during wait, 1 - clk24 ceases during wait.
peri_syswai_t3	I	System level wait signal: controls whether system clocks run during CPU wait mode. 0 - all clocks run during wait, 1 - no clocks run during wait.

8.3 Detailed Clock and Reset Signal Descriptions

General descriptions of the Core clock and reset interface signals are given in the subsections below. Also included are the stop and wait mode signals due to the necessary interaction with the clock and reset requirements. For detailed descriptions of these signals including timing information please consult the **STAR12 V1.5 Core Integration Guide**.

8.3.1 Clock and Reset Signals

These descriptions apply to system level clock and reset signals needed by the Core.

8.3.1.1 System Reset signal (peri_reset_ta4)

This single bit asynchronous input to the Core indicates the system reset condition.

8.3.1.2 System level reset input data (reset_pin_ind)

This active-low single bit input is used within the Core as a load enable for the MODE pin logic on Port E of the system.

8.3.1.3 System level clock for the Core (peri_clk2)

This clock input is one of the main clocks for the Core.

8.3.1.4 System level clock for the Core (peri_clk4)

This clock input is one of the main clocks for the Core.

8.3.1.5 System level clock for the Core (peri_clk24)

This clock input is one of the main clocks for the Core.

8.3.1.6 System level clock for peripheral blocks (peri_clk34)

This clock input is the main clock source for all peripheral blocks integrated in the system and accessed by the Core through the I.P. Bus Interface.

8.3.1.7 System ECLK clock (peri_clk23)

This clock input is the main clock source used by the Core to generate the system ECLK.

8.3.1.8 Divided Down System Oscillator Clock (peri_phase_oscdX)

This clock input to the Core is used within the Core by the Background Debug Mode sub-block to keep the BDM synchronized.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

8.3.1.9 System Test Clock enable (peri_test_clk_enable)

This single bit input to the Core indicates that the phase-locked loop (PLL) test clock should be output on the system Port E bit 6 pin when the PIPOE bit is zero.

8.3.1.10 System Test Clock (peri_test_clk)

This clock input to the Core is the PLL test clock.

8.3.1.11 System clock source select signal (peri_pllsel_t3)

This single bit input to the Core indicates whether clocks within the system are derived from the crystal or PLL.

8.3.1.12 ECLK load enable signal(core_eclk_load)

This single bit output from the Core is the load enable signal for the system external clock, ECLK.

8.3.1.13 ECLK disable signal(core_neclk_t2)

This single bit output from the Core is the disable signal for the system external clock, ECLK.

8.3.2 Stop and Wait Mode Control/Status Signals

These descriptions apply to signals that provide for controlling some of the functionality and status indication of CPU stop and wait modes.

8.3.2.1 CPU stop mode indicator (core_stop_t24)

This Core output signal indicates whether the CPU is in stop mode.

8.3.2.2 CPU wait mode indicator (core_wait_t24)

This Core output signal indicates whether the CPU is in wait mode.

8.3.2.3 Core wakeup indicator for wait and stop mode (core_wakeup_ta)

This asynchronous Core output signal indicates that the CPU has received an interrupt request and is ready to resume normal operation.

8.3.2.4 Core wait signal from system clock generation block (peri_cwai_t3)

This Core input signal indicates to the CPU whether the main Core clock, *peri_clk24*, will run during CPU wait mode.

8.3.2.5 System level wait signal (peri_syswai_t3)

This Core input signal indicates to the Core whether all system clocks will run during CPU wait mode.

Section 9 Core Power Connections

This section details the STAR12 V1.5 Core power connections.

9.1 Power Overview

The STAR12 V1.5 Core operates from a single power and a single ground connection.

9.1.1 Power and Ground Summary

The Core requires a single power (typically termed VDD) and a single ground (typically termed VSS) connection that is implicit when integrating into a synthesized design. There are no signals at the Core interface for power and ground.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 10 Interrupt (INT)

This section describes the functionality of the Interrupt (INT) sub-block of the Core.

10.1 Overview

The Interrupt sub-block decodes the priority of all system exception requests and provides the applicable vector for processing the exception. The INT supports I-bit maskable and X-bit maskable interrupts, a nonmaskable Unimplemented Opcode Trap, a nonmaskable software interrupt (SWI) or Background Debug Mode request, and three system reset vector requests. All interrupt related exception requests are handled by the Interrupt.

10.1.1 Features

- Provides 2 to 122 I bit maskable interrupt vectors ($FF00-$FFF2)
- Provides 1 X bit maskable interrupt vector ($FFF4)
- Provides a nonmaskable Unimplemented Opcode Trap (TRAP) vector ($FFF8)
- Provides a nonmaskable software interrupt (SWI) or Background Debug Mode request vector ($FFF6)
- Provides 3 system reset vectors ($FFFA-$FFFE)
- Determines the appropriate vector and drives it onto the address bus at the appropriate time
- Signals the CPU that interrupts are pending
- Provides control registers which allow testing of interrupts
- Provides additional input signals which prevents requests for servicing I and X interrupts
- Wakes the system from stop or wait mode when an appropriate interrupt occurs or whenever $\overline{\text{XIRQ}}$ is active, even if $\overline{\text{XIRQ}}$ is masked
- Provides asynchronous path for all I and X interrupts, ($FF00–$FFF4)
- (Optional) Selects and stores the highest priority I interrupt based on the value written into the HPRIO register

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

10.1.2 Block Diagram

A block diagram of the Interrupt sub-block is shown in **Figure 10-1** below.

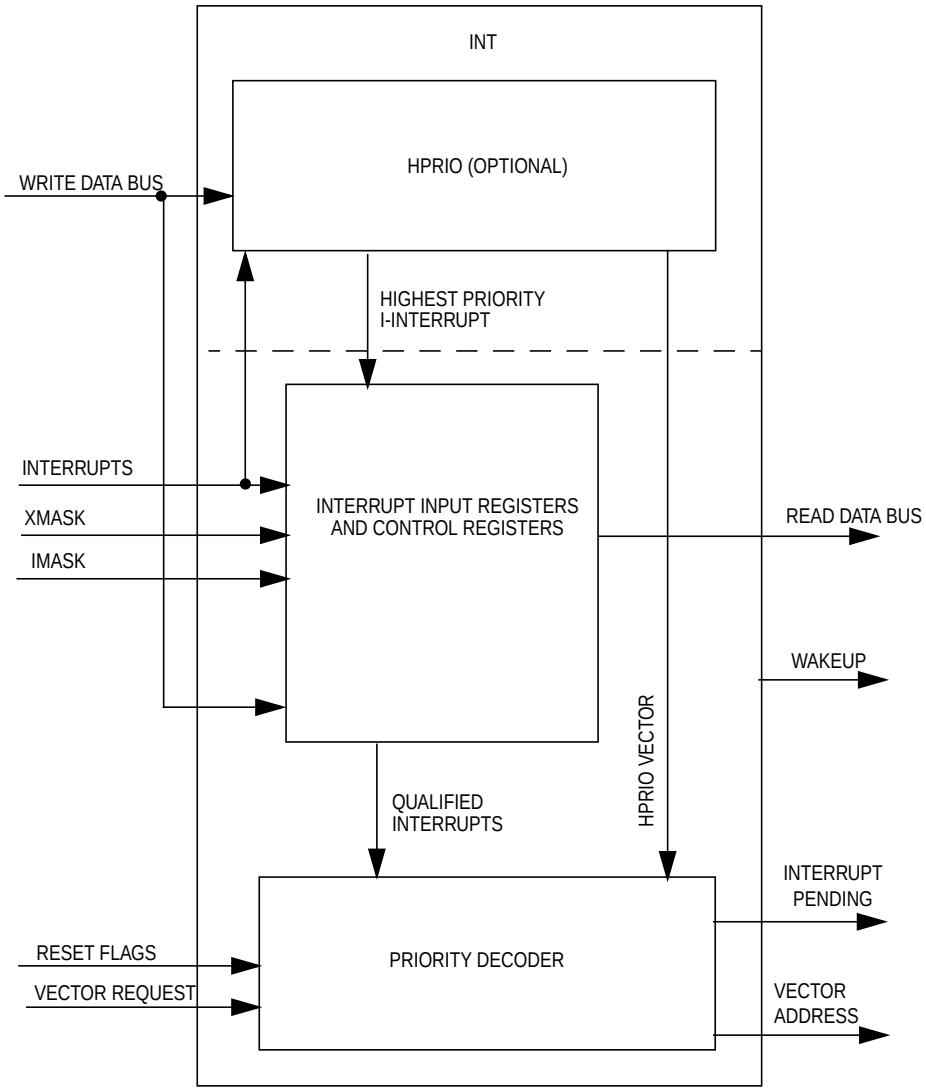


Figure 10-1 Interrupt Block Diagram

10.2 Interface Signals

All interfacing with the Interrupt sub-block is done within the Core. The Interrupt does however receive direct input from the Multiplexed External Bus Interface (MEBI) sub-block of the Core for the $\overline{IRQ}$ and $\overline{XIRQ}$ pin data.

10.3 Registers

A summary of the registers associated with the Interrupt sub-block is shown in **Figure 10-2** below. Detailed descriptions of the registers and associated bits are given in the subsections that follow.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$0015	ITCR	read	0	0	0	WRTINT	ADR3	ADR2	ADR1	ADR0
		write								
$0016	ITEST	read	INTE	INTC	INTA	INT8	INT6	INT4	INT2	INT0
		write								
$001F	HPRIO	read	PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0
		write								

= Unimplemented X = Indeterminate

Figure 10-2 Interrupt Register Summary

10.3.1 Interrupt Test Control Register

Address:$0015

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	WRTINT	ADR3	ADR2	ADR1	ADR0
Write:								
Reset:	0	0	0	0	1	1	1	1

Figure 10-3 Interrupt Test Control Register (ITCR)

Read: see individual bit descriptions
Write: see individual bit descriptions

WRTINT - Write to the Interrupt Test Registers

Read: anytime

Write: only in special modes and with I bit mask and X bit mask set.

1 = Disconnect the interrupt inputs from the priority decoder and use the values written into the ITEST registers instead.

0 = Disables writes to the test registers; reads of the test registers will return the state of the interrupt inputs.

NOTE: *Any interrupts which are pending at the time that WRTINT is set will remain until they are overwritten.*

ADR3 - ADR0 - Test register select bits

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Read: anytime

Write: anytime

These bits determine which test register is selected on a read or write. The hexadecimal value written here will be the same as the upper nibble of the lower byte of the vector selects. That is, an “F” written into ADR3 - ADR0 will select vectors $FFFE - $FFF0 while a “7” written to ADR3 - ADR0 will select vectors $FF7E - $FF70.

10.3.2 Interrupt Test Registers

Address:$0016

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	INTE	INTC	INTA	INT8	INT6	INT4	INT2	INT0
Reset:	0	0	0	0	0	0	0	0

Figure 10-4 Interrupt TEST Registers (ITEST)

Read: Only in special modes. Reads will return either the state of the interrupt inputs of the Interrupt sub-block (WRTINT = 0) or the values written into the TEST registers (WRTINT = 1). Reads will always return zeroes in normal modes.

Write: Only in special modes and with WRTINT = 1 and CCR I mask = 1.

INTE - INT0 - Interrupt TEST bits

These registers are used in special modes for testing the interrupt logic and priority independent of the system configuration. Each bit is used to force a specific interrupt vector by writing it to a logic one state. Bits are named with INTE through INT0 to indicate vectors $FFxE through $FFx0. These bits can be written only in special modes and only with the WRTINT bit set (logic one) in the Interrupt Test Control Register (ITCR). In addition, I interrupts must be masked using the I bit in the CCR. In this state, the interrupt input lines to the Interrupt sub-block will be disconnected and interrupt requests will be generated only by this register. These bits can also be read in special modes to view that an interrupt requested by a system block (such as a peripheral block) has reached the INT module.

There is a test register implemented for every 8 interrupts in the overall system. All of the test registers share the same address and are individually selected using the value stored in the ADR3 - ADR0 bits of the Interrupt Test Control Register (ITCR).

NOTE: *When ADR3-ADR0 have the value of $F, only bits 2-0 in the ITEST register will be accessible. That is, vectors higher than $FFF4 cannot be tested using the test registers and bits 7-3 will always read as a logic zero. If ADR3-ADR0 point to an unimplemented test register, writes will have no effect and reads will always return a logic zero value.*

10.3.3 Highest Priority I Interrupt (Optional)

Address:$001F

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	PSEL7	PSEL6	PSEL5	PSEL4	PSEL3	PSEL2	PSEL1	0
Write:								
Reset:	1	1	1	1	0	0	1	0

Figure 10-5 Highest Priority I Interrupt Register (HPRIO)

Read: anytime
Write: only if I mask in CCR = 1

PSEL7 - PSEL1 - Highest priority I interrupt select bits

The state of these bits determines which I bit maskable interrupt will be promoted to highest priority (of the I bit maskable interrupts). To promote an interrupt, the user writes the least significant byte of the associated interrupt vector address to this register. If an unimplemented vector address or a non I bit masked vector address (value higher than $F2) is written, IRQ ($FFF2) will be the default highest priority interrupt.

10.4 Operation

The Interrupt sub-block processes all exception requests made by the CPU. These exceptions include interrupt vector requests and reset vector requests. Each of these exception types and their overall priority level is discussed in the subsections below.

10.4.1 Interrupt Exception Requests

As shown in **Figure 10-1** above, the INT mainly contains a register block to provide interrupt status and control, an optional Highest Priority I Interrupt (HPRIO) block and a priority decoder to evaluate whether pending interrupts are valid and assess their priority.

10.4.1.1 Interrupt Registers

The INT registers are accessible only in special modes of operation and function as described in **10.3.1** and **10.3.2** previously.

10.4.1.2 Highest Priority I bit Maskable Interrupt

When the optional HPRIO block is implemented, the user is allowed to promote a single I bit maskable interrupt to be the highest priority I interrupt. The HPRIO evaluates all interrupt exception requests and passes the HPRIO vector to the priority decoder if the highest priority I interrupt is active.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

10.4.1.3 Interrupt Priority Decoder

The priority decoder evaluates all interrupts pending and determines their validity and priority. When the CPU requests an interrupt vector, the decoder will provide the vector for the highest priority interrupt request. Because the vector is not supplied until the CPU requests it, it is possible that a higher priority interrupt request could override the original exception that caused the CPU to request the vector. In this case, the CPU will receive the highest priority vector and the system will process this exception instead of the original request.

NOTE: *Care must be taken to ensure that all exception requests remain active until the system begins execution of the applicable service routine; otherwise, the exception request may not get processed.*

If for any reason the interrupt source is unknown (e.g. an interrupt request becomes inactive after the interrupt has been recognized but prior to the vector request), the vector address will default to that of the last valid interrupt that existed during the particular interrupt sequence. If the CPU requests an interrupt vector when there has never been a pending interrupt request, the INT will provide the Software Interrupt (SWI) vector address.

10.4.2 Reset Exception Requests

The INT supports three system reset exception request types: normal system reset or power-on-reset request, Crystal Monitor reset request and COP Watchdog reset request. The type of reset exception request must be decoded by the system and the proper request made to the Core. The INT will then provide the service routine address for the type of reset requested.

10.4.3 Exception Priority

The priority (from highest to lowest) and address of all exception vectors issued by the INT upon request by the CPU is shown in **Table 10-1** below.

Table 10-1 Exception Vector Map and Priority

Vector Address	Source
$FFFE–$FFFF	System reset
$FFFC–$FFFD	Crystal Monitor reset
$FFFA–$FFFB	COP reset
$FFF8–$FFF9	Unimplemented opcode trap
$FFF6–$FFF7	Software interrupt instruction (SWI) or BDM vector request
$FFF4–$FFF5	$\overline{\text{XIRQ}}$ signal
$FFF2–$FFF3	$\overline{\text{IRQ}}$ signal
$FFF0–$FF00	Device-specific I bit maskable interrupt sources (priority in descending order)

10.5 Modes of Operation

The functionality of the INT sub-block in various modes of operation is discussed in the subsections that follow.

10.5.1 Normal Operation

The INT operates the same in all normal modes of operation.

10.5.2 Special Operation

Interrupts may be tested in special modes through the use of the interrupt test registers as described in **10.3.1** and **10.3.2** previously.

10.5.3 Emulation Modes

The INT operates the same in emulation modes as in normal modes.

10.6 Low-Power Options

The INT does not contain any user-controlled options for reducing power consumption. The operation of the INT in low-power modes is discussed in the following subsections.

10.6.1 Run Mode

The INT does not contain any options for reducing power in run mode.

10.6.2 Wait Mode

Clocks to the INT can be shut off during system wait mode and the asynchronous interrupt path will be used to generate the wakeup signal upon recognition of a valid interrupt or any $\overline{\text{XIRQ}}$ request.

10.6.3 Stop Mode

Clocks to the INT can be shut off during system stop mode and the asynchronous interrupt path will be used to generate the wakeup signal upon recognition of a valid interrupt or any $\overline{\text{XIRQ}}$ request.

10.7 Motorola Internal Information

The INT does not contain any functionality that is considered to be for Motorola internal use only.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 11 Module Mapping Control (MMC)

This section describes the functionality of the Module Mapping Control (MMC) sub-block of the Core.

11.1 Overview

The Module Mapping Control (MMC) sub-block of the Core performs all mapping and select operations for the on-chip and external memory blocks. The MMC also handles mapping functions for the system peripheral blocks and provides a global peripheral select to be decoded by the Motorola I.P. Bus when the Core is addressing a portion of the peripheral register map space. All bus-related data flow and multiplexing for the Core is handled within the MMC as well. Finally, the MMC contains logic to determine the state of system security.

11.1.1 Features

- Registers for mapping of address space for on-chip RAM, EEPROM, and Flash EEPROM (or ROM) memory blocks and associated registers
- Memory mapping control and selection based upon address decode and system operating mode
- Core Address Bus control
- Core Data Bus control and multiplexing
- Core Security state decoding
- Emulation Chip Select signal generation ($\overline{\text{ECS}}$)
- External Chip Select signal generation ($\overline{\text{XCS}}$)
- Internal memory expansion
- Miscellaneous system control functions via the MISC register
- Reserved registers for test purposes
- Configurable system memory options defined at integration of Core into the System-on-a-Chip (SOC).

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.1.2 Block Diagram

The block diagram of the MMC is shown in **Figure 11-1** below.

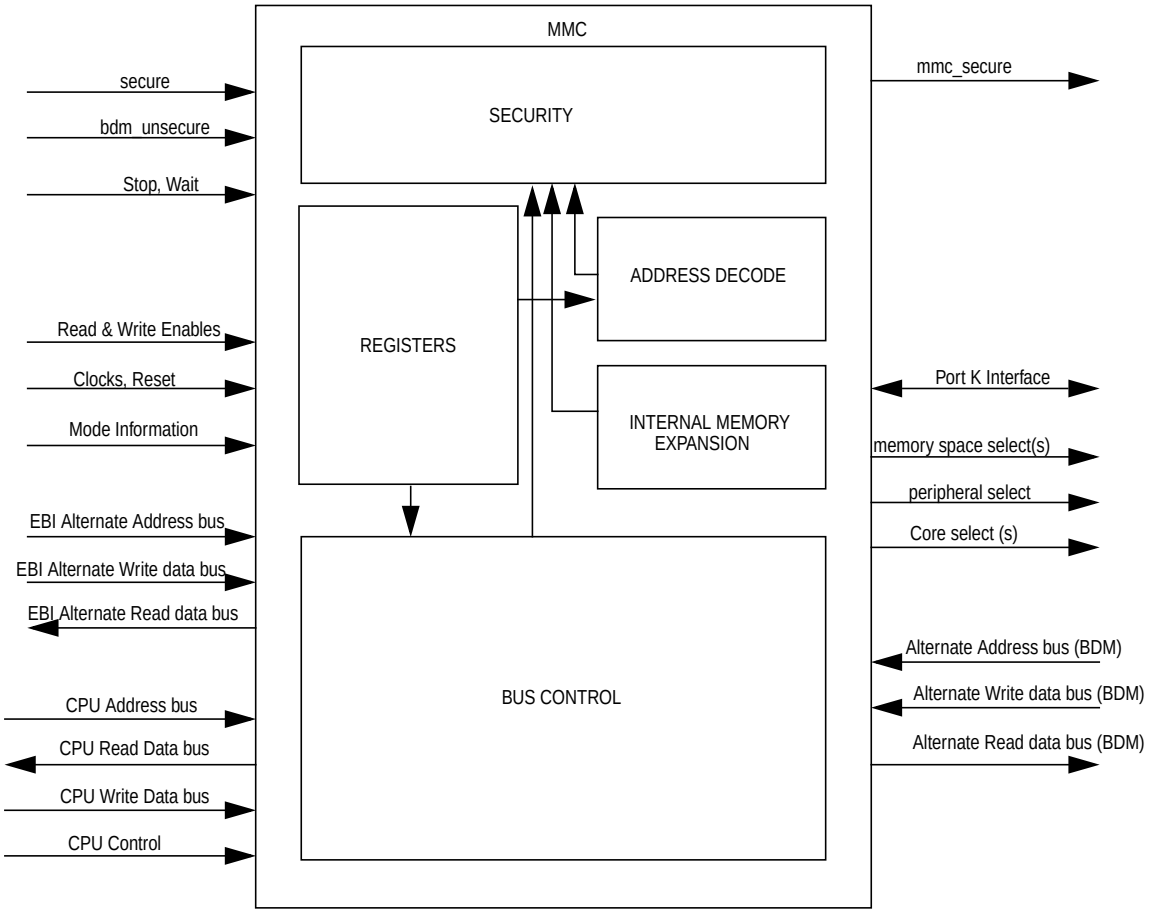


Figure 11-1 Module Mapping Control Block Diagram

11.2 Interface Signals

All interfacing with the MMC sub-block is done within the Core.

11.3 Registers

A summary of the registers associated with the MMC sub-block is shown in **Figure 11-2** below. Detailed descriptions of the registers and bits are given in the subsections that follow.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$0010	INITRM	read	RAM15	RAM14	RAM13	RAM12	RAM11	0	0	RAMHAL
		write								
$0011	INITRG	read	0	REG14	REG13	REG12	REG11	0	0	0
		write								
$0012	INITEE	read	EE15	EE14	EE13	EE12	EE11	0	0	EEON
		write								
$0013	MISC	read	0	0	0	0	EXSTR1	EXSTR0	ROMHM	ROMON
		write								
$0014	Reserved	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0017	Reserved	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$001C	MEMSIZ0	read	reg_sw0	0	eep_sw1	eep_sw0	0	ram_sw2	ram_sw1	ram_sw0
		write								
$001D	MEMSIZ1	read	rom_sw1	rom_sw0	0	0	0	0	pag_sw1	pag_sw0
		write								
$0030	PPAGE	read	0	0	PIX5	PIX4	PIX3	PIX2	PIX1	PIX0
		write								
$0031	Reserved	read	0	0	0	0	0	0	0	0
		write								

= Unimplemented X = Indeterminate

Figure 11-2 Module Mapping Control Register Summary

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.3.1 Initialization of Internal RAM Position Register(INITRM)

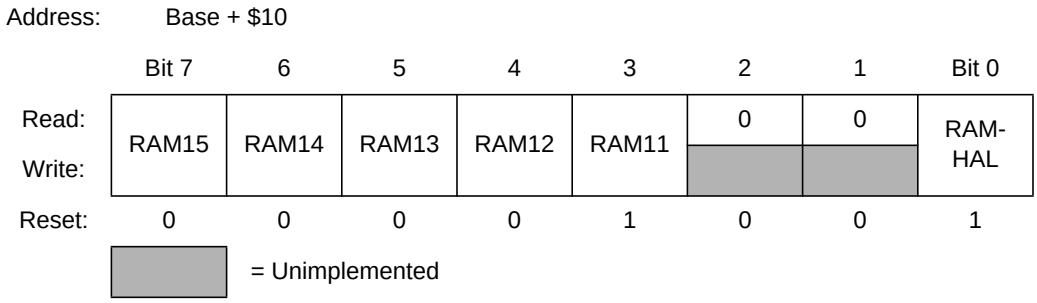


Figure 11-3 INITRM Register

Read: Anytime
Write: Once in Normal and Emulation Modes, anytime in Special Modes

NOTE: *Writes to this register take one cycle to go into effect.*

This register initializes the position of the internal RAM within the on-chip system memory map.

RAM15 - RAM11 - Internal RAM Map Position

These bits determine the upper five bits of the base address for the system's internal RAM array.

RAMHAL - RAM High-align

RAMHAL specifies the alignment of the internal RAM array.
0 = Aligns the RAM to the lowest address ($0000) of the mappable space
1 = Aligns the RAM to the higher address ($FFFF) of the mappable space

11.3.2 Initialization of Internal Registers Position Register (INITRG)

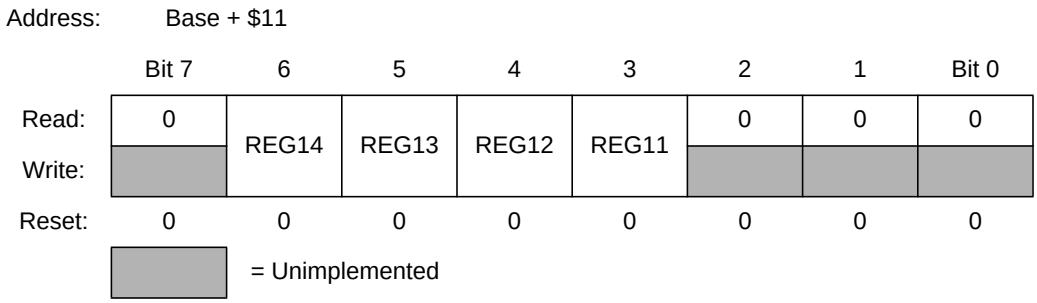


Figure 11-4 INITRG Register

Read: Anytime
Write: Once in Normal and Emulation modes and anytime in Special modes

This register initializes the position of the internal registers within the on-chip system memory map. The registers occupy either a 1K byte or 2K byte space and can be mapped to any 2K byte space within the first 32K bytes of the system's address space.

REG14 - REG11 - Internal Register Map Position

These four bits in combination with the leading zero supplied by bit 7 of INITRG determine the upper five bits of the base address for the system's internal registers (i.e. the minimum base address is $0000 and the maximum is $7FFF).

11.3.3 Initialization of Internal EEPROM Position Register (INITEE)

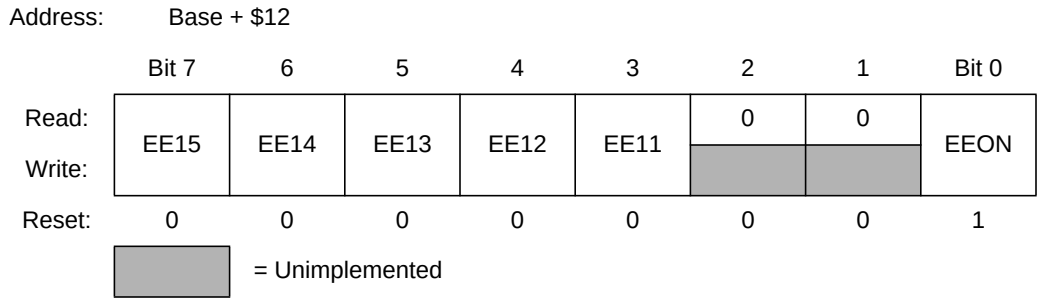


Figure 11-5 INITEE Register

Read: Anytime

Write: Once in Normal and Emulation modes with the exception of the EEON bit which can be written anytime and write anytime in Special modes

NOTE: *Writes to this register take one cycle to go into effect.*

This register initializes the position of the internal EEPROM within the on-chip system memory map.

EE15 - EE11 - Internal EEPROM map position

These bits determine the upper five bits of the base address for the system's internal EEPROM array.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.3.4 Miscellaneous System Control Register (MISC)

Address: Base + $13

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	0	0	EXSTR1	EXSTR0	ROMHM	ROMON
Write:								
Expanded Reset:	0	0	0	0	1	1	0	1
Peripheral or Single Chip Reset	0	0	0	0	1	1	0	1

= Unimplemented

Figure 11-6 Miscellaneous System Control Register (MISC)

NOTES:
1. The reset state of this bit is determined at the chip integration level.

Read: Anytime
Write: As stated in each bit description below

NOTE:*Writes to this register take one cycle to go into effect*

This register initializes miscellaneous control functions.

EXSTR1,0 - External Access Stretch Bits 1 & 0
Write: Once in Normal and Emulation modes and anytime in Special modes

This two bit field determines the amount of clock stretch on accesses to the external address space as shown in **Table 11-1** below. In Single Chip and Peripheral modes these bits have no meaning or effect.

Table 11-1 External Stretch Bit Definition

Stretch bit EXSTR1	Stretch bit EXSTR0	Number of E Clocks Stretched
0	0	0
0	1	1
1	0	2
1	1	3

ROMHM - Flash EEPROM or ROM only in second half of memory map
Write: Once in Normal and Emulation modes and anytime in Special modes

1 = Disables direct access to the Flash EEPROM or ROM in the lower half of the memory map. These physical locations of the Flash EEPROM or ROM can still be accessed through the Program Page window.
0 = The fixed page(s) of Flash EEPROM or ROM in the lower half of the memory map can be accessed.

ROMON - Enable Flash EEPROM or ROM
Write: Once in Normal and Emulation modes and anytime in Special modes

This bit is used to enable the Flash EEPROM or ROM memory in the memory map.
1 = Enables the Flash EEPROM or ROM in the memory map.
0 = Disables the Flash EEPROM or ROM from the memory map.

11.3.5 Reserved Test Register Zero (MTST0)

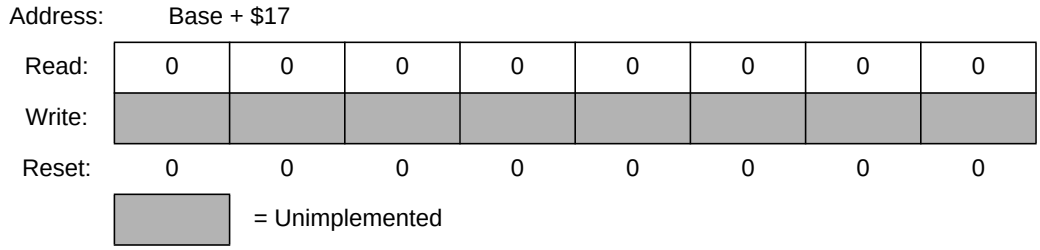


Figure 11-7 Reserved Test Register Zero (MTST0)

Read: Anytime

Write: No effect - this register location is used for internal test purposes.

11.3.6 Reserved Test Register One (MTST1)

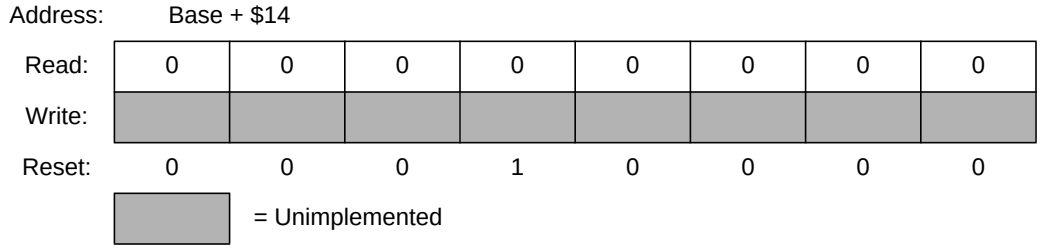


Figure 11-8 Reserved Test Register One (MTST1)

Read: Anytime

Write: No effect - this register location is used for internal test purposes.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.3.7 Memory Size Register Zero (MEMSIZ0)

Figure 11-9 Memory Size Register Zero

Read: Anytime
Write: Writes have no effect

The MEMSIZ0 register reflects the state of the register, EEPROM and RAM memory space configuration switches at the Core boundary which are configured at system integration. This register allows read visibility to the state of these switches.

reg_sw0 - Allocated System Register Space
1 = Allocated system register space size is 2K byte
0 = Allocated system register space size is 1K byte

eep_sw1:eep_sw0 - Allocated System EEPROM Memory Space

The allocated system EEPROM memory space size is as given in **Table 11-2** below.

Table 11-2 Allocated EEPROM Memory Space

eep_sw1:eep_sw0	Allocated EEPROM Space
00	0K byte
01	2K byte
10	4K byte
11	8K byte

ram_sw2:ram_sw0 - Allocated System RAM Memory Space

The allocated system RAM memory space size is as given in **Table 11-3** below.

Table 11-3 Allocated RAM Memory Space

ram_sw2:ram_sw0	Allocated RAM Space	RAM mappable region	INITRM bits used
000	2k Byte	2k Byte	RAM15-RAM11
001	4k Byte	4k Byte	RAM15-RAM12
010	6k Byte	8k Byte[1]	RAM15-RAM13
011	8k Byte	8k Byte	RAM15-RAM13
100	10k Byte	16k Byte[1]	RAM15-RAM14

Table 11-3 Allocated RAM Memory Space

ram_sw2:ram_sw0	Allocated RAM Space	RAM mappable region	INITRM bits used
101	12k Byte	16k Byte[1]	RAM15-RAM14
110	14k Byte	16k Byte[1]	RAM15-RAM14
111	16k Byte	16k Byte	RAM15-RAM14

NOTES:
1. Alignment of the Allocated RAM space within the RAM mappable region is dependent on the value of RAMHAL.

NOTE: *As stated, the bits in this register provide read visibility to the system physical memory space allocations defined at system integration. The actual array size for any given type of memory block may differ from the allocated size. Please refer to the chip-level documentation for actual sizes.*

11.3.8 Memory Size Register One (MEMSIZ1)

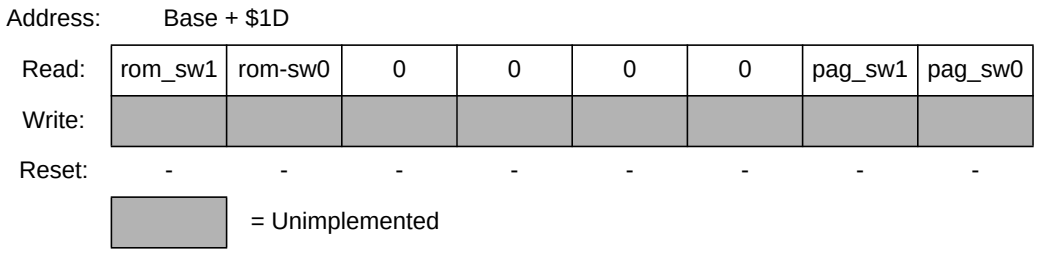


Figure 11-10 Memory Size Register One

Read: Anytime
Write: Writes have no effect

The MEMSIZ1 register reflects the state of the Flash EEPROM or ROM physical memory space and paging switches at the Core boundary which are configured at system integration. This register allows read visibility to the state of these switches.

rom_sw1:rom_sw0 - Allocated System Flash EEPROM or ROM Physical Memory Space

The allocated system Flash EEPROM or ROM physical memory space is as given in **Table 11-4** below.

Table 11-4 Allocated Flash EEPROM/ROM Physical Memory Space

rom_sw1:rom_sw0	Allocated Flash or ROM Space
00	0K byte
01	16K byte
10	48K byte[1]
11	64K byte[1]

1 The ROMHM software bit in the MISC register determines the accessibility of the Flash EEPROM/ROM memory space. Please refer to **11.3.4** for a detailed functional description of the ROMHM bit.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

pag_sw1:pag_sw0 - Allocated Off-Chip Flash EEPROM or ROM Memory Space

The allocated off-chip Flash EEPROM or ROM memory space size is as given in **Table 11-5** below.

Table 11-5 Allocated Off-Chip Memory Options

pag_sw1:pag_sw0	Off-Chip Space	On-Chip Space
00	876K byte	128K byte
01	768K byte	256K byte
10	512K byte	512K byte
11	0K byte	1M byte

NOTE: *As stated, the bits in this register provide read visibility to the system memory space and on-chip/off-chip partitioning allocations defined at system integration. The actual array size for any given type of memory block may differ from the allocated size. Please refer to the chip-level documentation for actual sizes.*

11.3.9 Program Page Index Register (PPAGE)

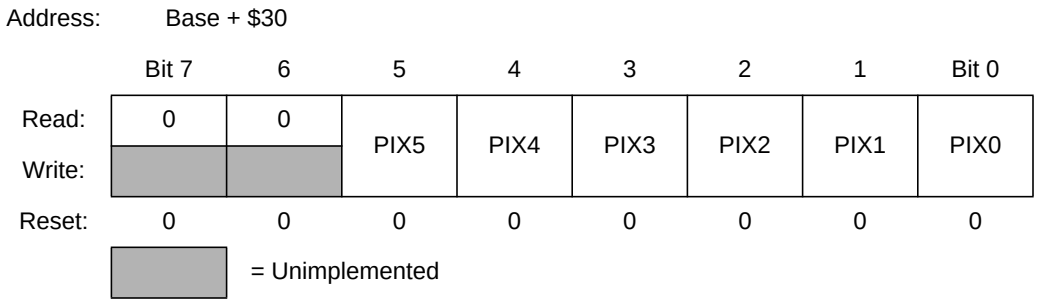


Figure 11-11 Program Page Index Register (PPAGE)

Read: Anytime
Write: Anytime

The STAR12 Core architecture limits the physical address space available to 64K bytes. The Program Page Index Register allows for integrating up to 1M byte of Flash EEPROM or ROM into the system by using the six page index bits to page 16K byte blocks into the Program Page Window located from $8000 to $BFFF as defined in **Table 11-6** below. CALL and RTC instructions have a special single wire mechanism to read and write this register without using the address bus.

NOTE: *Normal writes to this register take one cycle to go into effect. Writes to this register using the special single wire mechanism of the CALL and RTC instructions will be complete before the end of the associated instruction.*

PIX5 - PIX0 - Program Page Index Bits 5-0

These six page index bits are used to select which of the 64 Flash EEPROM or ROM array pages is to be accessed in the Program Page Window as shown in **Table 11-6**.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 11-6 Program Page Index Register Bits

PIX5	PIX4	PIX3	PIX2	PIX1	PIX0	Program Space Selected
0	0	0	0	0	0	16K page 0
0	0	0	0	0	1	16K page 1
0	0	0	0	1	0	16K page 2
0	0	0	0	1	1	16K page 3
. . . . .	. . . . .	. . . .	. . . . .	. . . . .	. . . . .	. . . . .
1	1	1	1	0	0	16K page 60
1	1	1	1	0	1	16K page 61
1	1	1	1	1	0	16K page 62
1	1	1	1	1	1	16K page 63

11.4 Operation

The MMC sub-block performs four basic functions of the Core operation: bus control, address decoding and select signal generation, memory expansion and security decoding for the system. Each aspect is described in the subsections following.

11.4.1 Bus Control

The MMC controls the address bus and data buses that interface the Core with the rest of the system. This includes the multiplexing of the input data buses to the Core onto the main CPU read data bus and control of data flow from the CPU to the output address and data buses of the Core. In addition, the MMC handles all CPU read data bus swapping operations.

11.4.2 Address Decoding

As data flows on the Core address bus, the MMC decodes the address information, determines whether the internal Core register or firmware space, the peripheral space or a memory register or array space is being addressed and generates the correct select signal. This decoding operation also interprets the mode of operation of the system and the state of the mapping control registers in order to generate the proper select. The MMC also generates two external chip select signals, Emulation Chip Select ($\overline{\text{ECS}}$) and External Chip Select ($\overline{\text{XCS}}$).

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

11.4.2.1 Select Priority and Mode Considerations

Although internal resources such as control registers and on-chip memory have default addresses, each can be relocated by changing the default values in control registers. Normally, I/O addresses, control registers, vector spaces, expansion windows, and on-chip memory are mapped so that their address ranges do not overlap. The MMC will make only one select signal active at any given time. This activation is based upon the priority outlined in **Table 11-7** below. If two or more blocks share the same address space, only the select signal for the block with the highest priority will become active. An example of this is if the registers and the RAM are mapped to the same space, the registers will have priority over the RAM and the portion of RAM mapped in this shared space will not be accessible. The expansion windows have the lowest priority. This means that registers, vectors, and on-chip memory are always visible to a program regardless of the values in the page select registers.

Table 11-7 Select Signal Priority

Priority	Address Space
Highest	BDM (internal to Core) firmware or register space
...	Internal register space
...	RAM memory block
...	EEPROM memory block
...	On-chip Flash EEPROM or ROM
Lowest	Remaining external space

In expanded modes, all address space not used by internal resources is by default external memory space. The data registers and data directions registers for Ports A and B are removed from the on-chip memory map and become external accesses. If the EME bit in the MODE register (see **12.3.8**) is set, the data and data direction registers for Port E are also removed from the on-chip memory map and become external accesses.

In Special Peripheral mode, the first 16 registers associated with bus expansion are removed from the on-chip memory map (PORTA, PORTB, DDRA, DDRB, PORTE, DDRE, PEAR, MODE, PUCR, RDRIV and the EBI reserved registers).

In emulation modes, if the EMK bit in the MODE register (see **12.3.8**) is set, the data and data direction registers for Port K are removed from the on-chip memory map and become external accesses.

11.4.2.2 Emulation Chip Select Signal

When the EMK bit in the MODE register (see **12.3.8**) is set, Port K bit 7 is used as an active-low emulation chip select signal, $\overline{\text{ECS}}$. This signal is active when the system is in Emulation mode, the EMK bit is set and the Flash EEPROM or ROM space is being addressed subject to the conditions outlined in **11.4.3.2** below. When the EMK bit is clear, this pin is used for general purpose I/O.

11.4.2.3 External Chip Select Signal

When the EMK bit in the MODE register (see **12.3.8**) is set, Port K bit 6 is used as an active-low external chip select signal, $\overline{\text{XCS}}$. This signal is active only when the $\overline{\text{ECS}}$ signal described above is not active and when the system is addressing the external address space. Accesses to unimplemented locations within the

register space or to locations that are removed from the map (i.e. Ports A and B in Expanded modes) will not cause this signal to become active. When the EMK bit is clear, this pin is used for general purpose I/O.

11.4.3 Memory Expansion

The STAR12 Core architecture limits the physical address space available to 64K bytes. The Program Page Index Register allows for integrating up to 1M byte of Flash EEPROM or ROM into the system by using the six page index bits to page 16K byte blocks into the Program Page Window located from $8000 to $BFFF in the physical memory space. The paged memory space can consist of solely on-chip memory or a combination of on-chip and off-chip memory. This partitioning is configured at system integration through the use of the paging configuration switches (*pag_sw1:pag_sw0*) at the Core boundary. The options available to the integrator are as given in **Table 11-8** below (this table matches **Table 11-5** but is repeated here for easy reference).

Table 11-8 Allocated Off-Chip Memory Options

pag_sw1:pag_sw0	Off-Chip Space	On-Chip Space
00	876K byte	128K byte
01	768K byte	256K byte
10	512K byte	512K byte
11	0K byte	1M byte

Based upon the system configuration, the Program Page Window will consider its access to be either internal or external as defined in **Table 11-9** below.

Table 11-9 External/Internal Page Window Access

pag_sw1:pag_sw0	Partitioning	PIX5:0 Value	Page Window Access
00	876K off-Chip, 128K on-Chip	$00 - $37	external
		$38 - $3F	internal
01	768K off-chip, 256K on-chip	$00 - $2F	external
		$30 - $3F	internal
10	512K off-chip, 512K on-chip	$00 - $1F	external
		$20 - $3F	internal
11	0K off-chip, 1M on-chip	n/a	external
		$00 - $3F	internal

NOTE: *The partitioning as defined in **Table 11-9** above applies only to the allocated memory space and the actual memory sizes implemented in the system may differ. Please refer to the chip-level documentation for actual sizes.*

The PPAGE register holds the page select value for the Program Page WIndow. The value of the PPAGE register can be manipulated by normal read and write instructions as well as the CALL and RTC instructions.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Control registers, vector spaces and a portion of on-chip memory are located in unpaged portions of the 64K byte physical address space. The stack and I/O addresses should also be in unpaged memory to make them accessible from any page.

The starting address of a service routine must be located in unpaged memory because the 16-bit exception vectors cannot point to addresses in paged memory. However, a service routine can call other routines that are in paged memory. The upper 16K byte block of memory space ($C000-$FFFF) is unpaged. It is recommended that all reset and interrupt vectors point to locations in this area.

11.4.3.1 CALL and Return from Call Instructions

CALL and RTC are uninterruptable instructions that automate page switching in the program expansion window. CALL is similar to a JSR instruction, but the subroutine that is called can be located anywhere in the normal 64K byte address space or on any page of program expansion memory. CALL calculates and stacks a return address, stacks the current PPAGE value, and writes a new instruction-supplied value to PPAGE. The PPAGE value controls which of the 64 possible pages is visible through the 16K byte expansion window in the 64K byte memory map. Execution then begins at the address of the called subroutine.

During the execution of a CALL instruction, the CPU:

- Writes the old PPAGE value into an internal temporary register and writes the new instruction-supplied PPAGE value into the PPAGE register.
- Calculates the address of the next instruction after the CALL instruction (the return address), and pushes this 16-bit value onto the stack.
- Pushes the old PPAGE value onto the stack.
- Calculates the effective address of the subroutine, refills the queue, and begins execution at the new address on the selected page of the expansion window.

This sequence is uninterruptable; there is no need to inhibit interrupts during CALL execution. A CALL can be performed from any address in memory to any other address.

The PPAGE value supplied by the instruction is 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. In indexed-indirect variations of CALL, a pointer specifies memory locations where the new page value and the address of the called subroutine are stored. Using indirect addressing for both the new page value and the address within the page allows values calculated at run time rather than immediate values that must be known at the time of assembly.

The RTC instruction terminates subroutines invoked by a CALL instruction. RTC unstacks the PPAGE value and the return address and refills the queue. Execution resumes with the next instruction after the CALL.

During the execution of an RTC instruction, the CPU:

- Pulls the old PPAGE value from the stack
- Pulls the 16-bit return address from the stack and loads it into the PC
- Writes the old PPAGE value into the PPAGE register

- Refills the queue and resumes execution at the return address

This sequence is uninterruptable; an RTC can be executed from anywhere in memory, even from a different page of extended memory in the expansion window.

The CALL and RTC instructions behave like JSR and RTS, except they use more execution cycles. Therefore, routinely substituting CALL/RTC for JSR/RTS is not recommended. JSR and RTS can be used to access subroutines that are on the same page in expanded memory. However, a subroutine in expanded memory that can be called from other pages must be terminated with an RTC. And the RTC unstacks a PPAGE value. So any access to the subroutine, even from the same page, must use a CALL instruction so that the correct PPAGE value is in the stack.

11.4.3.2 Extended Address (XAB19:14) and $\overline{\text{ECS}}$ Signal Functionality

If the EMK bit in the MODE register is set (see **12.3.8**) the PIX5:0 values will be output on XAB19:14 respectively (Port K bits 5:0) when the system is addressing within the physical Program Page Window address space ($8000 - $BFFF) and is in an expanded mode. When addressing anywhere else within the physical address space (outside of the paging space), the XAB19:14 signals will be assigned a constant value based upon the physical address space selected. In addition, the active-low emulation chip select signal, $\overline{\text{ECS}}$, will likewise function based upon the assigned memory allocation. In the cases of 48K byte and 64K byte allocated physical Flash/ROM space, the operation of the $\overline{\text{ECS}}$ signal will additionally depend upon the state of the ROMHM bit (see **11.3.4**) in the MISC register. **Table 11-10**, **Table 11-11**, **Table 11-12** and **Table 11-13** below summarize the functionality of these signals based upon the allocated memory configuration. Again, this signal information is only available externally when the EMK bit is set and the system is in an expanded mode.

Table 11-10 0K Byte Physical Flash/ROM Allocated

Address Space	Page Window Access	ROMHM	$\overline{\text{ECS}}$	XAB19:14
$0000 - $3FFF	n/a	n/a	1	$3D
$4000 - $7FFF	n/a	n/a	1	$3E
$8000 - $BFFF	n/a	n/a	0	PIX5:0
$C000 - $FFFF	n/a	n/a	0	$3F

Table 11-11 16K Byte Physical Flash/ROM Allocated

Address Space	Page Window Access	ROMHM	$\overline{\text{ECS}}$	XAB19:14
$0000 - $3FFF	n/a	n/a	1	$3D
$4000 - $7FFF	n/a	n/a	1	$3E
$8000 - $BFFF	n/a	n/a	1	PIX5:0
$C000 - $FFFF	n/a	n/a	0	$3F

Table 11-12 48K Byte Physical Flash/ROM Allocated

Address Space	Page Window Access	ROMHM	$\overline{\text{ECS}}$	XAB19:14
$0000 - $3FFF	n/a	n/a	1	$3D
$4000 - $7FFF	n/a	0	0	$3E
		1	1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 11-12 48K Byte Physical Flash/ROM Allocated

Address Space	Page Window Access	ROMHM	$\overline{ECS}$	XAB19:14
$8000 - $BFFF	external	n/a	1	PIX5:0
	internal		0	
$C000 - $FFFF	n/a	n/a	0	$3F

Table 11-13 64K Byte Physical Flash/ROM Allocated

Address Space	Page Window Access	ROMHM	$\overline{ECS}$	XAB19:14
$0000 - $3FFF	n/a	0	0	$3D
		1	1	
$4000 - $7FFF	n/a	0	0	$3E
		1	1	
$8000 - $BFFF	external	n/a	1	PIX5:0
	internal	n/a	0	
$C000 - $FFFF	n/a	n/a	0	$3F

A graphical example of a memory paging for a system configured as 1M byte on-chip Flash/ROM with 64K allocated physical space is given in **Figure 11-12** below for illustration.

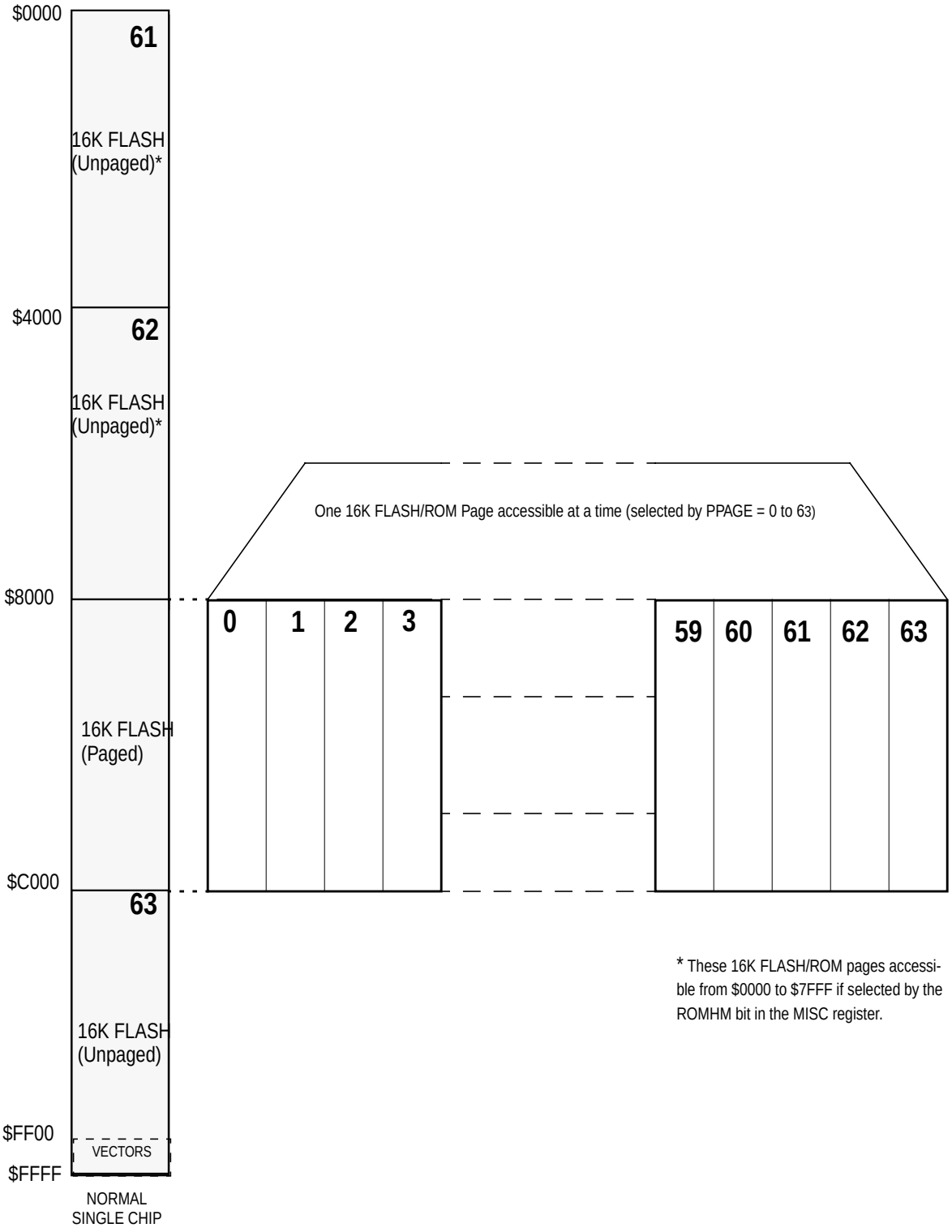


Figure 11-12 Memory Paging Example: 1M Byte On-Chip Flash/ROM, 64K Allocation

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

11.5 Motorola Internal Information

The subsection aspects of the MMC that are considered to be for Motorola internal use only.

11.5.1 Test Registers

There are two test registers for the MMC, MTST[1:0]. These registers are used for internal test purposes to gain visibility into the module select logic.

In all modes, if the FLAGSE bit in MTST1 is set, accesses to internal registers or memory will cause the associated flag to assert. For example, an access into the RAM array will cause the MT01 bit (Bit 1in MTST0 - RAM Array bit) to set. These registers can be read in any mode. If the FLAGSE bit is set, reading the register will cause it to be cleared. A write will have no effect in all modes.

11.5.1.1 Mapping Test Register 0 (MTST0)

Address:	Base + $14							
Read:	MT07	MT06	MT05	MT04	MT03	MT02	MT01	MT00
Write:								
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 11-13 Mapping Test Register Zero (MTST0)

Read: Anytime
Write: No effect

MT0 7-0 - Mapping Test 0

The individual bits are assigned as follows:

MT07 - Core*
MT06 - Peripheral
MT05 - EE Array
MT04 - EE Register
MT03 - Flash Array
MT02 - Flash Register
MT01 - RAM Array
MT00 - RAM Register

* This flag bit will not get set when you are accessing any of the MTST registers

11.5.1.2 Mapping Test Register 1(MTST1)

Address:	Base + $17							
Read:	MT17	MT16	MT15	MT14	MT13	MT12	MT11	MT10
Write:		PNORME	FLAGSE	BKGDPUE				
Reset:	0	0	0	1	0	0	0	0

= Unimplemented

Figure 11-14 Mapping Test Register One (MTST1)

Read: Anytime
Write: See individual bit descriptions

MT17 - Unimplemented (reads back zero)

MT16 - Mapping Test Register 1 Bit 6 (PNORME).

Normally, the system will enter peripheral mode and be in a special mode. Setting this bit will put the system into normal peripheral mode. This is so that testing of register normal mode read/write conditions can be performed while in peripheral mode.

Normal, Special & Emulation: Write never.

Peripheral: Write anytime.
1 = The system operates in normal peripheral mode.
0 = The system operates in special peripheral mode.

MT15 — Mapping Test Register 1 Bit 5 (FLAGSE).

This bit is used to enable the select signal flag function of the MTST registers. When asserted, the MTST registers that have an associated block select signal flag bit will act as flag registers, where an access to the block causes the flag bit to assert. When unasserted, the MTST registers will not act as flag bits.

Normal & Emulation: Write never.

Special: Write anytime.
1 = The MTST registers act as flag bits for the block select signals.
0 = The MTST registers do not act as flag bits for the block select signals.

MT14 - Mapping Test Register 3 Bit 4 (BKGDPUE)

This bit used to enable/disable the pull-up on the BKGD pin.

Normal & Emulation: Write never

Special: Write anytime
1 = The pull-up on the BKGD pin is enabled.
0 = The pull-up on the BKGD pin is disabled.

MT 13-10 — Mapping Test Register 1 Bits 3:0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

11.5.2 MMC Bus Control

This subsection discusses aspects of the bus control/multiplexing performed by the MMC.

11.5.2.1 Address Bus

The MMC multiplexes the EBI Alternate Address Bus, BDM Alternate Address Bus, and the CPU Address Bus to form the main address bus for the Core. The EBI Alternate Address Bus is the address bus source in peripheral mode. The BDM Alternate Address Bus is the address bus source whenever the BDM is driving the bus. The CPU Address Bus is the address bus source whenever the CPU has a valid address, the BDM is not driving the bus and the system is not operating in peripheral mode.

11.5.2.2 Write Data Bus

The CPU Write Data bus, EBI Alternate Write Data bus or BDM Alternate Write Data bus supply data to the master bus. The CPU Write Data bus is the write data source unless the cycle is a BDM access or the system is operating in peripheral mode. The BDM Alternate Write Data bus is the write data source only when the BDM is driving the bus. The EBI Alternate Write Data bus is the write data source in peripheral mode.

11.5.2.3 Read Data Bus

The MMC provides the control to split 16-bit accesses into two cycle operations, when needed. The CPU is paused during the second cycle of the two cycle access. For reads, the MMC takes care of swapping and holding the read data bus so that the CPU will receive the data on the correct location of its read data bus.

An access may also take two cycles when the Interrupt or BDM is driving the address bus, if the system is in a narrow mode and the 16-bit access is to external memory space. In these cases, AB[0] will be forced high during the second cycle.

The MMC will also force those accesses that would normally be two cycle operations into a single cycle operation based upon the Wide Bus Enable signal. This signal will assert when performing a 16-bit access in narrow mode to those locations that are removed from the memory map, as summarized by **Table 11-14**.

Table 11-14 Wide Bus Enable Signal Generation

Address	Register Names	Conditions	mmc_widebuse_t2
$0000 - $0003	PORTA PORTB DDRA DDRB	initrg[4:0] == mmc_ab_t2[15:11] & ebi_emul_t2 & ebi_narrow_t2	1
$0008 - $0009	PORTE DDRE	initrg[4:0] == mmc_ab_t2[15:11] & ebi_emul_t2 & ebi_narrow_t2 & ebi_eme_t2	1
$000A - $000D	PEAR MODE PUCR RDRIV	initrg[4:0] == mmc_ab_t2[15:11] & ebi_emul_t2 & ebi_narrow_t2	1

$0032 - $0033	PORTK DDRK	initrg[4:0] == mmc_ab_t2[15:11] & ebi_emul_t2 & ebi_narrow_t2 & ebi_emk_t2	1
All Others	-	-	0

Table 11-15 summarizes the different access types, where the data is on the internal or external read data bus and where the CPU is expecting the data. The source of the CPU's read data bus for external accesses is the ebi_extrdb and for internal accesses is the rdb_t2.

IMS refers to the Internal Memory Select signal (1 = Internal, 0 = External). FMTS refers to the Fast Memory Transfer Select signal, which asserts anytime an access is made to the RAM except for the last byte of the array.

Table 11-15 Read Data Bus Swapping

MODE	Wide Bus Enable	IMS	FMTS	SZ8	AB[0]	CYCLES	Read Data Bus (Internal or External) -> CPU Read Data Bus
Single Chip	X	1	X	0	0	1	rdbh -> core_rdbh rdbl -> core_rdbl
	X	1	0	0	1	2	1. rdbl -> core_rdbh 2. rdbh -> core_rdbl
	X	1	X	1	0	1	rdbh -> core_rdbl
	X	1	X	1	1	1	rdbl -> core_rdbl
	X	1	1	0	1	1	rdbl -> core_rdbh rdbh -> core_rdbl
Normal Expanded Narrow	X	0	X	0	0	2	1. extrdbh -> core_rdbh 2. extrdbl -> core_rdbl
	X	0	X	0	1	2	1. extrdbl -> core_rdbh 2. extrdbh -> core_rdbl
	X	0	X	1	0	1	extrdbh -> core_rdbl
	X	0	X	1	1	1	extrdbl -> core_rdbl
	X	1	X	0	0	1	rdbh -> core_rdbh rdbl -> core_rdbl
	X	1	0	0	1	2	1. rdbl -> core_rdbh 2. rdbh -> core_rdbl
	X	1	X	1	0	1	rdbh -> core_rdbl
	X	1	X	1	1	1	rdbl -> core_rdbl
	X	1	1	0	1	1	rdbl -> core_rdbh rdbh -> core_rdbl

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 11-15 Read Data Bus Swapping

MODE	Wide Bus Enable	IMS	FMTS	SZ8	AB[0]	CYCLES	Read Data Bus (Internal or External) -> CPU Read Data Bus
Emulation Expanded Narrow	0	0	X	0	0	2	1. extrdbh -> core_rdbh 2. extrdbl -> core_rdbl
	X	0	X	0	1	2	1. extrdbl -> core_rdbh 2. extrdbh -> core_rdbl
	1	0	X	0	0	1	extrdbh -> core_rdbh extrdbl -> core_rdbl
	X	0	X	1	0	1	extrdbh -> core_rdbl
	X	0	X	1	1	1	extrdbl -> core_rdbl
	X	1	X	0	0	1	rdbh -> core_rdbh rdbl -> core_rdbl
	X	1	0	0	1	2	1. rdbl -> core_rdbh 2. rdbh -> core_rdbl
	X	1	X	1	0	1	rdbh -> core_rdbl
	X	1	X	1	1	1	rdbl -> core_rdbl
	X	1	1	0	1	1	rdbl -> core_rdbh rdbh -> core_rdbl
Expanded Wide	X	X	X	0	0	1	(ext)rdbh -> core_rdbh (ext)rdbl -> core_rdbl
	X	X	0	0	1	2	1. (ext)rdbl -> core_rdbh 2. (ext)rdbh -> core_rdbl
	X	X	X	1	0	1	(ext)rdbh -> core_rdbl
	X	X	X	1	1	1	(ext)rdbl -> core_rdbl
	X	X	1	0	1	1	(ext)rdbl -> core_rdbh (ext)rdbh -> core_rdbl

Section 12 Multiplexed External Bus Interface (MEBI)

This section describes the functionality of the Multiplexed External Bus Interface (MEBI) sub-block of the Core.

12.1 Overview

The MEBI sub-block of the Core serves to provide access and/or visibility to internal Core data manipulation operations including timing reference information at the external boundary of the Core and/or system. Depending upon the system operating mode and the state of bits within the control registers of the MEBI, the internal 16-bit read and write data operations will be represented in 8-bit or 16-bit accesses externally. Using control information from other blocks within the system, the MEBI will determine the appropriate type of data access to be generated.

12.1.1 Features

- External bus controller with four 8-bit ports (A,B, E and K)
- Data and data direction registers for ports A, B E and K when used as general purpose I/O
- Control register to enable/disable alternate functions on Port E and Port K
- Mode control register
- Control register to enable/disable pullups on Ports A, B, E and K
- Control register to enable/disable reduced output drive on Ports A, B, E and K
- Control register to configure external clock behavior
- Control register to configure $\overline{\text{IRQ}}$ pin operation
- Logic to capture and synchronize external interrupt pin inputs

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.1.2 Block Diagram

The block diagram of the MEBI sub-block is shown in **Figure 12-1** below.

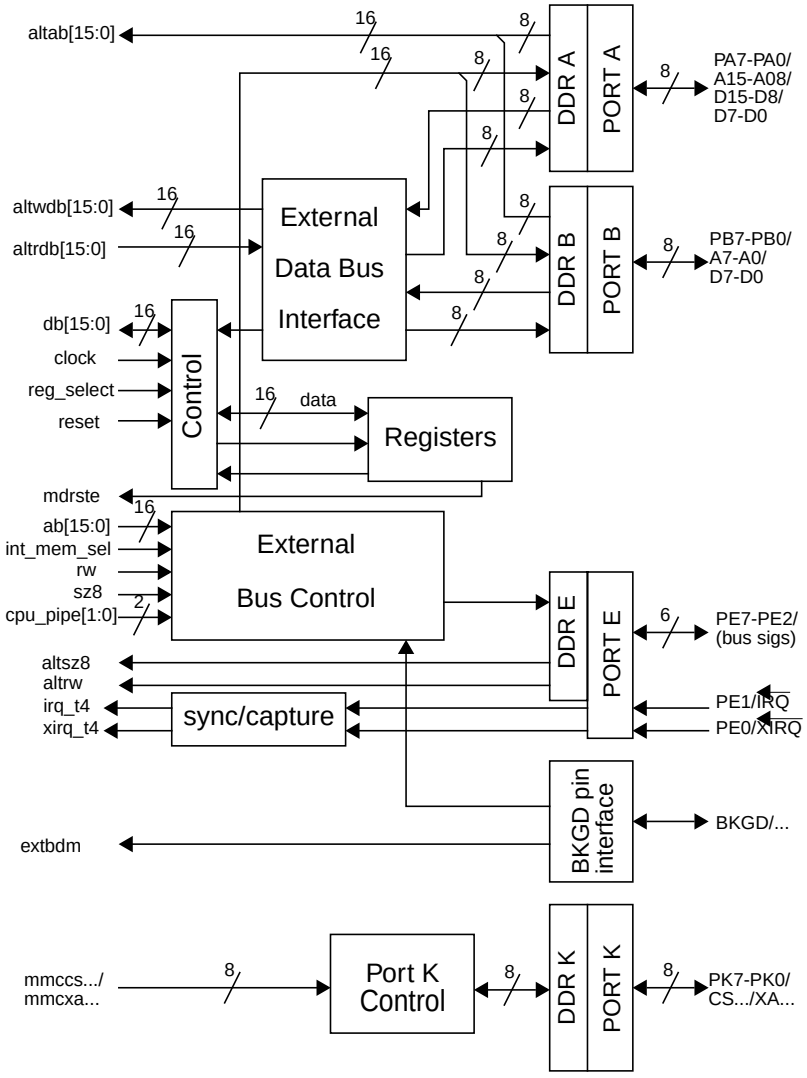


Figure 12-1 MEBI Block Diagram

In the figure, the signals on the right hand side represent pins that are accessible externally to the Core and/or system.

12.2 Interface Signals

Much of the interfacing with the MEBI sub-block is done within the Core; however, many of the MEBI signals pass through the Core boundary and interface with the system port/pad logic for Ports A, B, E and

K. The Core interface signals associated with the MEBI are shown in **Table 12-1** below. The functional descriptions of the signals are provided below for completeness.

Table 12-1 MEBI Interface Signal Definitions

Signal Name	Type	Functional Description
External Bus Interface Signals		
core_paind[7:0]	I	Port A input data [7:0]
core_pado[7:0]	O	Port A data output [7:0]
core_paobe[7:0]	O	Port A output buffer enable [7:0]
core_paibe_t2	O	Port A input buffer enable
core_papue_t2	O	Port A pullup enable
core_padse_t2	O	Port A drive strength enable
core_pbind[7:0]	I	Port B input data [7:0]
core_pbdo[7:0]	O	Port B data output [7:0]
core_pbobe[7:0]	O	Port B output buffer enable [7:0]
core_pbibe_t2	O	Port B input buffer enable
core_pbpue_t2	O	Port B pullup enable
core_pbdse_t2	O	Port B drive strength enable
core_peind[7:0]	I	Port E input data [7:0] *NOTE:* PE1 is $\overline{\text{IRQ}}$ pin input; PE0 is $\overline{\text{XIRQ}}$ pin input.
core_pedo[7:0]	O	Port E data output [7:0]
core_peobe[7:0]	O	Port E output buffer enable [7:0]
core_peibe_t2	O	Port E input buffer enable
core_pepue_t2	O	Port E pullup enable
core_mdrste	O	Enable signal for EBI Mode pin pullups at the pad
core_pedse_t2	O	Port E drive strength enable
core_pkind[7:0]	I	Port K input data [7:0]
core_pkdo[7:0]	O	Port K data output [7:0]
core_pkobe[7:0]	O	Port K output buffer enable [7:0]
core_pkibe_t2	O	Port K input buffer enable
core_pkpue_t2	O	Port K pullup enable
core_pkdse_t2	O	Port K drive strength enable

12.2.1 MEBI Signal Descriptions

These descriptions apply to the MEBI signals that pass through the Core boundary and interface with the system External Bus Interface port/pad logic.

12.2.1.1 Port A Input Data to Core (core_paind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port A.

12.2.1.2 Port A Output Data from Core (core_pado[7:0])

This 8-bit wide output from the Core provides the Port A data output to the system port/pad logic for Port A.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.2.1.3 Port A output buffer enable from Core (core_paobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port A.

12.2.1.4 Port A input buffer enable from Core (core_paibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port A.

12.2.1.5 Port A pullup enable from Core (core_papue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port A should be enabled for all Port A pins.

12.2.1.6 Port A drive strength enable from Core (core_padse_t2)

This single bit output from the Core indicates whether all Port A pins will operate with full or reduced drive strength.

12.2.1.7 Port B Input Data to Core (core_pbind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port B.

12.2.1.8 Port B Output Data from Core (core_pbdo[7:0])

This 8-bit wide output from the Core provides the Port B data output to the system port/pad logic for Port B.

12.2.1.9 Port B output buffer enable from Core (core_pbobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port B.

12.2.1.10 Port B input buffer enable from Core (core_pbibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port B.

12.2.1.11 Port B pullup enable from Core (core_pbpue_t2)

When asserted (logic 1), this single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port B should be enabled for all Port B pins.

12.2.1.12 Port B drive strength enable from Core (core_pbdse_t2)

This single bit output from the Core indicates whether all Port B pins will operate with full or reduced drive strength.

12.2.1.13 Port E Input Data to Core (core_peind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port E. When the system has an external $\overline{\text{IRQ}}$ pin implemented, the input signal from the $\overline{\text{IRQ}}$ pin pad logic must be tied to Port E Input Data Bit 1. Likewise, when the system has an external $\overline{\text{XIRQ}}$ pin implemented, the input signal from the $\overline{\text{XIRQ}}$ pin pad logic must be tied to Port E Input Data Bit 0. Both the $\overline{\text{IRQ}}$ and $\overline{\text{XIRQ}}$ signals are active low (i.e. their asserted state is logic 0).

12.2.1.14 Port E Output Data from Core (core_pedo[7:0])

This 8-bit wide output from the Core provides the Port E data output to the system port/pad logic for Port E.

12.2.1.15 Port E output buffer enable from Core (core_peobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port E.

12.2.1.16 Port E input buffer enable from Core (core_peibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port E.

12.2.1.17 Port E pullup enable from Core (core_pepue_t2)

This single bit output from the Core indicates whether or not the pullup devices within the system port/pad logic for Port E should be enabled for all Port E pins except the MODA (PE5) and MODB (PE6) pins.

12.2.1.18 Port E MODE pin pullup enable from Core (core_mdrste)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for the MODA (PE5) and MODB (PE6) pins within Port E should be enabled.

12.2.1.19 Port E drive strength enable from Core (core_pedse_t2)

This single bit output from the Core indicates whether all Port E pins will operate with full or reduced drive strength.

12.2.1.20 Port K Input Data to Core (core_pkind[7:0])

This 8-bit wide input to the Core provides the Core with the input data from the system port/pad logic for Port K.

12.2.1.21 Port K Output Data from Core (core_pkdo[7:0])

This 8-bit wide output from the Core provides the Port K data output to the system port/pad logic for Port K.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.2.1.22 Port K output buffer enable from Core (core_pkobe[7:0])

This 8-bit wide output from the Core provides the bit-by-bit output buffer enable signal to the system port/pad logic for Port K.

12.2.1.23 Port K input buffer enable from Core (core_pkibe_t2)

This single bit output from the Core provides the input buffer enable signal to the system port/pad logic for Port K.

12.2.1.24 Port K pullup enable from Core (core_pkpue_t2)

This single bit output from the Core indicates that the pullup devices within the system port/pad logic for Port K should be enabled for all Port K pins.

12.2.1.25 Port K drive strength enable from Core (core_pkdse_t2)

This single bit output from the Core indicates whether all Port K pins will operate with full or reduced drive strength.

12.3 Registers

A summary of the registers associated with the MEBI sub-block is shown in **Figure 12-2** below. Detailed descriptions of the registers and bits are given in the subsections that follow.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$0000	PORTA	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0001	PORTB	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0002	DDRA	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0003	DDRB	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0004	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0005	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0006	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0007	Reserved	read	0	0	0	0	0	0	0	0
		write								
$0008	PORTE	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0009	DDRE	read	Bit 7	6	5	4	3	2	0	0
		write								
$000A	PEAR	read	NOACCE	0	PIPOE	NECLK	LSTRE	RDWE	0	0
		write								
$000B	MODE	read	MODC	MODB	MODA	0	IVIS	0	EMK	EME
		write								
$000C	PUCR	read	PUPKE	0	0	PUPEE	0	0	PUPBE	PUPAE
		write								
$000D	RDRIV	read	RDPK	0	0	RDPE	0	0	RDPB	RDPA
		write								
$000E	EBICTL	read	0	0	0	0	0	0	0	ESTR
		write								
$000F	Reserved	read	0	0	0	0	0	0	0	0
		write								
$001E	IRQCR	read	IRQE	IRQEN	0	0	0	0	0	0
		write								
$0032	PORTK	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$0033	DDRK	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								

= Unimplemented X = Indeterminate

Figure 12-2 MEBI Register Map Summary

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.3.1 Port A Data Register(PORTA)

Address: Base + $___0

	BIT 7	6	5	4	3	2	1	BIT 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	-	-	-	-	-	-	-	-
Single Chip:	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Exp Wide, Emul. Nar with IVIS & Periph:	AB/ DB15	AB/ DB14	AB/ DB13	AB/ DB12	AB/ DB11	AB/ DB10	AB/ DB9	AB/ DB8
Expanded Narrow:	AB15 & DB15/ DB7	AB14 & DB14/ DB6	AB13 & DB13/ DB5	AB12 & DB12/ DB4	AB11 & DB11/ DB3	AB10 & DB10/ DB2	AB9 & DB9/ DB1	AB8 & DB8/ DB0

Figure 12-3 Port A Data Register (PORTA)

Read: anytime when register is in the map

Write: anytime when register is in the map

Port A bits 7 through 0 are associated with address lines A15 through A8 respectively and data lines D15/D7 through D8/D0 respectively. When this port is not used for external addresses such as in single-chip mode, these pins can be used as general purpose I/O. Data Direction Register A (DDRA) determines the primary direction of each pin. DDRA also determines the source of data for a read of PORTA.

This register is not in the on-chip memory map in expanded and peripheral modes.

CAUTION:

To ensure that you read the value present on the PORTA pins, always wait at least one cycle after writing to the DDRA register before reading from the PORTA register.

12.3.2 Data Direction Register A (DDRA)

Address: Base + $___2

	BIT 7	6	5	4	3	2	1	BIT 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

Figure 12-4 Data Direction Register A (DDRA)

Read: anytime when register is in the map
Write: anytime when register is in the map

This register controls the data direction for Port A. When Port A is operating as a general purpose I/O port, DDRA determines the primary direction for each Port A pin. A “1” causes the associated port pin to be an output and a “0” causes the associated pin to be a high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTA register. If the DDR bit is zero (input) the buffered pin input state is read. If the DDR bit is one (output) the associated port data register bit state is read.

This register is not in the on-chip map in expanded and peripheral modes. It is reset to $00 so the DDR does not override the three-state control signals.

DDRA7-0 — Data Direction Port A
1 = Configure the corresponding I/O pin as an output
0 = Configure the corresponding I/O pin as an input

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.3.3 Port B Data Register (PORTB)

Address: Base + $___1

	BIT 7	6	5	4	3	2	1	BIT 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	-	-	-	-	-	-	-	-
Single Chip:	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
Exp Wide, Emul. Nar with IVIS & Periph:	AB/DB7	AB/DB6	AB/DB5	AB/DB4	AB/DB3	AB/DB2	AB/DB1	AB/DB0
Expanded Narrow:	AB7	AB6	AB5	AB4	AB3	AB2	AB1	AB0

Figure 12-5 Port B Data Register (PORTB)

Read: anytime when register is in the map

Write: anytime when register is in the map Port B bits 7 through 0 are associated with address lines A7 through A0 respectively and data lines D7 through D0 respectively. When this port is not used for external addresses, such as in single-chip mode, these pins can be used as general purpose I/O. Data Direction Register B (DDRB) determines the primary direction of each pin. DDRB also determines the source of data for a read of PORTB.

This register is not in the on-chip map in expanded and peripheral modes

CAUTION:

To ensure that you read the value present on the PORTB pins, always wait at least one cycle after writing to the DDRB register before reading from the PORTB register.

12.3.4 Data Direction Register B (DDRB)

Address: Base + $___3

	BIT 7	6	5	4	3	2	1	BIT 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

Figure 12-6 Data Direction Register B (DDRB)

Read: anytime when register is in the map
Write: anytime when register is in the map

This register controls the data direction for Port B. When Port B is operating as a general purpose I/O port, DDRB determines the primary direction for each Port B pin. A "1" causes the associated port pin to be an output and a “0” causes the associated pin to be a high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTB register. If the DDR bit is zero (input) the buffered pin input state is read. If the DDR bit is one (output) the associated port data register bit state is read.

This register is not in the on-chip map in expanded and peripheral modes. It is reset to $00 so the DDR does not override the three-state control signals.

DDRB7-0 — Data Direction Port B

1 = Configure the corresponding I/O pin as an output

0 = Configure the corresponding I/O pin as an input

12.3.5 Port E Data Register (PORTE)

Address: Base + $___8

	BIT 7	6	5	4	3	2	1	BIT 0
Read:	Bit 7	6	5	4	3	2	Bit 1	Bit 0
Write:								
Reset:	-	-	-	-	-	-	-	-
Alt. Pin Function:	NOACC	MODB or IPIPE1 or CLKTO	MODA or IPIPE0	ECLK	$\overline{\text{LSTRB}}$ or $\overline{\text{TAG-LO}}$	R/$\overline{\text{W}}$	$\overline{\text{IRQ}}$	$\overline{\text{XIRQ}}$

= Unimplemented

Figure 12-7 Port E Data Register (PORTE)

Read: anytime when register is in the map

Write: anytime when register is in the map

Port E is associated with external bus control signals and interrupt inputs. These include mode select (MODB/IPIPE1, MODA/IPIPE0), E clock, size ($\overline{\text{LSTRB}}$/$\overline{\text{TAGLO}}$), read / write (R/$\overline{\text{W}}$), $\overline{\text{IRQ}}$, and $\overline{\text{XIRQ}}$. When not used for one of these specific functions, Port E pins 7-2 can be used as general purpose I/O and pins 1-0 can be used as general purpose input. The Port E Assignment Register (PEAR) selects the function of each pin and DDRE determines whether each pin is an input or output when it is configured to be general purpose I/O. DDRE also determines the source of data for a read of PORTE.

Some of these pins have software selectable pullups (PE7, ECLK, $\overline{\text{LSTRB}}$, R/$\overline{\text{W}}$, $\overline{\text{IRQ}}$ and $\overline{\text{XIRQ}}$). A single control bit enables the pullups for all of these pins when they are configured as inputs

This register is not in the on-chip map in peripheral mode or in expanded modes when the EME bit is set

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CAUTION:

It is unwise to write PORTE and DDRE as a word access. If you are changing Port E pins from being inputs to outputs, the data may have extra transitions during the write. It is best to initialize PORTE before enabling as outputs.

CAUTION:

To ensure that you read the value present on the PORTE pins, always wait at least one cycle after writing to the DDRE register before reading from the PORTE register

12.3.6 Data Direction Register E (DDRE)

Address: Base + $___9

	BIT 7	6	5	4	3	2	1	BIT 0
Read:	Bit 7	6	5	4	3	Bit 2	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 12-8 Data Direction Register E (DDRE)

Read: anytime when register is in the map
Write: anytime when register is in the map

Data Direction Register E is associated with Port E. For bits in Port E that are configured as general purpose I/O lines, DDRE determines the primary direction of each of these pins. A “1” causes the associated bit to be an output and a “0” causes the associated bit to be an input. Port E bit 1 (associated with $\overline{\text{IRQ}}$) and bit 0 (associated with $\overline{\text{XIRQ}}$) cannot be configured as outputs. Port E, bits 1 and 0, can be read regardless of whether the alternate interrupt function is enabled. The value in a DDR bit also affects the source of data for reads of the corresponding PORTE register. If the DDR bit is zero (input) the buffered pin input state is read. If the DDR bit is one (output) the associated port data register bit state is read.

This register is not in the on-chip map in peripheral mode. It is also not in the map in expanded modes while the EME control bit is set.

DDRE7-2 — Data Direction Port E

1 = Configure the corresponding I/O pin as an output
0 = Configure the corresponding I/O pin as an input

CAUTION:

It is unwise to write PORTE and DDRE as a word access. If you are changing Port E pins from inputs to outputs, the data may have extra transitions during the write. It is best to initialize PORTE before enabling as outputs.

12.3.7 Port E Assignment Register (PEAR)

Address: Base + $___A

	BIT 7	6	5	4	3	2	1	BIT 0	
Read:	NOACCE	0	PIPOE	NECLK	LSTRE	RDWE	0	0	
Write:									
Reset:	0	0	0	0	0	0	0	0	Special Single Chip
Reset:	0	0	1	0	1	1	0	0	Special Test
Reset:	0	0	0	0	0	0	0	0	Peripheral
Reset:	1	0	1	0	1	1	0	0	Emulation Exp Nar
Reset:	1	0	1	0	1	1	0	0	Emulation Exp Wide
Reset:	0	0	0	1	0	0	0	0	Normal Single Chip
Reset:	0	0	0	0	0	0	0	0	Normal Exp Nar
Reset:	0	0	0	0	0	0	0	0	Normal Exp Wide

= Unimplemented

Figure 12-9 Port E Assignment Register (PEAR)

Read: anytime (provided this register is in the map).

Write: each bit has specific write conditions. Please refer to the descriptions of each bit on the following pages.Port E serves as general purpose I/O or as system and bus control signals. The PEAR register is used to choose between the general purpose I/O function and the alternate control functions. When an alternate control function is selected, the associated DDRE bits are overridden.

The reset condition of this register depends on the mode of operation because bus control signals are needed immediately after reset in some modes. In normal single chip mode, no external bus control signals are needed so all of Port E is configured for general purpose I/O. In normal expanded modes, only the E clock is configured for its alternate bus control function and the other bits of Port E are configured for general purpose I/O. As the reset vector is located in external memory, the E clock is required for this access. R/$\overline{W}$ is only needed by the system when there are external writable resources. If the normal expanded system needs any other bus control signals, PEAR would need to be written before any access that needed the additional signals. In special test and emulation modes, IPIPE1, IPIPE0, E, $\overline{LSTRB}$ and R/$\overline{W}$ are configured out of reset as bus control signals

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

This register is not in the on-chip map in emulation and peripheral modes.

NOACCE - CPU No Access Output Enable

Normal: write once

Emulation: write never

Special: write anytime

1 = The associated pin (Port E bit 7) is output and indicates whether the cycle is a CPU free cycle.
0 = The associated pin (Port E bit 7) is general purpose I/O.

This bit has no effect in single chip or peripheral modes.

PIPOE - Pipe Status Signal Output Enable

Normal: write once

Emulation: write never

Special: write anytime.

1 = The associated pins (Port E bits 6:5) are outputs and indicate the state of the instruction queue
0 = The associated pins (Port E bits 6:5) are general purpose I/O.

This bit has no effect in single chip or peripheral modes.

NECLK - No External E Clock

Normal and Special: write anytime

Emulation: write never

1 = The associated pin (Port E bit-4) is a general purpose I/O pin.
0 = The associated pin (Port E bit-4) is the external E clock pin. External E clock is free-running if ESTR=0

External E clock is available as an output in all modes.

LSTRE - Low Strobe ($\overline{\text{LSTRB}}$) Enable

Normal: write once

Emulation: write never

Special: write anytime.

1 = The associated pin (Port E bit-3) is configured as the $\overline{\text{LSTRB}}$ bus control output. If BDM tagging is enabled, $\overline{\text{TAGLO}}$ is multiplexed in on the rising edge of ECLK and $\overline{\text{LSTRB}}$ is driven out on the falling edge of ECLK.
0 = The associated pin (Port E bit-3) is a general purpose I/O pin.

This bit has no effect in single chip, peripheral or normal expanded narrow modes.

NOTE: *$\overline{LSTRB}$ is used during external writes. After reset in normal expanded mode, $\overline{LSTRB}$ is disabled to provide an extra I/O pin. If $\overline{LSTRB}$ is needed, it should be enabled before any external writes. External reads do not normally need $\overline{LSTRB}$ because all 16 data bits can be driven even if the system only needs 8 bits of data.*

RDWE - Read / Write Enable

Normal: write once

Emulation: write never

Special: write anytime

1 = The associated pin (Port E bit-2) is configured as the R/$\overline{W}$ pin

0 = The associated pin (Port E bit-2) is a general purpose I/O pin.

This bit has no effect in single chip or peripheral modes.

NOTE: *R/$\overline{W}$ is used for external writes. After reset in normal expanded mode, R/$\overline{W}$ is disabled to provide an extra I/O pin. If R/$\overline{W}$ is needed it should be enabled before any external writes.*

12.3.8 MODE Register (MODE)

Address: Base + $___B

	BIT 7	6	5	4	3	2	1	BIT 0	
Read:	MODC	MODB	MODA	0	IVIS	0	EMK	EME	
Write:									
Reset:	0	0	0	0	0	0	0	0	Special Single chip
Reset:	0	0	1	0	1	0	1	1	Emulation Exp Nar
Reset:	0	1	0	0	1	0	0	0	Special Test
Reset:	0	1	1	0	1	0	1	1	Emulation Exp Wide
Reset:	1	0	0	0	0	0	0	0	Normal Single Chip
Reset:	1	0	1	0	0	0	0	0	Normal Exp Nar
Reset:	1	1	0	0	0	0	0	0	Peripheral
Reset:	1	1	1	0	0	0	0	0	Normal Exp Wide

= Unimplemented

Figure 12-10 MODE Register (MODE)

Read: anytime (provided this register is in the map).

Write: each bit has specific write conditions. Please refer to the descriptions of each bit on the following pages.

The MODE register is used to establish the operating mode and other miscellaneous functions (i.e. internal visibility and emulation of Port E and K).

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

In peripheral modes, this register is not accessible but it is reset as shown to configure system features. Changes to bits in the MODE register are delayed one cycle after the write.

This register is not in the on-chip map in emulation and peripheral modes.

MODC, MODB, MODA - Mode Select bits

These bits indicate the current operating mode.

If MODA=1, then MODC, MODB, MODA are write never.

If MODC=MODA=0, then MODC, MODB, MODA are write anytime except that you cannot change to or from peripheral mode

If MODC=1, MODB=0 and MODA=0, then MODC is write never, and MODB, MODA are write once, except that you cannot change to peripheral, special test, special single chip, or emulation modes.

Table 12-2 MODC, MODB, MODA Write Capability[1]

MODC	MODB	MODA	Mode	MODx Write Capability
0	0	0	Special Single Chip	MODC, B, A write anytime but not to 110[2]
0	0	1	Emulation Narrow	no write
0	1	0	Special Test	MODC, B, A write anytime but not to 110[2]
0	1	1	Emulation Wide	no write
1	0	0	Normal Single Chip	MODC write never, MODB, A write once but not to 110
1	0	1	Normal Expanded Narrow	no write
1	1	0	Special Peripheral	no write
1	1	1	Normal Expanded Wide	no write

NOTES:

1. No writes to the MOD bits are allowed while operating in a SECURE mode. For more details refer to the security specification document.
2. If you are in a special single chip or special test mode and you write to this register, changing to normal single chip mode, then one allowed write to this register remains. If you write to normal expanded or emulation mode, then no writes remain.

Table 12-3 Mode Select and State of Mode Bits

Input BKGD & bit MODC	Input & bit MODB	Input & bit MODA	Mode Description
0	0	0	Special Single Chip, BDM allowed and ACTIVE. BDM is “allowed” in all other modes but a serial command is required to make BDM “active”.
0	0	1	Emulation Expanded Narrow, BDM allowed

Table 12-3 Mode Select and State of Mode Bits

Input BKGD & bit MODC	Input & bit MODB	Input & bit MODA	Mode Description
0	1	0	Special Test (Expanded Wide), BDM allowed
0	1	1	Emulation Expanded Wide, BDM allowed
1	0	0	Normal Single Chip, BDM allowed
1	0	1	Normal Expanded Narrow, BDM allowed
1	1	0	Peripheral; BDM allowed but bus operations would cause bus conflicts (must not be used)
1	1	1	Normal Expanded Wide, BDM allowed

IVIS - Internal Visibility (for both read and write accesses)

This bit determines whether internal accesses generate a bus cycle that is visible on the external bus.

Normal: write once

Emulation: write never

Special: write anytime

1 = Internal bus operations are visible on external bus.

0 = No visibility of internal bus operations on external bus.

Reference Section **12.4.9** for mode availability of this bit.

EMK - Emulate Port K

Normal: write once

Emulation: write never

Special: write anytime

1 = If in any expanded mode, PORTK and DDRK are removed from the memory map.

0 = PORTK and DDRK are in the memory map so Port K can be used for general purpose I/O.

In single-chip modes, PORTK and DDRK are always in the map regardless of the state of this bit.

In peripheral modes, PORTK and DDRK are never in the map regardless of the state of this bit.

EME - Emulate Port E

Normal and Emulation: write never

Special: write anytime

1 = If in any expanded mode or special peripheral mode, PORTE and DDRE are removed from the memory map. Removing the registers from the map allows the user to emulate the function of these registers externally.

0 = PORTE and DDRE are in the memory map so Port E can be used for general purpose I/O.

In single-chip modes, PORTE and DDRE are always in the map regardless of the state of this bit.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.3.9 Pullup Control Register (PUCR)

Address: Base + $___C

	BIT 7	6	5	4	3	2	1	BIT 0
Read:	PUPKE	0	0	PUPEE	0	0	PUPBE	PUPAE
Write:								
Reset:[1]	1	0	0	1	0	0	0	0

= Unimplemented

Figure 12-11 Pullup Control Register (PUCR)

NOTES:
1. The reset state of this register may be controlled by an instantiation parameter as described in the STAR12 V1.5 Core Integration Guide. The default value of this parameter is shown. Please refer to the specific device User's Guide to determine the actual reset state of this register.

Read: anytime (provided this register is in the map).
Write: anytime (provided this register is in the map).

This register is used to select pullup resistors for the pins associated with the core ports. Pullups are assigned on a per-port basis and apply to any pin in the corresponding port that is currently configured as an input.

This register is not in the on-chip map in emulation and peripheral modes.

NOTE: These bits have no effect when the associated pin(s) are outputs. (The pullups are inactive.)

PUPKE - Pullup Port K Enable
1 = Enable pullup devices for Port K input pins.
0 = Port K pullups are disabled.

PUPEE - Pullup Port E Enable
1 = Enable pullup devices for Port E input pins bits 7, 4-0.
0 = Port E pullups on bit 7, 4-0 are disabled.

PUPBE - Pullup Port B Enable
1 = Enable pullup devices for all Port B input pins.
0 = Port B pullups are disabled.

PUPAE - Pullup Port A Enable
1 = Enable pullup devices for all Port A input pins.
0 = Port A pullups are disabled.

12.3.10 Reduced Drive Register (RDRIV)

Address: Base + $___D

	BIT 7	6	5	4	3	2	1	BIT 0
Read:	RDPK	0	0	RDPE	0	0	RDPB	RDPA
Write:								
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 12-12 Reduced Drive Register (RDRIV)

Read: anytime (provided this register is in the map)
Write: anytime (provided this register is in the map)

This register is used to select reduced drive for the pins associated with the core ports. This gives reduced power consumption and reduced RFI with a slight increase in transition time (depending on loading). This feature would be used on ports which have a light loading. The reduced drive function is independent of which function is being used on a particular port.

This register is not in the on-chip map in emulation and peripheral modes.

RDPK - Reduced Drive of Port K
1 = All Port K output pins have reduced drive enabled.
0 = All Port K output pins have full drive enabled.

RDPE - Reduced Drive of Port E
1 = All Port E output pins have reduced drive enabled.
0 = All Port E output pins have full drive enabled.

RDPB - Reduced Drive of Port B
1 = All Port B output pins have reduced drive enabled.
0 = All Port B output pins have full drive enabled.

RDPA - Reduced Drive of Ports A
1 = All Port A output pins have reduced drive enabled.
0 = All Port A output pins have full drive enabled.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.3.11 External Bus Interface Control Register (EBICTL)

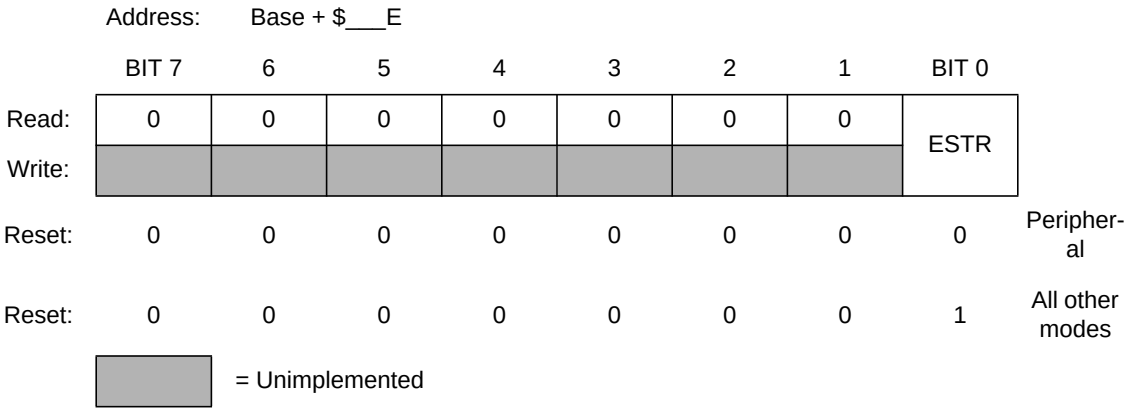


Figure 12-13 External Bus Interface Control Register (EBICTL)

Read: anytime (provided this register is in the map)
Write: refer to individual bit descriptions below

The EBICTL register is used to control miscellaneous functions (i.e. stretching of external E clock).

This register is not in the on-chip map in peripheral mode.

ESTR - E clock Stretches

This control bit determines whether the E clock behaves as a simple free-running clock or as a bus control signal that is active only for external bus cycles.

Normal and Emulation: write once

Special: write anytime

1 = E stretches high during stretch cycles and low during non-visible internal accesses.
0 = E never stretches (always free running).

This bit has no effect in single chip modes.

12.3.12 IRQ Control Register (IRQCR)

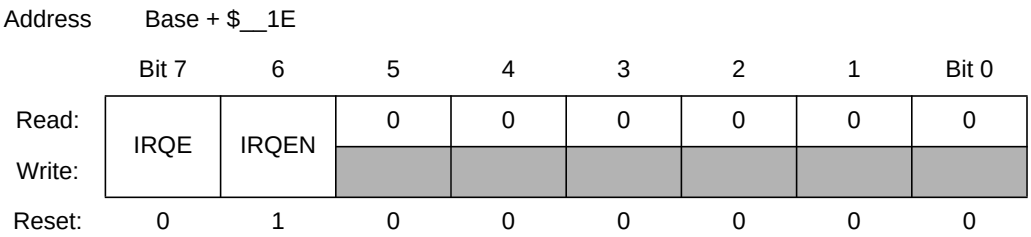


Figure 12-14 IRQ Control Register (IRQCR)

Read: see individual bit descriptions below
Write: see individual bit descriptions below

IRQE - IRQ select edge sensitive only

Special: read or write anytime

Normal: read anytime, write once

Emulation: read anytime, write never

1 = IRQ configured to respond only to falling edges. Falling edges on the IRQ pin will be detected anytime IRQE = 1 and will be cleared only upon a reset or the servicing of the IRQ interrupt (i.e. vector = $FFF2).

0 = IRQ configured for low level recognition

IRQEN - External IRQ enable

Normal, emulation, and special modes: read or write anytime

1 = External IRQ pin is connected to interrupt logic.

0 = External IRQ pin is disconnected from interrupt logic

NOTE: *In this state the edge detect latch is disabled.*

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.3.13 Reserved Registers.

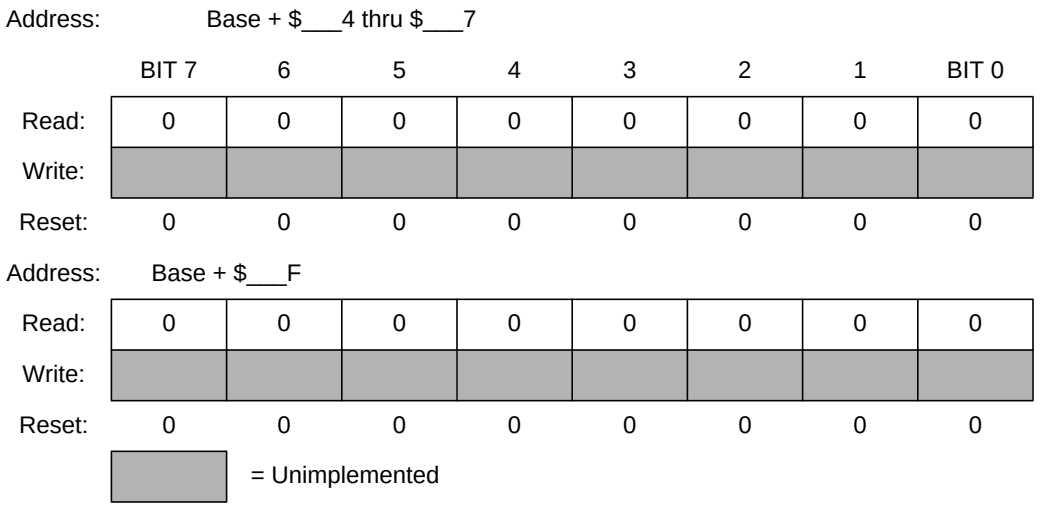


Figure 12-15 Reserved Registers

These register locations are not used (reserved). All unused registers and bits in this block return logic zeros when read. Writes to these registers have no effect.

These registers are not in the on-chip map in peripheral mode.

12.3.14 Port K Data Register (PORTK).

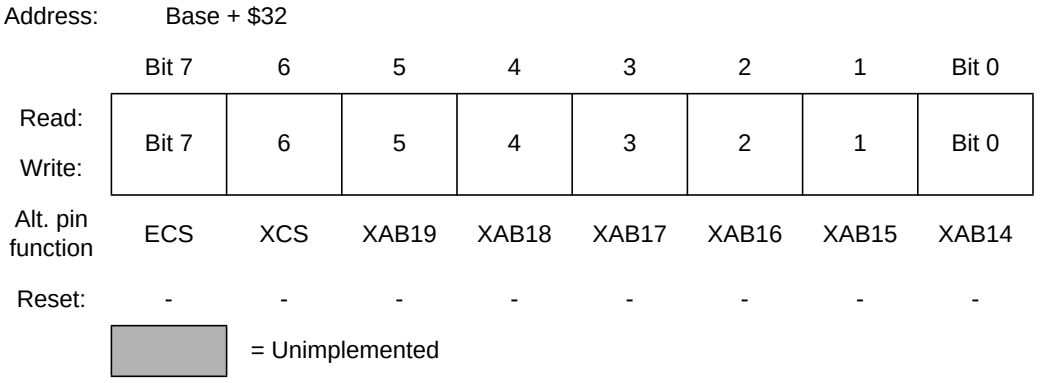


Figure 12-16 Port K Data Register (PORTK)

Read: anytime
Write: anytime

This port is associated with the internal memory expansion emulation pins. When the port is not enabled to emulate the internal memory expansion, the port pins are used as general-purpose I/O. When Port K is operating as a general purpose I/O port, DDRK determines the primary direction for each Port K pin. A “1” causes the associated port pin to be an output and a “0” causes the associated pin to be a

high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTK register. If the DDR bit is zero (input) the buffered pin input is read. If the DDR bit is one (output) the output of the port data register is read.This register is not in the map in peripheral or expanded modes while the EMK control bit in MODE register is set.

When inputs, these pins can be selected to be high impedance or pulled up, based upon the state of the PUPKE bit in the PUCR register.

Bit 7- Port K bit 7.

This bit is used as an emulation chip select signal for the emulation of the internal memory expansion, or as general purpose I/O, depending upon the state of the EMK bit in the MODE register. While this bit is used as a chip select, the external bit will return to its de-asserted state (vdd) for approximately 1/4 cycle just after the negative edge of ECLK, unless the external access is stretched and ECLK is free-running (ESTR bit in EBICTL = 0). See the STAR12v1.5 MMC spec for additional details on when this signal will be active.

Bit 6 — Port K bit 6.

This bit is used as an external chip select signal for most external accesses that are not selected by $\overline{\text{ECS}}$ (see the MMC spec for more details), depending upon the state the of the EMK bit in the MODE register. While this bit is used as a chip select, the external pin will return to its de-asserted state (vdd) for approximately 1/4 cycle just after the negative edge of ECLK, unless the external access is stretched and ECLK is free-running (ESTR bit in EBICTL = 0).

Bit 5 - Bit 0 — Port K bits 5 - 0.

These six bits are used to determine which Flash/ROM or external memory array page is being accessed. They can be viewed as expanded addresses XAB19 - XAB14 of the 20-bit address used to access up to1M byte internal Flash/ROM or external memory array. Alternatively, these bits can be used for general purpose I/O depending upon the state of the EMK bit in the MODE register.

12.3.15 Port K Data Direction Register (DDRK)

Address: Base + $33

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	Bit 7	Bit 6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 12-17 Port K Data Direction Register (DDRK)

Read: anytime.
Write: anytime.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

This register determines the primary direction for each port K pin configured as general-purpose I/O. This register is not in the map in peripheral or expanded modes while the EMK control bit in MODE register is set.

DDRK 7-0 - The Data Direction Port K.
1 = Associated pin is an output
0 = Associated pin is a high-impedance input

CAUTION:

It is unwise to write PORTK and DDRK as a word access. If you are changing Port K pins from inputs to outputs, the data may have extra transitions during the write. It is best to initialize PORTK before enabling as outputs.

CAUTION:

To ensure that you read the correct value from the PORTK pins, always wait at least one cycle after writing to the DDRK register before reading from the PORTK register.

12.4 Operation

There are four main sub-blocks within the MEBI: external bus control, external data bus interface, control and registers.

12.4.1 External Bus Control

The external bus control generates the miscellaneous control functions (pipe signals, ECLK, $\overline{\text{LSTRB}}$ and R/$\overline{\text{W}}$) that will be sent external on Port E, bits 6-2. It also generates the external addresses.

12.4.2 External Data Bus Interface

The external data bus interface block manages data transfers from/to the external pins to/from the internal read and write data buses. This block selectively couples 8-bit or 16-bit data to the internal data bus to implement a variety of data transfers including 8-bit, 16-bit, 16-bit swapped and 8-bit external to 16-bit internal accesses. Modes, addresses, chip selects, etc. affect the type of accesses performed during each bus cycle.

12.4.3 Control

The control block generates the register read/write control signals and miscellaneous port control signals.

12.4.4 Registers

The register block includes the fourteen 8-bit registers and five reserved register locations associated with the MEBI sub-block.

12.4.5 External System Pin Functional Descriptions

In typical SoC implementations, the MEBI sub-block of the Core interfaces directly with external system pins. **Table 12-4** below outlines the pin names and functions and gives a brief description of their operation.

Table 12-4 External System Pins Associated With MEBI

Pin Name	Pin Functions	Description
PA7/A15/D15/D7 thru PA0/A8/D8/D0	PA7 - PA0	General purpose I/O pins, see PORTA and DDRA registers.
	A15 - A8	High-order address lines multiplexed during ECLK low. Outputs except in special peripheral mode where they are inputs from an external tester system.
	D15 - D8	High-order bidirectional data lines multiplexed during ECLK high in expanded wide modes, peripheral mode & visible internal accesses (IVIS=1) in emulation expanded narrow mode. Direction of data transfer is generally indicated by R/$\overline{W}$.
	D15/D7 thru D8/D0	Alternate high-order and low-order bytes of the bidirectional data lines multiplexed during ECLK high in expanded narrow modes and narrow accesses in wide modes. Direction of data transfer is generally indicated by R/W.
PB7/A7/D7 thru PB0/A0/D0	PB7 - PB0	General purpose I/O pins, see PORTB and DDRB registers.
	A7 - A0	Low-order address lines multiplexed during ECLK low. Outputs except in special peripheral mode where they are inputs from an external tester system.
	D7 - D0	Low-order bidirectional data lines multiplexed during ECLK high in expanded wide modes, peripheral mode & visible internal accesses (with IVIS=1) in emulation expanded narrow mode. Direction of data transfer is generally indicated by R/$\overline{W}$.
PE7/ NOACC	PE7	General purpose I/O pin, see PORTE and DDRE registers.
	NOACC	CPU No Access output. Indicates whether the current cycle is a free cycle. Only available in expanded modes.
PE6/IPIPE1/ MODB/CLKTO	MODB	At the rising edge of $\overline{\text{RESET}}$, the state of this pin is registered into the MODB bit to set the mode.
	PE6	General purpose I/O pin, see PORTE and DDRE registers.
	IPIPE1	Instruction pipe status bit 1, enabled by PIPOE bit in PEAR.
	CLKTO	System Clock Test Output. Only available in special modes. PIPOE=1 overrides this function. The enable for this function is in the clock module.
PE5/IPIPE0/ MODA	MODA	At the rising edge on $\overline{\text{RESET}}$, the state of this pin is registered into the MODA bit to set the mode.
	PE5	General purpose I/O pin, see PORTE and DDRE registers.
	IPIPE0	Instruction pipe status bit 0, enabled by PIPOE bit in PEAR.
PE4/ECLK	PE4	General purpose I/O pin, see PORTE and DDRE registers.
	ECLK	Bus timing reference clock, can operate as a free-running clock at the system clock rate or to produce one low-high clock per visible access, with the high period stretched for slow accesses. ECLK is controlled by the NECLK bit in PEAR, the IVIS bit in MODE and the ESTR bit in EBICTL.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table 12-4 External System Pins Associated With MEBI

Pin Name	Pin Functions	Description
PE3/$\overline{\text{LSTRB}}$/ $\overline{\text{TAGLO}}$	PE3	General purpose I/O pin, see PORTE and DDRE registers.
	$\overline{\text{LSTRB}}$	Low strobe bar, 0 indicates valid data on D7-D0.
	SZ8	In peripheral mode, this pin is an input indicating the size of the data transfer (0=16-bit; 1=8-bit).
	$\overline{\text{TAGLO}}$	In expanded wide mode or emulation narrow modes, when instruction tagging is on and low strobe is enabled, a 0 at the falling edge of E tags the low half of the instruction word being read into the instruction queue.
PE2/R/$\overline{\text{W}}$	PE2	General purpose I/O pin, see PORTE and DDRE registers.
	R/$\overline{\text{W}}$	Read/write, indicates the direction of internal data transfers. This is an output except in peripheral mode where it is an input.
PE1/$\overline{\text{IRQ}}$	PE1	General purpose input-only pin, can be read even if $\overline{\text{IRQ}}$ enabled.
	$\overline{\text{IRQ}}$	Maskable interrupt request, can be level sensitive or edge sensitive.
PE0/$\overline{\text{XIRQ}}$	PE0	General purpose input-only pin.
	$\overline{\text{XIRQ}}$	Non-maskable interrupt input.
PK7/$\overline{\text{ECS}}$	PK7	General purpose I/O pin, see PORTK and DDRK registers.
	$\overline{\text{ECS}}$	emulation chip select
PK6/$\overline{\text{XCS}}$	PK6	General purpose I/O pin, see PORTK and DDRK registers.
	$\overline{\text{XCS}}$	external data chip select
PK5/X19 thru PK0/X14	PK5 - PK0	General purpose I/O pins, see PORTK and DDRK registers.
	X19 - X14	Memory expansion addresses
BKGD/MODC/ $\overline{\text{TAGHI}}$	MODC	At the rising edge on $\overline{\text{RESET}}$, the state of this pin is registered into the MODC bit to set the mode. (This pin always has an internal pullup.)
	BKGD	Pseudo-open-drain communication pin for the single-wire background debug mode. There is an internal pullup resistor on this pin.
	TAGHI	When instruction tagging is on, a 0 at the falling edge of E tags the high half of the instruction word being read into the instruction queue.

12.4.6 Detecting Access Type from External Signals

The external signals $\overline{\text{LSTRB}}$, R/$\overline{\text{W}}$, and AB0 indicate the type of bus access that is taking place. Accesses to the internal RAM module are the only type of access that would produce $\overline{\text{LSTRB}}$=AB0=1, because the internal RAM is specifically designed to allow misaligned 16-bit accesses in a single cycle. In these cases

the data for the address that was accessed is on the low half of the data bus and the data for address+1 is on the high half of the data bus.

Table 12-5 Access Type vs. Bus Control Pins

$\overline{\text{LSTRB}}$	AB0	R/$\overline{\text{W}}$	Type of Access
1	0	1	8-bit read of an even address
0	1	1	8-bit read of an odd address
1	0	0	8-bit write of an even address
0	1	0	8-bit write of an odd address
0	0	1	16-bit read of an even address
1	1	1	16-bit read of an odd address (low/high data swapped)
0	0	0	16-bit write to an even address
1	1	0	16-bit write to an odd address (low/high data swapped)

12.4.7 Stretched Bus Cycles

In order to allow fast internal bus cycles to coexist in a system with slower external memory resources, the STAR12 supports the concept of stretched bus cycles (module timing reference clocks for timers and baud rate generators are not affected by this stretching). Control bits in the MISC register in the MMC sub-block of the Core specify the amount of stretch (0, 1, 2, or 3 periods of the internal bus-rate clock). While stretching, the CPU state machines are all held in their current state. At this point in the CPU bus cycle, write data would already be driven onto the data bus so the length of time write data is valid is extended in the case of a stretched bus cycle Read data would not be captured by the system until the E clock falling edge. In the case of a stretched bus cycle, read data is not required until the specified setup time before the falling edge of the stretched E clock. The external address, chip selects, and R/W signals remain valid during the period of stretching (throughout the stretched E high time)

12.4.8 Modes of Operation

The MEBI sub-block controls the mode of the Core operation through the use of the BKGD, MODB and MODA external system pins which are captured into the MODC, MODB and MODA controls bits, respectively, at the rising edge of the system $\overline{\text{RESET}}$ pin. The setup and hold times associated with these pins are given in **Table 12-6** below.

Table 12-6 Mode Pin Setup and Hold Timing

Characteristic	Timing
Mode programming setup time (time before reset is detected high that mode pins must hold their state to guarantee the proper state is entered)	2 bus clock cycles

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Table 12-6 Mode Pin Setup and Hold Timing

Characteristic	Timing
Mode programming hold (time after reset is detected high that mode pins must hold their state to guarantee the proper state is entered)	0 ns

The four 8-bit Ports (A, B, E and K) associated with the MEBI sub-block can serve as general purpose I/O pins or alternatively as the address, data and control signals for a multiplexed expansion bus. Address and data are multiplexed on Ports A and B. The control pin functions are dependent on the operating mode and the control registers PEAR and MODE. The initial state of bits in the PEAR and MODE registers are also established during reset to configure various aspects of the expansion bus. After the system is running, application software can access the PEAR and MODE registers to modify the expansion bus configuration.

Some aspects of Port E are not mode dependent. Bit 1 of Port E is a general purpose input or the $\overline{\text{IRQ}}$ interrupt input. $\overline{\text{IRQ}}$ can be enabled by bits in the CPU condition code register but it is inhibited at reset so this pin is initially configured as a simple input with a pullup. Bit-0 of Port E is a general purpose input or the $\overline{\text{XIRQ}}$ interrupt input. $\overline{\text{XIRQ}}$ also can be enabled by bits in the CPU condition code register but it is inhibited at reset so this pin is initially configured as a simple input with a pullup. The ESTR bit in the EBICTL register is set to one by reset in any user mode. This assures that the reset vector can be fetched even if it is located in an external slow memory device. The PE6/MODB/IPIPE1 and PE5/MODA/IPIPE0 pins act as high-impedance mode select inputs during reset.

The following subsections discuss the default bus setup and describe which aspects of the bus can be changed after reset on a per mode basis.

12.4.8.1 Special Single Chip Mode

When the system is reset in this mode, the background debug mode is enabled and “active”. The system does not fetch the reset vector and execute application code as it would in other modes. Instead, the active background mode is in control of CPU execution and BDM firmware is waiting for additional serial commands through the BKGD pin. When a serial command instructs the system to return to normal execution, the system will be configured as described below unless the reset states of internal control registers have been changed through background commands after the system was reset.

There is no external expansion bus after reset in this mode. Ports A and B are initially simple bidirectional I/O pins that are configured as high-impedance inputs with internal pullups enabled; however, writing to the mode select bits in the MODE register (which is allowed in special modes) can change this after reset. All of the Port E pins (except PE4/ECLK) are initially configured as general purpose high-impedance inputs with pullups enabled. PE4/ECLK is configured as the E clock output in this mode.

The pins associated with Port E bits 6, 5, 3, and 2 cannot be configured for their alternate functions IPIPE1, IPIPE0, $\overline{\text{LSTRB}}$, and R/$\overline{\text{W}}$, respectively, while the system is in single chip modes.The associated control bits PIPOE, LSTRE and RDWE are reset to zero. Writing the opposite value into these bits in this mode does not change the operation of the associated Port E pins.

Port E, bit 4 can be configured for a free-running E clock output by clearing NECLK=0. Typically, the only use for an E clock output while the system is in single chip modes would be to get a constant speed clock for use in the external application system.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.4.8.2 Emulation Expanded Narrow Mode

Expanded narrow modes are intended to allow connection of single 8-bit external memory devices for lower cost systems that do not need the performance of a full 16-bit external data bus. Accesses to internal resources that have been mapped external (i.e. PORTA, PORTB, DDRA, DDRB, PORTE, DDRE, PEAR, PUCR, RDRIV) will be accessed with a 16-bit data bus on Ports A and B. Accesses of 16-bit external words to addresses which are normally mapped external will be broken into two separate 8-bit accesses using Port A as an 8-bit data bus. Internal operations continue to use full 16-bit data paths. They are only visible externally as 16-bit information if IVIS=1.

Ports A and B are configured as multiplexed address and data output ports. During external accesses, address A15, data D15 and D7 are associated with PA7, address A0 is associated with PB0 and data D8 and D0 are associated with PA0. During internal visible accesses and accesses to internal resources that have been mapped external, address A15 and data D15 is associated with PA7 and address A0 and data D0 is associated with PB0.

The bus control related pins in Port E (PE7/NOACC, PE6/MODB/IPIPE1, PE5/MODA/IPIPE0, PE4/ECLK, PE3/$\overline{\text{LSTRB}}$/$\overline{\text{TAGLO}}$, and PE2/R/$\overline{\text{W}}$) are all configured to serve their bus control output functions rather than general purpose I/O. Notice that writes to the bus control enable bits in the PEAR register in emulation mode are restricted.

The main difference between emulation modes and normal modes is that some of the bus control and system control signals cannot be written in emulation modes.

12.4.8.3 Peripheral Mode

This mode is intended for Motorola factory testing of the system. In this mode, the CPU is inactive and an external (tester) bus master drives address, data and bus control signals in through Ports A, B and E. In effect, the whole system acts as if it was a peripheral under control of an external CPU. This allows faster testing of on-chip memory and peripherals than previous testing methods. Since the mode control register is not accessible in peripheral mode, the only way to change to another mode is to reset the system into a different operating mode.

12.4.8.4 Emulation Expanded Wide Mode

In expanded wide modes, Ports A and B are configured as a 16-bit multiplexed address and data bus and Port E provides bus control and status signals. These signals allow external memory and peripheral devices to be interfaced to the system. These signals can also be used by a logic analyzer to monitor the progress of application programs.

The bus control related pins in Port E (PE7/NOACC, PE6/MODB/IPIPE1, PE5/MODA/IPIPE0, PE4/ECLK, PE3/$\overline{\text{LSTRB}}$/$\overline{\text{TAGLO}}$, and PE2/R/$\overline{\text{W}}$) are all configured to serve their bus control output functions rather than general purpose I/O. Notice that writes to the bus control enable bits in the PEAR register in emulation mode are restricted.

The main difference between emulation modes and normal modes is that some of the bus control and system control signals cannot be written in emulation modes.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

12.4.8.5 Normal Single Chip Mode

There is no external expansion bus in this mode. All pins of Ports A, B and K are configured as general purpose I/O pins. Port E bits 1 and 0 are available as general purpose input only pins with internal pullups and the other remaining pins are bidirectional I/O pins that are initially configured as high-impedance inputs with internal pullups enabled.

The pins associated with Port E bits 6, 5, 3, and 2 cannot be configured for their alternate functions IPIPE1, IPIPE0, $\overline{\text{LSTRB}}$, and R/$\overline{\text{W}}$ while the system is in single chip modes. The associated control bits PIPOE, LSTRE, and RDWE, respectively, are reset to zero. Writing the opposite state into them in this mode does not change the operation of the associated Port E pins.

In normal single chip mode, the MODE register is writable one time. This allows a user program to change the bus mode to narrow or wide expanded mode and/or turn on visibility of internal accesses.

Port E, bit 4 can be configured for a free-running E clock output by clearing NECLK=0. Typically, the only use for an E clock output while the system is in single chip modes would be to get a constant speed clock for use in the external application system.

12.4.8.6 Normal Expanded Narrow Mode

This mode is used for lower cost production systems that use 8-bit wide external EPROMs or RAMs. Such systems take extra bus cycles to access 16-bit locations but this may be preferred over the extra cost of additional external memory devices.

Ports A and B are configured as a 16-bit address bus and Port A is multiplexed with data. Internal visibility is not available in this mode because the internal cycles would need to be split into two 8-bit cycles.

Since the PEAR register can only be written one time in this mode, use care to set all bits to the desired states during the single allowed write.

The PE3/$\overline{\text{LSTRB}}$ pin is always a general purpose I/O pin in normal expanded narrow mode. Although it is possible to write the LSTRE bit in PEAR to “1” in this mode, the state of LSTRE is overridden and Port E bit 3 cannot be reconfigured as the $\overline{\text{LSTRB}}$ output.

It is possible to enable the pipe status signals on Port E bits 6 and 5 by setting the PIPOE bit in PEAR, but it would be unusual to do so in this mode. $\overline{\text{LSTRB}}$ would also be needed to fully understand system activity. Development systems where pipe status signals are monitored would typically use special test mode or occasionally emulation expanded narrow mode.

The PE4/ECLK pin is initially configured as ECLK output with stretch. The E clock output function depends upon the settings of the NECLK bit in the PEAR register, the IVIS bit in the MODE register and the ESTR bit in the EBICTL register. In normal expanded narrow mode, the E clock is available for use in external select decode logic or as a constant speed clock for use in the external application system.

The PE2/R/$\overline{\text{W}}$ pin is initially configured as a general purpose input with a pullup but this pin can be reconfigured as the R/$\overline{\text{W}}$ bus control signal by writing “1” to the RDWE bit in PEAR. If the expanded narrow system includes external devices that can be written such as RAM, the RDWE bit would need to be set before any attempt to write to an external location. If there are no writable resources in the external system, PE2 can be left as a general purpose I/O pin.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

12.4.8.7 Special Test Mode

In expanded wide modes, Ports A and B are configured as a 16-bit multiplexed address and data bus and Port E provides bus control and status signals. In special test mode, the write protection of many control bits is lifted so that they can be thoroughly tested without needing to go through reset.

12.4.8.8 Normal Expanded Wide Mode

In expanded wide modes, Ports A and B are configured as a 16-bit multiplexed address and data bus and Port E bit 4 is configured as the E clock output signal. These signals allow external memory and peripheral devices to be interfaced to the system.

Port E pins other than PE4/ECLK are configured as general purpose I/O pins (initially high-impedance inputs with internal pullup resistors enabled). Control bits PIPOE, NECLK, LSTRE, and RDWE in the PEAR register can be used to configure Port E pins to act as bus control outputs instead of general purpose I/O pins.

It is possible to enable the pipe status signals on Port E bits 6 and 5 by setting the PIPOE bit in PEAR, but it would be unusual to do so in this mode. Development systems where pipe status signals are monitored would typically use the emulation variation of this mode.

The Port E bit 2 pin can be reconfigured as the R/$\overline{W}$ bus control signal by writing “1” to the RDWE bit in PEAR. If the expanded system includes external devices that can be written, such as RAM, the RDWE bit would need to be set before any attempt to write to an external location. If there are no writable resources in the external system, PE2 can be left as a general purpose I/O pin.

The Port E bit 3 pin can be reconfigured as the $\overline{LSTRB}$ bus control signal by writing “1” to the LSTRE bit in PEAR. The default condition of this pin is a general purpose input because the $\overline{LSTRB}$ function is not needed in all expanded wide applications.

The Port E bit 4 pin is initially configured as ECLK output with stretch. The E clock output function depends upon the settings of the NECLK bit in the PEAR register, the IVIS bit in the MODE register and the ESTR bit in the EBICTL register. The E clock is available for use in external select decode logic or as a constant speed clock for use in the external application system.

12.4.9 Internal Visibility

Internal visibility is available when the system is operating in expanded wide modes, special test mode, or emulation narrow mode. It is not available in single-chip, peripheral or normal expanded narrow modes. Internal visibility is enabled by setting the IVIS bit in the MODE register.

If an internal access is made while E, R/$\overline{W}$, and $\overline{LSTRB}$ are configured as bus control outputs and internal visibility is off (IVIS=0), E will remain low for the cycle, R/$\overline{W}$ will remain high, and address, data and the $\overline{LSTRB}$ pins will remain at their previous state.

When internal visibility is enabled (IVIS=1), certain internal cycles will be blocked from going external to prevent possible corruption of external devices. Specifically, during cycles when the BDM is selected, R/$\overline{W}$ will remain high, data will maintain its previous state, and address and $\overline{LSTRB}$ pins will be updated

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

with the internal value. During CPU no access cycles when the BDM is not driving, R/$\overline{W}$ will remain high, and address, data and the $\overline{LSTRB}$ pins will remain at their previous state.

12.4.10 Secure Mode

When the system is operating in a secure mode, internal visibility is not available (i.e. IVIS=1 has no effect). Also, the IPIPE signals will not be visible, regardless of operating mode. IPIPE1-IPIPE0 will display zeroes if they are enabled. In addition, the MOD bits in the MODE control register cannot be written.

12.5 Low-Power Options

The MEBI does not contain any user-controlled options for reducing power consumption. The operation of the MEBI in low-power modes is discussed in the following subsections.

12.5.1 Run Mode

The MEBI does not contain any options for reducing power in run mode; however, the external addresses are conditioned with expanded mode to reduce power in single chip modes.

12.5.2 Wait Mode

The MEBI does not contain any options for reducing power in wait mode.

12.5.3 Stop Mode

The MEBI will cease to function during execution of a CPU STOP instruction.

12.6 Motorola Internal Information

This subsection details information about the MEBI sub-block that is for Motorola use only and should not be published in any form outside of Motorola.

12.6.1 Peripheral Mode Operation

The only way to enter peripheral mode is via reset with the pins configured as shown in **Table 12-7**. The only way to exit peripheral mode is to change the mode pin configuration and pull reset. It is not possible to enter/exit peripheral mode by writing the MODx bits in the MODE register.

Table 12-7 Peripheral Mode Pin Configuration

MODC (BKGD)	MODB (PE6)	MODA (PE5)
1	1	0

Peripheral mode is a special mode immediately out of reset. It may be changed to a normal mode by writing the PNORME bit in the MTST1 register of the MMC sub-block to ‘1’.

In peripheral mode, the direction of the address and data buses is reversed compared to other modes of operation. Address, R/$\overline{W}$ and SZ8 all come from the external test system and drive the bus interface pins of Ports A, B and E of the system. The data bus is configured to pass data directly through Ports A and B to the internal data bus. Accesses are all initiated by the external test system.

The burden of deciding which port to access for 8-bit data or swapped data is the responsibility of the external test system. The MEBI does not modify peripheral mode accesses in any way. Misaligned 16-bit accesses are not allowed to blocks that require two cycles to complete such as system peripherals. Misaligned 16-bit accesses are allowed to blocks that can handle fast transfers such as a RAM memory block.

12.6.2 Special Test Clock

When the *peri_test_clk_enable* signal at the Core interface is asserted in special modes, the *peri_test_clk* signal will be driven out on Port E, bit 6 when PIPOE=0.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 13 Breakpoint (BKP)

This section describes the functionality of the Breakpoint (BKP) sub-block of the Core.

13.1 Overview

The Breakpoint sub-block of the Core provides for hardware breakpoints that are used to debug software on the CPU by comparing actual address and data values to predetermined data in setup registers. A successful comparison will place the CPU in Background Debug Mode or initiate a software interrupt (SWI).

The Breakpoint sub-block contains two modes of operation:

- Dual Address Mode, where a match on either of two addresses will cause the system to enter Background Debug Mode or initiate a Software Interrupt (SWI).
- Full Breakpoint Mode, where a match on address and data will cause the system to enter Background Debug Mode or initiate a Software Interrupt (SWI).

There are two types of breakpoints, forced and tagged. Forced breakpoints occur at the next instruction boundary if a match occurs and tagged breakpoints allow for breaking just before a specific instruction executes. Tagged breakpoints will only occur on addresses. Tagging on data is not allowed; however, if this occurs nothing will happen within the BKP.

The BKP allows breaking within a 256 byte address range and/or within expanded memory. It allows matching of the data as well as the address and to match 8-bit or 16-bit data. Forced breakpoints can match on a read or a write cycle.

13.1.1 Features

- Full or Dual Breakpoint Mode
  - Compare on address and data (Full)
  - Compare on either of two addresses (Dual)
- BDM or SWI Breakpoint
  - Enter BDM on breakpoint (BDM)
  - Execute SWI on breakpoint (SWI)
- Tagged or Forced Breakpoint
  - Break just before a specific instruction will begin execution (TAG)
  - Break on the first instruction boundary after a match occurs (Force)
- Single, Range or Page address compares
  - Compare on address (Single)
  - Compare on address 256 byte (Range)

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

  - Compare on any 16K Page (Page)
- Compare address on read or write on forced breakpoints
- High and/or low byte data compares

13.1.2 Block Diagram

A block diagram of the Breakpoint sub-block is shown in **Figure 13-1** below. The Breakpoint contains three main sub-blocks: the Register Block, the Compare Block and the Control Block. The Register Block consists of the eight registers that make up the Breakpoint register space. The Compare Block performs all required address and data signal comparisons. The Control Block generates the signals for the CPU for the tag high, tag low, force SWI and force BDM functions. In addition, it generates the register read and write signals and the comparator block enable signals.

NOTE: There is a two cycle latency for address compares for forces, a two cycle latency for write data compares, and a three cycle latency for read data compares.

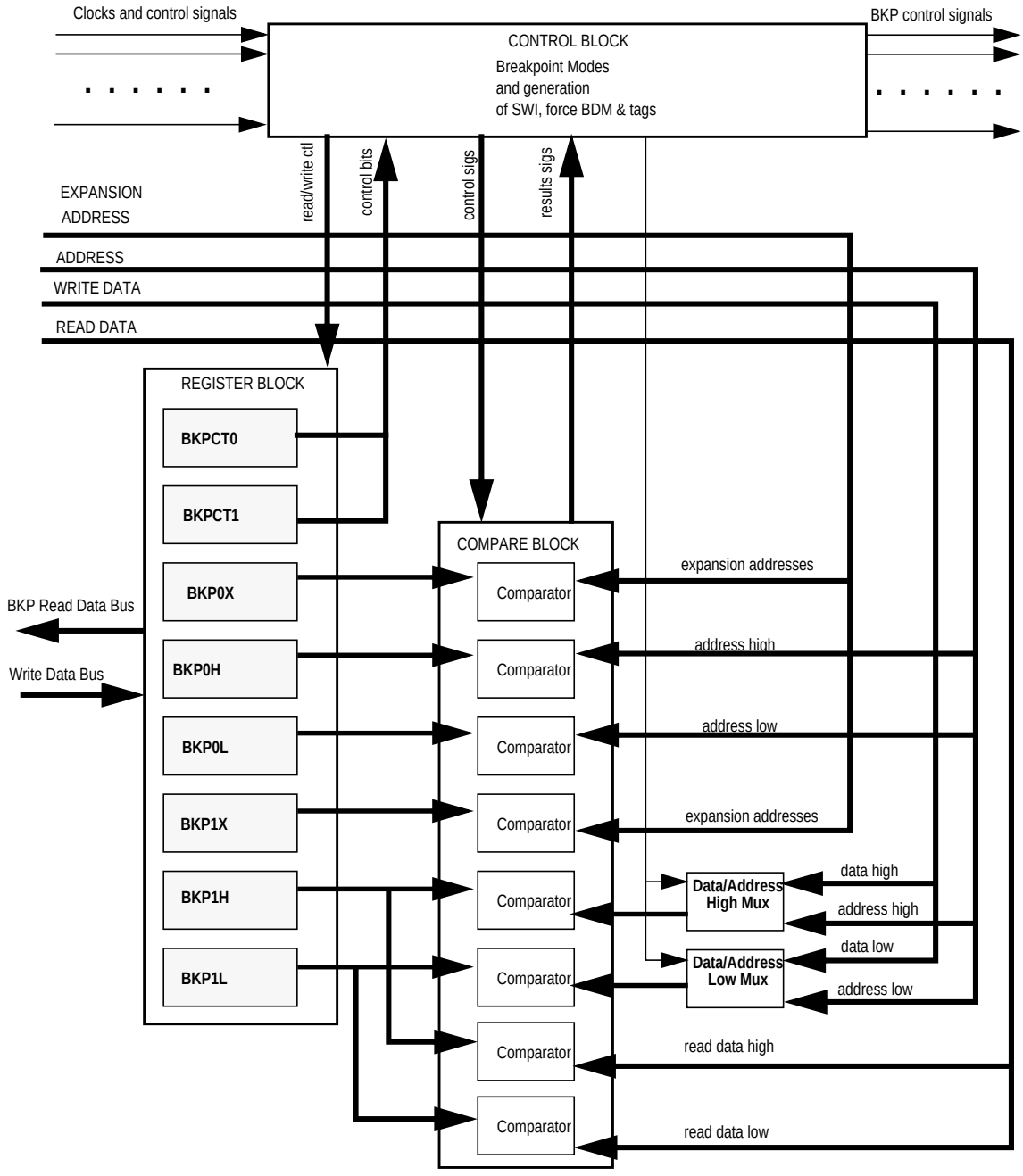


Figure 13-1 Breakpoint Block Diagram

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

13.2 Interface Signals

All interfacing with the Breakpoint sub-block is done within the Core.

13.3 Registers

A summary of the registers associated with the Breakpoint sub-block is shown in **Figure 13-2** below. Detailed descriptions of the registers and bits are given in the subsections that follow.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$0028	BKPCT0	read	BKEN	BKFULL	BKBDM	BKTAG	0	0	0	0
		write								
$0029	BKPCT1	read	BK0MBH	BK0MBL	BK1MBH	BK1MBL	BK0RWE	BK0RW	BK1RWE	BK1RW
		write								
$002A	BKP0X	read	0	0	BK0V5	BK0V4	BK0V3	BK0V2	BK0V1	BK0V0
		write								
$002B	BKP0H	read	Bit 15	14	13	12	11	10	9	Bit 8
		write								
$002C	BKP0L	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								
$002D	BKP1X	read	0	0	BK1V5	BK1V4	BK1V3	BK1V2	BK1V1	BK1V0
		write								
$002E	BKP1H	read	Bit 15	14	13	12	11	10	9	Bit 8
		write								
$002F	BKP1L	read	Bit 7	6	5	4	3	2	1	Bit 0
		write								

= Unimplemented X = Indeterminate

Figure 13-2 Breakpoint Register Summary

13.3.1 Breakpoint Control Register 0 (BKPCT0)

Read: anytime
Write: anytime

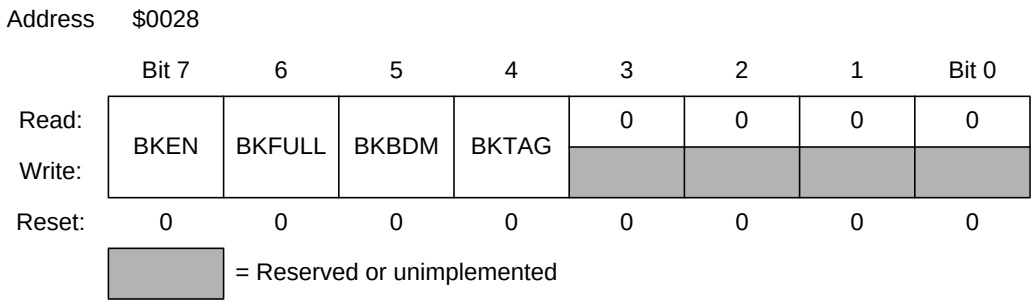


Figure 13-3 Breakpoint Control Register 0 (BKPCT0)

This register is used to set the breakpoint modes.

BKEN - Breakpoint Enable

This bit enables the module
0 = Breakpoint module off
1 = Breakpoint module on

BKFULL - Full Breakpoint Mode Enable

This bit controls whether the breakpoint module is in Dual Mode or Full Mode
0 = Dual Address Mode enabled
1 = Full Breakpoint Mode enabled

BKBDM - Breakpoint Background Debug Mode Enable

This bit determines if the breakpoint causes the system to enter Background Debug Mode(BDM) or initiate a Software Interrupt (SWI)
0 = Go to Software Interrupt on a compare
1 = Go to BDM on a compare

BKTAG — Breakpoint on Tag

This bit controls whether the breakpoint will cause a break on the next instruction boundary (force) or on a match that will be an executable opcode (tagged). Non-executed opcodes cannot cause a tagged breakpoint
0 = On match, break at the next instruction boundary (force)
1 = On match, break if the match is an instruction that will be executed (tagged)

13.3.2 Breakpoint Control Register 1 (BKPCT1)

Read: anytime
Write: anytime

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Address $0029

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	BK0MBH	BK0MBL	BK1MBH	BK1MBL	BK0RWE	BK0RW	BK1RWE	BK1RW
Reset:	0	0	0	0	0	0	0	0

Figure 13-4 Breakpoint Control Register 1 (BKPCT1)

This register is used to configure the functionality of the Breakpoint sub-block within the Core.

BK0MBH:BK0MBL - Breakpoint Mask High Byte and Low Byte for First Address

In Dual or Full Mode, these bits may be used to mask (disable) the comparison of the high and low bytes of the first address breakpoint. The functionality is as given in **Table 13-1** below

Table 13-1 Breakpoint Mask Bits for First Address

BK0MBH:BK0MBL	Address Compare	BKP0X	BKP0H	BKP0L
x:0	Full Address Compare	Yes[1]	Yes	Yes
0:1	256 byte Address Range	Yes(1)	Yes	No
1:1	16K byte Address Range	Yes(1)	No	No

NOTES:

1. If page is selected.

The x:0 case is for a Full Address Compare. When a program page is selected, the full address compare will be based on bits for a 20-bit compare. The registers used for the compare are {BKP0X[5:0],BKP0H[5:0],BKP0L[7:0]}. When a program page is not selected, the full address compare will be based on bits for a 16-bit compare. The registers used for the compare are {BKP0H[7:0],BKP0L[7:0]}.

The 1:0 case is not sensible because it would ignore the high order address and compare the low order and expansion addresses. Logic forces this case to compare all address lines (effectively ignoring the BK0MBH control bit).

The 1:1 case is useful for triggering a breakpoint on any access to a particular expansion page. This only makes sense if a program page is being accessed so that the breakpoint trigger will occur only if BKP0X compares.

BK1MBH:BK1MBL - Breakpoint Mask High Byte and Low Byte of Data (Second Address)

In Dual Mode, these bits may be used to mask (disable) the comparison of the high and/or low bytes of the second address breakpoint. The functionality is as given in **Table 13-2** below.

Table 13-2 Breakpoint Mask Bits for Second Address (Dual Mode)

BK1MBH:BK1MBL	Address Compare	BKP1X	BKP1H	BKP1L
x:0	Full Address Compare	Yes[1]	Yes	Yes
0:1	256 byte Address Range	Yes(1)	Yes	No
1:1	16K byte Address Range	Yes(1)	No	No

NOTES:

1. If page is selected.

The x:0 case is for a Full Address Compare. When a program page is selected, the full address compare will be based on bits for a 20-bit compare. The registers used for the compare are {BKP1X[5:0],BKP1H[5:0],BKP1L[7:0]}. When a program page is not selected, the full address compare will be based on bits for a 16-bit compare. The registers used for the compare are {BKP1H[7:0],BKP1L[7:0]}.

The 1:0 case is not sensible because it would ignore the high order address and compare the low order and expansion addresses. Logic forces this case to compare all address lines (effectively ignoring the BK1MBH control bit).

The 1:1 case is useful for triggering a breakpoint on any access to a particular expansion page. This only makes sense if a program page is being accessed so that the breakpoint trigger will occur only if BKP1X compares.

In Full Mode, these bits may be used to mask (disable) the comparison of the high and/or low bytes of the data breakpoint. The functionality is as given in **Table 13-3** below.

Table 13-3 Breakpoint Mask Bits for Data Breakpoints (Full Mode)

BK1MBH:BK1MBL	Data Compare	BKP1X	BKP1H	BKP1L
0:0	High and Low Byte Compare	No[1]	Yes	Yes
0:1	High Byte	No(1)	Yes	No
1:0	Low Byte	No(1)	No	Yes
1:1	No Compare	No(1)	No	No

NOTES:

1. Expansion addresses for breakpoint 1 are not available in this mode.

BK0RWE - R/$\overline{W}$ Compare Enable

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Enables the comparison of the R/$\overline{W}$ signal for first address breakpoint. This bit is not useful in tagged breakpoints.

0 = R/$\overline{W}$ is not used in the comparisons
1 = R/$\overline{W}$ is used in comparisons

BK0RW - R/$\overline{W}$ Compare Value

When BK0RWE=1, this bit determines the type of bus cycle to match on first address breakpoint. When BK0RWE=0, this bit has no effect.

0 = Write cycle will be matched
1 = Read cycle will be matched

BK1RWE - R/$\overline{W}$ Compare Enable

In Dual Mode, this bit enables the comparison of the R/$\overline{W}$ signal to further specify what causes a match for the second address breakpoint. This bit is not useful on tagged breakpoints or in Full Mode and is therefore a don't care.

0 = R/$\overline{W}$ is not used in comparisons
1 = R/$\overline{W}$ is used in comparisons

BK1RW — R/$\overline{W}$ Compare Value

When BK1RWE=1, this bit determines the type of bus cycle to match on the second address breakpoint.When BK1RWE=0, this bit has no effect.

0 = Write cycle will be matched
1 = Read cycle will be matched

13.3.3 Breakpoint First Address Expansion Register (BKP0X)

Read: anytime
Write: anytime

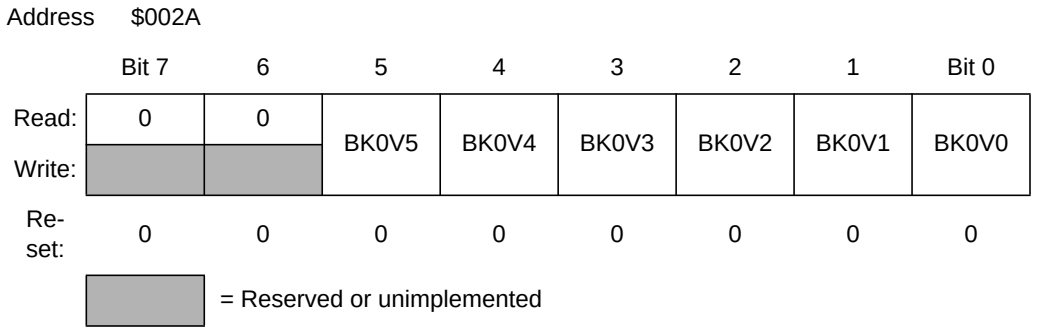


Figure 13-5 Breakpoint First Address Expansion Register (BKP0X)

This register contains the data to be matched against expansion address lines for the first address breakpoint when a page is selected.

BK0V[5:0] - Value of first breakpoint address to be matched in memory expansion space.

13.3.4 Breakpoint First Address High Byte Register (BKP0H)

Read: anytime
Write: anytime

Address	$002B							
	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	Bit 15	14	13	12	11	10	9	Bit 8
Re-set:	0	0	0	0	0	0	0	0

Figure 13-6 Breakpoint First Address High Byte Register (BKP0H)

This register is used to set the breakpoint when compared against the high byte of the address.

13.3.5 Breakpoint First Address Low Byte Register (BKP0L)

Read: anytime
Write: anytime

Address	$002C							
	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Re-set:	0	0	0	0	0	0	0	0

Figure 13-7 Breakpoint First Address Low Byte Register (BKP0L)

This register is used to set the breakpoint when compared against the low byte of the address.

13.3.6 Breakpoint Second Address Expansion Register (BKP1X)

Read: anytime
Write: anytime

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Address $002D

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	0	0	BK1V5	BK1V4	BK1V3	BK1V2	BK1V1	BK1V0
Write:								
Re-set:	0	0	0	0	0	0	0	0

= Reserved or unimplemented

Figure 13-8 Breakpoint Second Address Expansion Register (BKP1X)

In Dual Mode, this register contains the data to be matched against expansion address lines for the second address breakpoint when a page is selected. In Full Mode, this register is not used.

BK1V[5:0] - Value of first breakpoint address to be matched in memory expansion space.

13.3.7 Breakpoint Data (Second Address) High Byte Register (BKP1H)

Read: anytime
Write: anytime

Address $002E

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	Bit 15	14	13	12	11	10	9	Bit 8
Write:								
Re-set:	0	0	0	0	0	0	0	0

Figure 13-9 Breakpoint Data High Byte Register (BKP1H)

In Dual Mode, this register is used to compare against the high order address lines. In Full Mode, this register is used to compare against the high order data lines.

13.3.8 Breakpoint Data (Second Address) Low Byte Register (BKP1L)

Read: anytime
Write: anytime

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Address	$002F							
	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

Figure 13-10 Breakpoint Data Low Byte Register (BKP1L)

In Dual Mode, this register is used to compare against the low order address lines. In Full Mode, this register is used to compare against the low order data lines.

13.4 Operation

The Breakpoint sub-block supports two modes of operation: Dual Address Mode and Full Breakpoint Mode. Within each of these modes, forced or tagged breakpoint types can be used. Forced breakpoints occur at the next instruction boundary if a match occurs and tagged breakpoints allow for breaking just before a specific instruction executes. The action taken upon a successful match can be to either place the CPU in Background Debug Mode or to initiate a software interrupt.

13.4.1 Modes of Operation

The Breakpoint can operate in Dual Address Mode or Full Breakpoint Mode. Each of these modes is discussed in the subsections below.

13.4.1.1 Dual Address Mode

When Dual Address Mode is enabled, two address breakpoints can be set. Each breakpoint can cause the system to enter Background Debug Mode or to initiate a software interrupt based upon the state of the BKBDM bit in the BKPCT0 Register being logic one or logic zero, respectively. BDM requests have a higher priority than SWI requests. No data breakpoints are allowed in this mode.

The BKTAG bit in the BKPCT0 register selects whether the breakpoint mode is force or tag. The BKxMBH:L bits in the BKPCT1 register select whether or not the breakpoint is matched exactly or is a range breakpoint. They also select whether the address is matched on the high byte, low byte, both bytes, and/or memory expansion. The BKxRW and BKxRWE bits in the BKPCT1 register select whether the type of bus cycle to match is a read, write, or both when performing forced breakpoints.

13.4.1.2 Full Breakpoint Mode

Full Breakpoint Mode requires a match on address and data for a breakpoint to occur. Upon a successful match, the system will enter Background Debug Mode or initiate a software interrupt based upon the state of the BKBDM bit in the BKPCT0 Register being logic one or logic zero, respectively. BDM requests have a higher priority than SWI requests. R/$\overline{W}$ matches are also allowed in this mode.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

The BKTAG bit in the BKPCT0 register selects whether the breakpoint mode is forced or tagged. If the BKTAG bit is set in BKPCT0, then only address is matched, and data is ignored. The BK0MBH:L bits in the BKPCT1 register select whether or not the breakpoint is matched exactly, is a range breakpoint, or is in page space. The BK1MBH:L bits in the BKPCT1 register select whether the data is matched on the high byte, low byte, or both bytes. The BK0RW and BK0RWE bits in the BKPCT1 register select whether the type of bus cycle to match is a read or a write when performing forced breakpoints. BK1RW and BK1RWE bits in the BKPCT1 register are not used in Full Breakpoint Mode.

13.4.2 Breakpoint Priority

Breakpoint operation is first determined by the state of BDM. If BDM is already active, meaning the CPU is executing out of BDM firmware, Breakpoints are not allowed. In addition, while in BDM trace mode, tagging into BDM is not allowed. If BDM is not active, the Breakpoint will give priority to BDM requests over SWI requests. This condition applies to both forced and tagged breakpoints.

In all cases, BDM related breakpoints will have priority over those generated by the Breakpoint sub-block. This priority includes breakpoints enabled by the $\overline{\text{TAGLO}}$ and $\overline{\text{TAGHI}}$ external pins of the system that interface with the BDM directly and whose signal information passes through and is used by the Breakpoint sub-block.

NOTE: *BDM should not be entered from a breakpoint unless the ENABLE bit is set in the BDM. Even if the ENABLE bit in the BDM is negated, the CPU actually executes the BDM firmware code. It checks the ENABLE and returns if enable is not set. If the BDM is not serviced by the monitor then the breakpoint would be re-asserted when the BDM returns to normal CPU flow.*

There is no hardware to enforce restriction of breakpoint operation if the BDM is not enabled.

13.5 Motorola Internal Information

The Breakpoint sub-block does not contain any information that is considered to be for Motorola use only.

Section 14 Background Debug Mode (BDM)

This section describes the functionality of the Background Debug Mode (BDM) sub-block of the Core.

14.1 Overview

The Background Debug Mode (BDM) sub-block is a single-wire, background debug system implemented in on-chip hardware for minimal CPU intervention. All interfacing with the BDM is done via the BKGD pin.

14.1.1 Features

- Single-wire communication with host development system
- Active out of reset in special single-chip mode
- Nine hardware commands using free cycles, if available, for minimal CPU intervention
- Hardware commands not requiring active BDM
- 15 firmware commands execute from the standard BDM firmware lookup table
- Instruction tagging capability
- Software control of BDM operation during wait mode
- Software selectable clocks
- BDM disabled when secure feature is enabled

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

14.1.2 Block Diagram

The block diagram of the BDM is shown in **Figure 14-1** below.

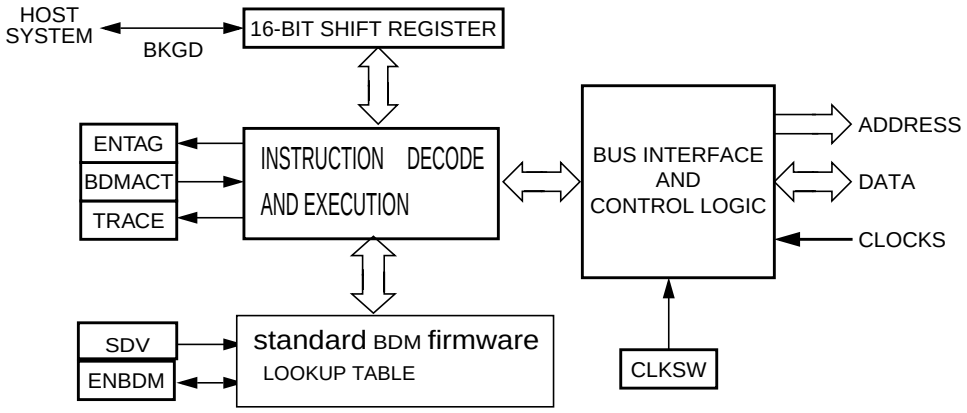


Figure 14-1 BDM Block Diagram

14.2 Interface Signals

A single-wire interface pin is used to communicate with the BDM system. Two additional pins are used for instruction tagging. These pins are part of the Multiplexed External Bus Interface (MEBI) sub-block and all interfacing between the MEBI and BDM is done within the Core interface boundary. The functional descriptions of the pins are provided below for completeness.

- BKGD — Background interface pin
- $\overline{\text{TAGHI}}$ — High byte instruction tagging pin
- $\overline{\text{TAGLO}}$ — Low byte instruction tagging pin

BKGD and $\overline{\text{TAGHI}}$ share the same pin. $\overline{\text{TAGLO}}$ and $\overline{\text{LSTRB}}$ share the same pin.

14.2.1 Background Interface Pin (BKGD)

Debugging control logic communicates with external devices serially via the single-wire background interface pin (BKGD). During reset, this pin is a mode select input which selects between normal and special modes of operation. After reset, this pin becomes the dedicated serial interface pin for the background debug mode.

14.2.2 High Byte Instruction Tagging Pin ($\overline{\text{TAGHI}}$)

This pin is used to tag the high byte of an instruction. When instruction tagging is on, a logic 0 at the falling edge of the external clock (ECLK) tags the high half of the instruction word being read into the instruction queue.

14.2.3 Low Byte Instruction Tagging Pin ($\overline{\text{TAGLO}}$)

This pin is used to tag the low byte of an instruction. When instruction tagging is on and low strobe is enabled, a logic 0 at the falling edge of the external clock (ECLK) tags the low half of the instruction word being read into the instruction queue.

14.3 Registers

A summary of the registers associated with the BDM is shown in **Figure 14-2** below. Registers are accessed by host-driven communications to the BDM hardware using READ_BD and WRITE_BD commands. Detailed descriptions of the registers and associated bits are given in the subsections that follow.

Address	Register Name		Bit 7	6	5	4	3	2	1	Bit 0
$FF00	Reserved	Read:	X	X	X	X	X	X	0	0
		Write:								
$FF01	BDMSTS	Read:	ENBDM	BDMACT	ENTAG	SDV	TRACE	CLKSW	UNSEC	0
		Write:								
$FF02	Reserved	Read:	X	X	X	X	X	X	X	X
		Write:								
$FF03	Reserved	Read:	X	X	X	X	X	X	X	X
		Write:								
$FF04	Reserved	Read:	X	X	X	X	X	X	X	X
		Write:								
$FF05	Reserved	Read:	X	X	X	X	X	X	X	X
		Write:								
$FF06	BDMCCR	Read:	CCR7	CCR6	CCR5	CCR4	CCR3	CCR2	CCR1	CCR0
		Write:								
$FF07	BDMINR	Read:	REG15	REG14	REG13	REG12	REG11	0	0	0
		Write:								

= Unimplemented X = Indeterminate

Figure 14-2 BDM Register Map Summary

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

14.3.1 BDM Status Register

Address: $FF01

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	ENBDM	BDMACT	ENTAG	SDV	TRACE	CLKSW	UNSEC	0
Write:								
Reset:								
Special single-chip mode:	0	1	0	0	0	0	0	0
Special peripheral mode:	0	1	0	0	0	0	0	0
All other modes:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 14-3 BDM Status Register (BDMSTS)

Read: All modes through BDM operation
Write: All modes but subject to the following:

- BDMACT can only be set by BDM hardware upon entry into BDM. It can only be cleared by the standard BDM firmware lookup table upon exit from BDM active mode.
- CLKSW can only be written via BDM hardware or standard BDM firmware write commands.
- All other bits, while writable via BDM hardware or standard BDM firmware write commands, should only be altered by the BDM hardware or standard firmware lookup table as part of BDM command execution.
- ENBDM should only be set via a BDM hardware command if the BDM firmware commands are needed. (This does not apply in Special Single Chip Mode).

ENBDM - Enable BDM

This bit controls whether the BDM is enabled or disabled. When enabled, BDM can be made active to allow firmware commands to be executed. When disabled, BDM cannot be made active but BDM hardware commands are still allowed.

1 = BDM enabled
0 = BDM disabled

NOTE: *ENBDM is set by the firmware immediately out of reset in special single-chip mode. In secure mode, this bit will not be set by the firmware until after the EEPROM and FLASH erase verify tests are complete.*

BDMACT - BDM active status

This bit becomes set upon entering BDM. The standard BDM firmware lookup table is then enabled and put into the memory map. BDMACT is cleared by a carefully timed store instruction in the standard BDM firmware as part of the exit sequence to return to user code and remove the BDM memory from the map.

1 = BDM active
0 = BDM not active

ENTAG - Tagging enable

This bit indicates whether instruction tagging in enabled or disabled. It is set when the TAGGO command is executed and cleared when BDM is entered. The serial system is disabled and the tag function enabled 16 cycles after this bit is written. BDM cannot process serial commands while tagging is active.

1 = Tagging enabled
0 = Tagging not enabled, or BDM active

SDV - Shift data valid

This bit is set and cleared by the BDM hardware. It is set after data has been transmitted as part of a firmware read command or after data has been received as part of a firmware write command. It is cleared when the next BDM command has been received or BDM is exited. SDV is used by the standard BDM firmware to control program flow execution.

1 = Data phase of command is complete
0 = Data phase of command not complete

TRACE - TRACE1 BDM firmware command is being executed

This bit gets set when a BDM TRACE1 firmware command is first recognized. It will stay set as long as continuous back-to-back TRACE1 commands are executed. This bit will get cleared when the next command that is not a TRACE1 command is recognized.

1 = TRACE1 command is being executed
0 = TRACE1 command is not being executed

CLKSW - Clock switch

The CLKSW bit controls which clock the BDM operates with. It is only writable from a hardware BDM command. A 150 cycle delay at the clock speed that is active during the data portion of the command will occur before the new clock source is guaranteed to be active. The start of the next BDM command uses the new clock for timing subsequent BDM communications.

1 = BDM system operates with bus rate
0 = BDM system operates with alternate clock

WARNING:

The BDM will not operate with CLKSW = 0 if the frequency of the alternate clock source, peri_phase_oscdX, is greater than one half of the bus frequency. Please refer to the users guide for the clock generation module to determine if this condition can occur.

UNSEC - Unsecure

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

This bit is only writable in special single chip mode from the BDM secure firmware and always gets reset to zero. It is in a zero state as secure mode is entered so that the secure BDM firmware lookup table is enabled and put into the memory map along with the standard BDM firmware lookup table.

The secure BDM firmware lookup table verifies that the on-chip EEPROM and Flash EEPROM are erased. This being the case, the UNSEC bit is set and the BDM program jumps to the start of the standard BDM firmware lookup table and the secure BDM firmware lookup table is turned off. If the erase test fails, the UNSEC bit will not be asserted.

1 = the system is in a unsecured mode
0 = the system is in a secured mode

WARNING:

When UNSEC is set, security is off and the user can change the state of the secure bits in the on-chip Flash EEPROM. Note that if the user does not change the state of the bits to "unsecured" mode, the system will be secured again when it is next taken out of reset.

14.3.2 BDM CCR Holding Register

Address: $FF06

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	CCR7	CCR6	CCR5	CCR4	CCR3	CCR2	CCR1	CCR0
Reset:	0	0	0	0	0	0	0	0

Figure 14-4 BDM CCR Holding Register (BDMCCR)

Read: All modes
Write: All modes

NOTE: *When BDM is made active, the CPU stores the value of the CCR register in the BDMCCR register. However, out of special single-chip reset, the BDMCCR is set to $D8 and not $D0 which is the reset value of the CCR register.*

When entering background debug mode, the BDM CCR holding register is used to save the contents of the condition code register of the user's program. It is also used for temporary storage in the standard BDM firmware mode. The BDM CCR holding register can be written to modify the CCR value.

14.3.3 BDM Internal Register Position Register

Address: $FF07

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	REG15	REG14	REG13	REG12	REG11	0	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0

= Unimplemented

Figure 14-5 BDM Internal Register Position (BDMINR)

Read: All modes
Write: Never

REG15–REG11 - Internal register map position

These five bits show the state of the upper five bits of the base address for the system's relocatable register block. BDMINR is a shadow of the INITRG register which maps the register block to any 2K byte space within the first 32K bytes of the 64K byte address space.

14.4 Operation

The BDM receives and executes commands from a host via a single wire serial interface. There are two types of BDM commands, namely, hardware commands and firmware commands.

Hardware commands are used to read and write target system memory locations and to enter active background debug mode (see **14.4.3**). Target system memory includes all memory that is accessible by the CPU.

Firmware commands are used to read and write CPU resources and to exit from active background debug mode (see **14.4.4**). The CPU resources referred to are the accumulator (D), X index register (X), Y index register (Y), stack pointer (SP), and program counter (PC).

Hardware commands can be executed at any time and in any mode excluding a few exceptions as highlighted in **14.5** below. Firmware commands can only be executed when the system is in active background debug mode (BDM).

14.4.1 Security

If the user resets into special single chip mode with the system secured, a secured mode BDM firmware lookup table is brought into the map overlapping a portion of the standard BDM firmware lookup table. The secure BDM firmware verifies that the on-chip EEPROM and Flash EEPROM are erased. This being the case, the UNSEC bit will get set. The BDM program jumps to the start of the standard BDM firmware and the secured mode BDM firmware is turned off. If the EEPROM and FLASH do not verify as erased, the BDM firmware sets the ENBDM bit, without asserting UNSEC, and the firmware enters a loop. This

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

causes the BDM hardware commands to become enabled, but does not enable the software commands. This allows the BDM hardware to be used to erase the EEPROM and FLASH.

14.4.2 Enabling and Activating BDM

The system must be in active BDM to execute standard BDM firmware commands. BDM can be activated only after being enabled. BDM is enabled by setting the ENBDM bit in the BDM status (BDMSTS) register. The ENBDM bit is set by writing to the BDM status (BDMSTS) register, via the single-wire interface, using a hardware command such as WRITE_BD_BYTE.

After being enabled, BDM is activated by one of the following[1]:

- Hardware BACKGROUND command
- BDM external instruction tagging mechanism
- CPU BGND instruction
- Breakpoint sub-block's force or tag mechanism[2]

When BDM is activated, the CPU finishes executing the current instruction and then begins executing the firmware in the standard BDM firmware lookup table. When BDM is activated by the breakpoint sub-block, the type of breakpoint used determines if BDM becomes active before or after execution of the next instruction.

NOTE: *If an attempt is made to activate BDM before being enabled, the CPU resumes normal instruction execution after a brief delay. If BDM is not enabled, any hardware BACKGROUND commands issued are ignored by the BDM and the CPU is not delayed.*

In active BDM, the BDM registers and standard BDM firmware lookup table are mapped to addresses $FF00 to $FFFF. BDM registers are mapped to addresses $FF00 to $FF07. The BDM uses these registers which are readable anytime by the BDM. These registers are not, however, readable by user programs.

14.4.3 BDM Hardware Commands

Hardware commands are used to read and write target system memory locations and to enter active background debug mode. Target system memory includes all memory that is accessible by the CPU such as on-chip RAM, EEPROM, Flash EEPROM, I/O and control registers, and all external memory.

Hardware commands are executed with minimal or no CPU intervention and do not require the system to be in active BDM for execution, although, they can still be executed in this mode. When executing a hardware command, the BDM sub-block waits for a free CPU bus cycle so that the background access does not disturb the running application program. If a free cycle is not found within 128 clock cycles, the CPU is momentarily frozen so that the BDM can steal a cycle. When the BDM finds a free cycle, the operation does not intrude on normal CPU operation provided that it can be completed in a single cycle. However,

NOTES:
1. BDM is enabled and active immediately out of special single-chip reset (see **14.5.2**).
2. This method is only available on systems that have a a Breakpoint sub-block.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

if an operation requires multiple cycles, the CPU is frozen until the operation is complete, even though the BDM found a free cycle.

The BDM hardware commands are listed in **Table 14-1**.

Table 14-1 Hardware Commands

Command	Opcode (hex)	Data	Description
BACKGROUN	90	None	Enter background mode if firmware is enabled.
READ_BD_BYTE	E4	16-bit address 16-bit data out	Read from memory with standard BDM firmware lookup table in map. Odd address data on low byte; even address data on high byte
READ_BD_WORD	EC	16-bit address 16-bit data out	Read from memory with standard BDM firmware lookup table in map. Must be aligned access.
READ_BYTE	E0	16-bit address 16-bit data out	Read from memory with standard BDM firmware lookup table out of map. Odd address data on low byte; even address data on high byte
READ_WORD	E8	16-bit address 16-bit data out	Read from memory with standard BDM firmware lookup table out of map. Must be aligned access.
WRITE_BD_BYTE	C4	16-bit address 16-bit data in	Write to memory with standard BDM firmware lookup table in map. Odd address data on low byte; even address data on high byte
WRITE_BD_WORD	CC	16-bit address 16-bit data in	Write to memory with standard BDM firmware lookup table in map. Must be aligned access
WRITE_BYTE	C0	16-bit address 16-bit data in	Write to memory with standard BDM firmware lookup table out of map. Odd address data on low byte; even address data on high byte
WRITE_WORD	C8	16-bit address 16-bit data in	Write to memory with standard BDM firmware lookup table out of map. Must be aligned access.

The READ_BD and WRITE_BD commands allow access to the BDM register locations. These locations are not normally in the system memory map but share addresses with the application in memory. To distinguish between physical memory locations that share the same address, BDM memory resources are enabled just for the READ_BD and WRITE_BD access cycle. This allows the BDM to access BDM locations unobtrusively, even if the addresses conflict with the application memory map.

14.4.4 Standard BDM Firmware Commands

Firmware commands are used to access and manipulate CPU resources. The system must be in active BDM to execute standard BDM firmware commands (see **14.4.2**). Normal instruction execution is suspended while the CPU executes the firmware located in the standard BDM firmware lookup table. The hardware command BACKGROUND is the usual way to activate BDM.

As the system enters active BDM, the standard BDM firmware lookup table and BDM registers become visible in the on-chip memory map at $FF00-$FFFF, and the CPU begins executing the standard BDM

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

firmware. The standard BDM firmware watches for serial commands and executes them as they are received. The firmware commands are shown in **Table 14-2**.

Table 14-2 Firmware Commands

Command	Opcode (hex)	Data	Description
READ_NEXT	62	16-bit data out	Increment X by 2 (X = X + 2), then read word X points to.
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.
WRITE_NEXT	42	16-bit data in	Increment X by 2 (X=X+2), then write word to location pointed to by 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.
GO	08	none	Go to user program.
TRACE1	10	none	Execute one user instruction then return to active BDM.
TAGGO	18	none	Enable tagging and go to user program.

14.4.5 BDM Command Structure

Hardware and firmware BDM commands start with an 8-bit opcode followed by a 16-bit address and/or a 16-bit data word depending on the command. All the read commands return 16 bits of data despite the byte or word implication in the command name.

NOTE: *8-bit reads return 16-bits of data, of which, only one byte will contain valid data. If reading an even address, the valid data will appear in the MSB. If reading an odd address, the valid data will appear in the LSB.*

NOTE: *16-bit misaligned reads and writes are not allowed. If attempted, the BDM will ignore the least significant bit of the address and will assume an even address from the remaining bits.*

For hardware data read commands, the external host must wait 150 target clock cycles[1] after sending the address before attempting to obtain the read data. This is to be certain that valid data is available in the BDM shift register, ready to be shifted out. For hardware write commands, the external host must wait 150 target clock cycles after sending the data to be written before attempting to send a new command. This is to avoid disturbing the BDM shift register before the write has been completed. The 150 target clock cycle delay in both cases includes the maximum 128 cycle delay that can be incurred as the BDM waits for a free cycle before stealing a cycle.

For firmware read commands, the external host must wait 32 target clock cycles after sending the command opcode before attempting to obtain the read data. This allows enough time for the requested data to be made available in the BDM shift register, ready to be shifted out. For firmware write commands, the external host must wait 32 target clock cycles after sending the data to be written before attempting to send a new command. This is to avoid disturbing the BDM shift register before the write has been completed.

The external host should wait 64 target clock cycles after a TRACE1 or GO command before starting any new serial command. This is to allow the CPU to exit gracefully from the standard BDM firmware lookup table and resume execution of the user code. Disturbing the BDM shift register prematurely may adversely affect the exit from the standard BDM firmware lookup table.

Figure 14-6 represents the BDM command structure. The command blocks illustrate a series of eight bit times starting with a falling edge. The bar across the top of the blocks indicates that the BKGD line idles in the high state. The time for an 8-bit command is 8×16 target clock cycles.

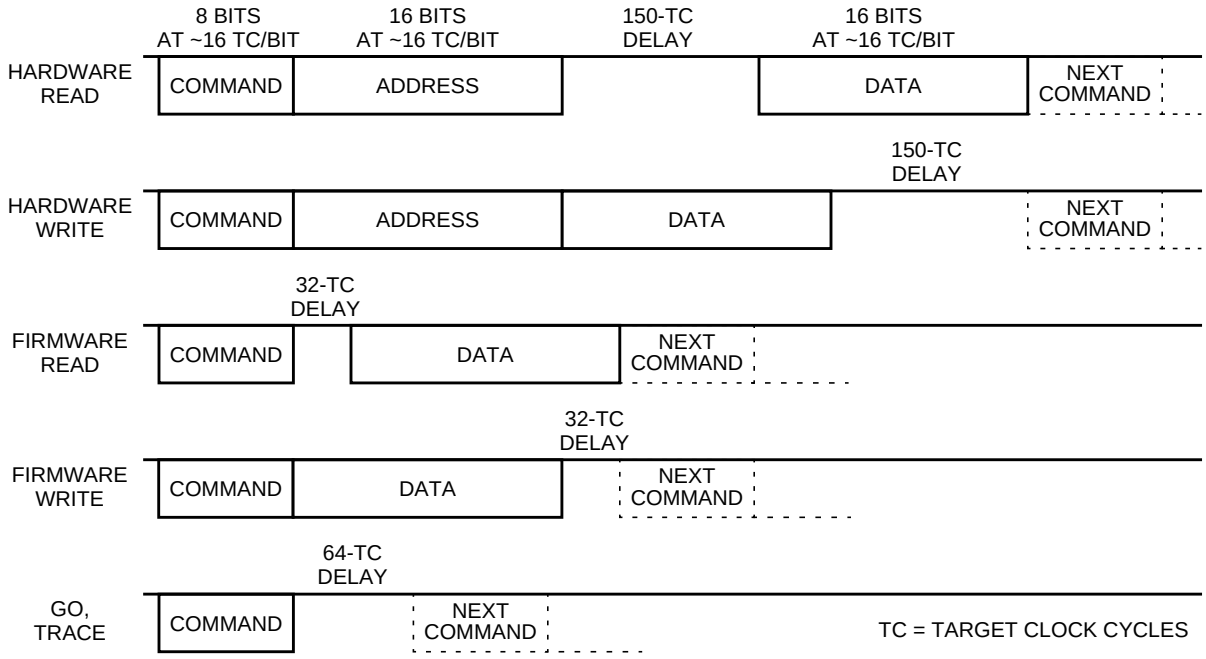


Figure 14-6 BDM Command Structure

NOTES:

1. Target clock cycles are cycles measured using the target system's serial clock rate. See **14.4.6** and **14.3.1** for information on how serial clock rate is selected.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

14.4.6 BDM Serial Interface

The BDM communicates with external devices serially via the BKGD pin. During reset, this pin is a mode select input which selects between normal and special modes of operation. After reset, this pin becomes the dedicated serial interface pin for the BDM.

The BDM serial interface is timed using the clock selected by the CLKSW bit in the status register (see **14.3.1**). This clock will be referred to as the target clock in the following explanation.

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 is sent for every bit whether data is transmitted or received. Data is transferred most significant bit (MSB) first at 16 target clock cycles per bit. The interface times out if 512 clock cycles occur between falling edges from the host.

The BKGD pin is a pseudo open-drain pin and has an weak on-chip active pull-up that is enabled at all times. It is assumed that there is an external pullup and that drivers connected to BKGD do not typically drive the high level. Since R-C rise time could be unacceptably long, the target system and host provide brief driven-high (speedup) pulses to drive BKGD to a logic 1. The source of this speedup pulse is the host for transmit cases and the target for receive cases.

The timing for host-to-target is shown in **Figure 14-7** and that of target-to-host in **Figure 14-8** and **Figure 14-9** below. All four cases begin when the host drives the BKGD pin low to generate a falling edge. Since the host and target are operating from separate clocks, it can take the target system up to one full clock cycle to recognize this edge. The target measures delays from this perceived start of the bit time while the host measures delays from the point it actually drove BKGD low to start the bit up to one target clock cycle earlier. Synchronization between the host and target is established in this manner at the start of every bit time.

Figure 14-7 shows an external host transmitting a logic 1 and transmitting a logic 0 to the BKGD pin of a target system. The host is asynchronous to the target, so there is up to a one clock-cycle delay from the host-generated falling edge to where the target recognizes this edge as the beginning of the bit time. Ten target clock cycles later, the target senses the bit level on the BKGD pin. Internal glitch detect logic requires the pin be driven high no later that eight target clock cycles after the falling edge for a logic 1 transmission.

Since the host drives the high speedup pulses in these two cases, the rising edges look like digitally driven signals.

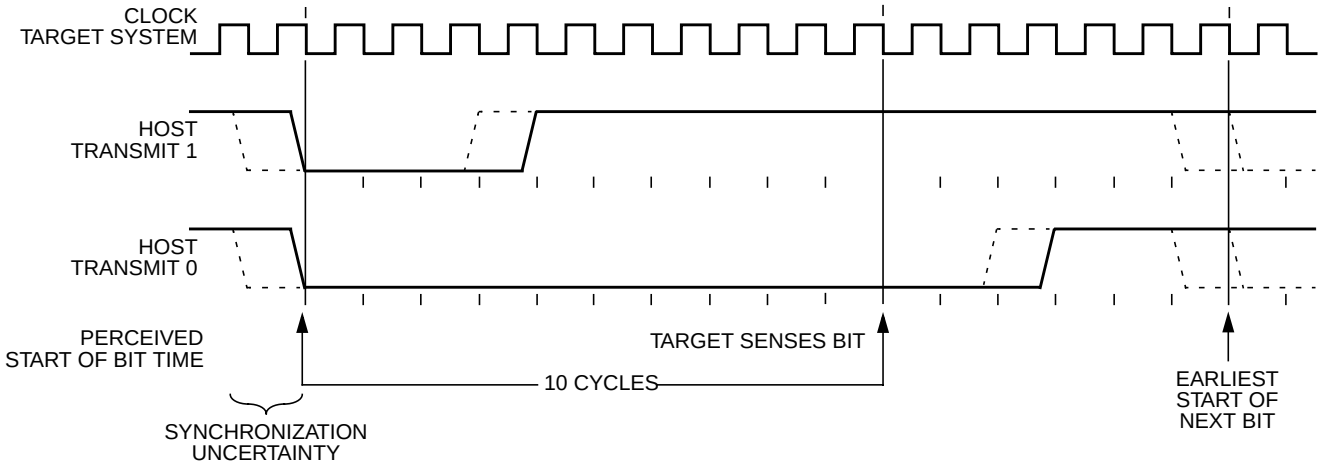


Figure 14-7 BDM Host-to-Target Serial Bit Timing

The receive cases are more complicated. **Figure 14-8** shows the host receiving a logic 1 from the target system. Since the host is asynchronous to the target, there is up to one clock-cycle delay from the host-generated falling edge on BKGD to the perceived start of the bit time in the target. The host holds the BKGD pin low long enough for the target to recognize it (at least two target clock cycles). The host must release the low drive before the target drives a brief high speedup pulse seven target clock cycles after the perceived start of the bit time. The host should sample the bit level about 10 target clock cycles after it started the bit time.

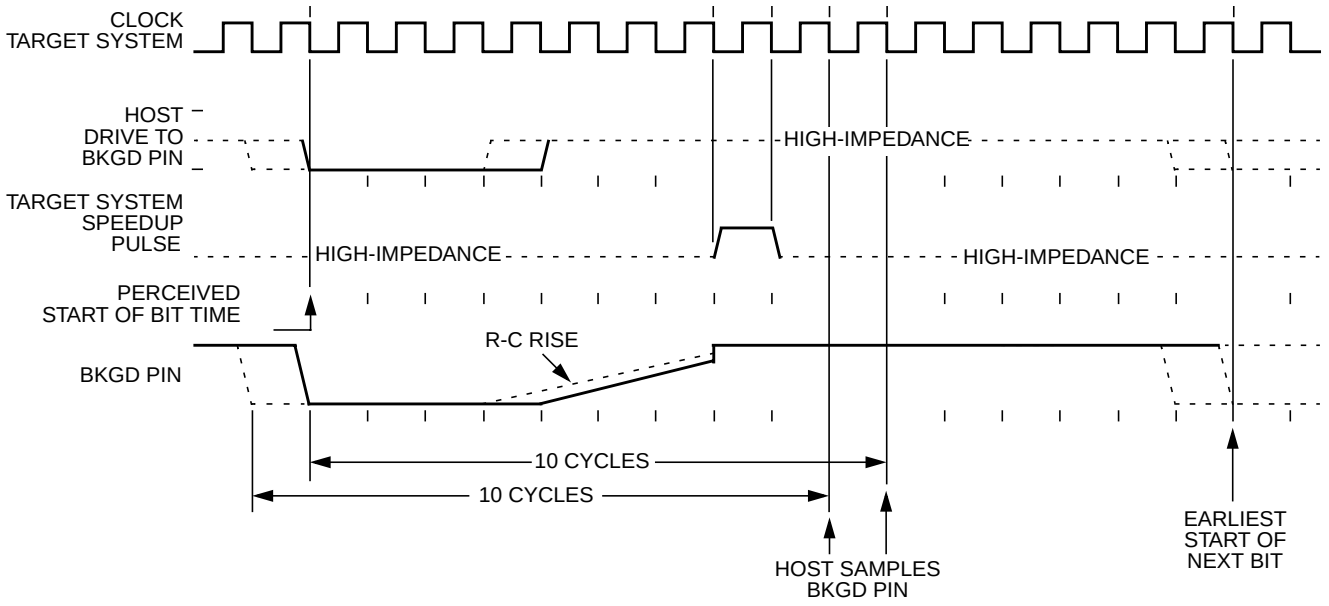


Figure 14-8 BDM Target-to-Host Serial Bit Timing (Logic 1)

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Figure 14-9 shows the host receiving a logic 0 from the target. Since the host is asynchronous to the target, there is up to a one clock-cycle delay from the host-generated falling edge on BKGD to the start of the bit time as perceived by the target. The host initiates the bit time but the target finishes it. Since the target wants the host to receive a logic 0, it drives the BKGD pin low for 13 target clock cycles then briefly drives it high to speed up the rising edge. The host samples the bit level about 10 target clock cycles after starting the bit time.

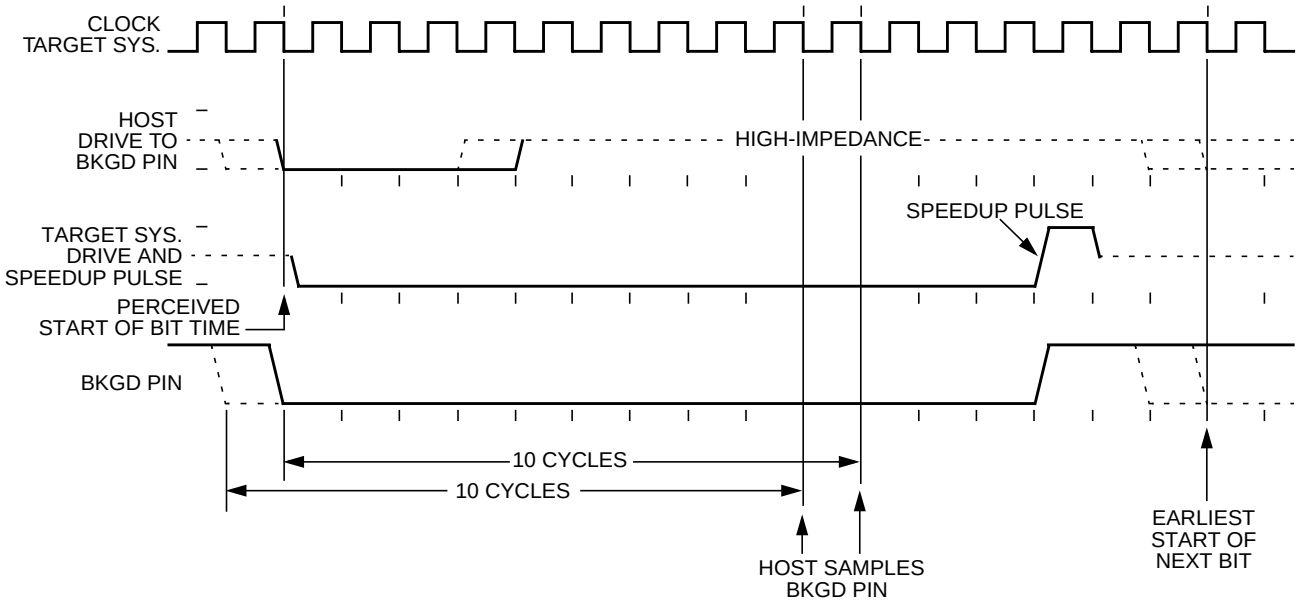


Figure 14-9 BDM Target-to-Host Serial Bit Timing (Logic 0)

14.4.7 Instruction Tracing

When a TRACE1 command is issued to the BDM in active BDM, the CPU exits the standard BDM firmware and executes a single instruction in the user code. Once this has occurred, the CPU is forced to return to the standard BDM firmware and the BDM is active and ready to receive a new command. If the TRACE1 command is issued again, the next user instruction will be executed. This facilitates stepping or tracing through the user code one instruction at a time.

If an interrupt is pending when a TRACE1 command is issued, the interrupt stacking operation occurs but no user instruction is executed. Once back in standard BDM firmware execution, the program counter points to the first instruction in the interrupt service routine.

14.4.8 Instruction Tagging

The instruction queue and cycle-by-cycle CPU activity are reconstructible in real time or from trace history that is captured by a logic analyzer. However, the reconstructed queue cannot be used to stop the CPU at a specific instruction, because execution already has begun by the time an operation is visible outside the system. A separate instruction tagging mechanism is provided for this purpose.

The tag follows program information as it advances through the instruction queue. When a tagged instruction reaches the head of the queue, the CPU enters active BDM rather than executing the instruction.

NOTE: *Tagging is disabled when BDM becomes active and BDM serial commands are not processed while tagging is active.*

Executing the BDM TAGGO command configures two system pins for tagging. The $\overline{\text{TAGLO}}$ signal shares a pin with the $\overline{\text{LSTRB}}$ signal, and the $\overline{\text{TAGHI}}$ signal shares a pin with the BKGD signal.

Table 14-3 shows the functions of the two tagging pins. The pins operate independently, that is, 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 the external clock (ECLK) performs the indicated function. High tagging is allowed in all modes. Low tagging is allowed only when low strobe is enabled (LSTRB is allowed only in wide expanded modes and emulation expanded narrow mode).

Table 14-3 Tag Pin Function

TAGHI	TAGLO	Tag
1	1	No tag
1	0	Low byte
0	1	High byte
0	0	Both bytes

14.5 Modes of Operation

BDM is available in all operating modes but must be enabled before firmware commands are executed.

Some system peripherals may have a control bit which allows suspending the peripheral function during background debug mode.

In special single-chip mode, background operation is enabled and active out of reset. This allows programming a system with blank memory.

BDM is also active out of special peripheral mode reset and can be turned off by clearing the BDMACT bit in the BDM status (BDMSTS) register. This allows testing of the BDM memory space as well as the user's memory space.

NOTE: *The BDM serial system should not be used in special peripheral mode since the CPU, which in other modes interfaces with the BDM to relinquish control of the bus during a free cycle or a steal operation, is not operating in this mode.*

14.5.1 Normal Operation

BDM operates the same in all normal modes.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

14.5.2 Special Operation

14.5.2.1 Special single-chip mode

BDM is enabled and active immediately out of reset. This allows programming a system with blank memory.

14.5.2.2 Special peripheral mode

BDM is enabled and active immediately out of reset. BDM can be disabled by clearing the BDMACT bit in the BDM status (BDMSTS) register. The BDM serial system should not be used in special peripheral mode.

14.5.3 Emulation Modes

In emulation modes, the BDM operates as in all normal modes.

14.6 Low-Power Options

14.6.1 Run Mode

The BDM does not include disable controls that would conserve power during run mode.

14.6.2 Wait Mode

The BDM cannot be used in wait mode if the system disables the clocks to the BDM.

14.6.3 Stop Mode

The BDM is completely shutdown in stop mode.

14.7 Interrupt Operation

The BDM does not generate interrupt requests.

14.8 Motorola Internal Information

This subsection details information about the BDM sub-block that is for Motorola use only and should not be published in any form outside of Motorola.

14.8.1 Registers

This section gives detailed descriptions of all internally accessible registers and bits that are either not available or not disclosed to users external to Motorola. These registers were highlighted as being reserved BDM registers previously in this section of the guide.

The BDM instruction (BDMIST) register is written by the BDM hardware as a result of a BDM command sent to the system via the BKGD pin. The individual bits decode into categories of BDM instruction. The two descriptions of the BDMIST below show the instruction decode when categorized as hardware or firmware instructions.

All of the BDM registers are readable and writable in special peripheral mode on the parallel bus until the BDMACT bit in the BDMSTS register is cleared at which time the BDM resources are no longer accessible via the peripheral bus and require a reset to be restored.

A full summary of the registers associated with the BDM is shown in **Figure 14-10** below.

Address	Name		Bit 7	6	5	4	3	2	1	Bit 0
$FF00	BDMIST	read	H/F	DATA	R/W	BKGND	W/B	BD/U	0	0
		write								
$FF01	BDMSTS	read	ENBDM	BDMACT	ENTAG	SDV	TRACE	CLKSW	UNSEC	0
		write								
$FF02	BDMSHTH	read	S15	S14	S13	S12	S11	S10	S9	S8
		write								
$FF03	BDMSHTL	read	S7	S6	S5	S4	S3	S2	S1	S0
		write								
$FF04	BDMADDH	read	A15	A14	A13	A12	A11	A10	A9	A8
		write								
$FF05	BDMADDL	read	A7	A6	A5	A4	A3	A2	A1	A0
		write								
$FF06	BDMCCR	read	CCR7	CCR6	CCR5	CCR4	CCR3	CCR2	CCR1	CCR0
		write								
$FF07	BDMINR	read	REG15	REG14	REG13	REG12	REG11	0	0	0
		write								

= Unimplemented X = Indeterminate

Figure 14-10 BDM Register Map

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

14.8.2 BDM Instruction Register (Hardware)

Address: $FF00

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	H/F	DATA	R/W	BKGND	W/B	BD/U	0	0
Write:								
Reset:	0	0	0	0	0	0	0	0

Figure 14-11 BDM Instruction Register (BDMIST)

Read: All modes

Write: All modes; BDM hardware writes this register when a BDM command is received. Hardware clears the register if 512 BDM clock cycles occur between falling edges from the host. Firmware clears this register when exiting from BDM active mode.

H/F - Hardware/firmware flag

When the BDM is active, standard BDM firmware checks for this bit to be set by the BDM hardware as part of a BDM instruction load.

1 = Hardware command
0 = Firmware command

DATA - Data flag

Shows that data accompanies the command.

1 = Data follows the command
0 = No data

R/W - Read/write flag

1 = Read
0 = Write

BKGND - Enter active background mode

1 = Hardware background command
0 = Not a hardware background command

W/B - Word/byte transfer flag

1 = Word transfer
0 = Byte transfer

BD/U - BDM map/user map flag

Indicates whether BDM access is to BDM registers and standard BDM firmware lookup table mapped to addresses $FF00 to $FFFF or the user resources in this range. Used only by hardware read/write commands.

1 = standard BDM firmware lookup table and registers in map
0 = User resources in map.

14.8.3 BDM Instruction Register (Firmware)

Address: $FF00

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	H/F	DATA	R/W	TTAGO		RNEXT		
Reset:	0	0	0	0	0	0	0	0

Figure 14-12 BDM Instruction Register (BDMIST)

Read: All modes

Write: All modes; BDM hardware writes this register when a BDM command is received. Hardware clears the register if 512 BDM clock cycles occur between falling edges from the host. Firmware clears this register when exiting from BDM active mode.

H/F - Hardware/firmware flag

When the BDM is active, standard BDM firmware checks for this bit to be set by the BDM hardware as part of a BDM instruction load.

1 = Hardware command
0 = Firmware command

DATA - Data flag

This bit indicates that data accompanies the command.

1 = Data follows the command
0 = No data

R/W - Read/write flag

1 = Read
0 = Write

TTAGO - Trace, tag, go bits.

The decoding of TTAGO is shown in **Table 14-4** below.

Table 14-4 TTAGO Decoding

TTAGO value	Instruction
00	—
01	GO
10	TRACE1
11	TAGGO

RNEXT - Register/next bits

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Indicates which register is being affected by a command. In the case of a READ_NEXT or WRITE_NEXT command, index register X is pre-incremented by 2 and the word pointed to by X is then read or written. The decoding of RNEXT is shown in **Table 14-5** below.

Table 14-5 RNEXT Decoding

RNEXT value	Instruction
000	—
001	—
010	READ/WRITE NEXT
011	PC
100	D
101	X
110	Y
111	SP

14.8.4 BDM Status Register

The BDM status (BDMSTS) register is described in **14.3.1**. In addition, it is readable and writable in special peripheral mode on the parallel bus.

BDMACT - BDM active status

BDMACT is set by the BDM and is cleared in the exit sequence of the standard BDM firmware. BDMACT can be written to in special peripheral mode via the peripheral bus. It cannot be written to via BDM hardware commands in any mode, that is, it cannot be written to if the H/F bit in the BDMIST register is set.

Clearing BDMACT causes the standard BDM firmware lookup table and registers to be removed from the memory map and BDM to become inactive.

Setting BDMACT in special peripheral mode via the peripheral bus causes BDM to become active but does not put the standard BDM firmware lookup table and registers into the memory map; therefore, BDMACT should not be set in this manner but should instead be set by resetting the system.

14.8.5 BDM Shift Register

Address: $FF02

	Bit 15	14	13	12	11	10	9	Bit 8
Read:	S15	S14	S13	S12	S11	S10	S9	S8
Write:								
Reset:								

Figure 14-13 BDM Shift Register (BDMSHTH)

Address: $FF03

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	S7	S6	S5	S4	S3	S2	S1	S0
Write:								
Reset:								

Figure 14-14 BDM Shift Register (BDMSHTL)

Read: All modes
Write: All modes

The 16-bit BDM shift register contains data being received or transmitted via the serial interface. It is also used by the standard BDM firmware for temporary storage.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

14.8.6 BDM Address Register

Address: $FF04

	Bit 15	14	13	12	11	10	9	Bit 8
Read: Write:	A15	A14	A13	A12	A11	A10	A9	A8
Reset:								

Figure 14-15 BDM Address Register (BDMADDH)

Address: $FF05

	Bit 7	6	5	4	3	2	1	Bit 0
Read: Write:	A7	A6	A5	A4	A3	A2	A1	A0
Reset:								

Figure 14-16 BDM Address Register (BDMADDL)

Read: All modes

Write: Can only be written by BDM hardware

In secure mode, if the BDM hardware commands have been enabled by the secure firmware, the upper 5 bits of the address register will always be forced to the value from the BDMINR register. This restricts access of the hardware commands to the register space only.

The 16-bit address register is loaded with the address to be accessed by BDM hardware commands.

14.8.7 Special Peripheral Mode

In Special Peripheral Mode the BDM is enabled and active immediately out of reset. BDM can be disabled by clearing the BDMACT bit in the BDM status (BDMSTS) register (see **14.8.4**). This allows testing the BDM memory space as well as the user's program memory space. The BDM serial system should not be used in special peripheral mode since the CPU, which in other modes relinquishes control of the bus during a free cycle or a steal operation, is not operating in this mode.

14.8.8 Standard BDM Firmware Listing

```
;*******************************************************************************
;
;   Copyright (C) 1997 by Motorola Inc.
;                         6501 William Cannon Drive West
;                         Advanced MCU HC11 Group
;                         Austin, TX 78735-8598
;
;   All rights reserved. No part of this software may be sold or distributed
```

```
;   in any form or by any means without the prior written permission of
;   Motorola, Inc.
;
;         MOTOROLA CONFIDENTIAL PROPRIETARY INFORMATION
;
;*******************************************************************************
;===============================================================================
;                                       VERSION HISTORY
;
; Started from UDR HC12 BDM ROM code
;
;===============================================================================
;
; Design Strategy:
;         -standard BDM firmware for M68HC12
;         -There are MANY traps that someone modifying this code MUST be aware of.
;         Those areas that have traps that we have fallen into and requiring
;         special care have been marked with CAUTION. Here is a list of
;         items to BEWARE of. Review this list after ANY ROM code changes.
;         CAUTION 1. There is an inherent cpudead cycle that we rely on in the
;                    INST_LOOP loop when that ldaa instruction falls on an even
;                    address. For this reason, an ALIGN directive MUST be used
;                    at that location. See AR#156.
;         CAUTION 2. The first event that occurs in code that may interfere
;                    with user code is the saving of all internal registers. When
;                    this BDM code is entered, all the internal registers such as
;                    CCR, PC, X, etc. MUST be saved so that they may be restored to
;                    the user's value upon an exit from this code.
;         CAUTION 3. DO NOT insert code that affects the user CCR value before
;                    it gets saved. The code that saves the user CCR should be one
;                    of the very first items that occur at the beginning of this
;                    code. See AR#166.
;         CAUTION 4. The PC value MUST be checked to see if it was a BDM (op=00)
;                    instruction that got us into BDM. If so, PC gets adjusted by 1.
;                    This works only if the user enters BDM from locations $0000
;                    thru $FEFF because locations $FF00-$FFFF are blocked out for the
;                    BDM. So, the BDM ROM is in the map and not the user's code.
;         CAUTION 5. Any unused space should be set to $00 to ensure ROM
;                    is plugged and verified properly. Be careful to NOT OVERLAP
;                    vector space when filling unused space!!! Using the ZMB
;                    directive helps because the assembler version we used just hangs
;                    up when code OVERLAPs BUT some other assembler version may
;                    not catch this.
;         CAUTION 6. The ROM code size is limited in available space. Make
;                    sure that when instructions are added, the vector space is not
;                    overwritten.
;         CAUTION 7. The reset vector was INST_DONE. Added code
;                    so that after a reset, the ccr value at reset is saved because
;                    the exit sequence was changing the CCR to the value that was
;                    saved before the reset occurred. The user should really
;                    initialize the CCR, but we do it here to avoid confusion.
;         CAUTION 8. The ENBDM bit MUST be set out of reset, otherwise it won't
;                    pass the "brset STATUS $80 INST_LOOP" test and the user gets
;                    kicked out of background unintentionally.
;         CAUTION 9. The Dev. Tools PRU relies on the BDM entry point "START"
;                    being at location $FF24. They also rely on the exit point
;                    being at location $FF77 (the exit jump). Any changes to the
;                    start and exit points MUST be reviewed with them.
;         CAUTION 10. Be careful that the BDMACT bit in the STATUS register is
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

```
;                   not unintentionally changed from a 1 to a 0 during 16-bit
;                   manipulation of the INSTRUCTION register. This will cause a
;                   race condition because BDMACT=0 will disable the standard BDM firmware
;                   ROM while the CPU is executing this firmware.
;         -This is a list of instructions which use the temp2 (t2) and temp3 (t3)
;         instructions. List as of 7-27-94. Gotten from Tom Poterek's BDMcode.
;                   temp2
;                   =====
;                   bgnd
;                   emacs
;                   etbl
;                   mem
;                   revw
;                   stop
;                   tbl
;                   wai
;                   execution of BDM ROM
;
;                   temp3
;                   =====
;                   emacs
;                   etbl
;                   mem
;                   puld
;                   pulx
;                   puly
;                   rtc
;                   rti
;                   rts
;                   tbl
;                   wav
;                   execution of BDM ROM
;
********************************************************************************

********************************************************************************
                    * EQUATES

fff6                BDMVEC              equ$fff6  ;First BDM ROM vector.

ff00                                    org$ff00  ;Start of BDM map (registers)
ff00                INSTR               rmb1      ;Instruction (command) register
                    * s/w  !  H/S ! DATA !  R/W ! TTAG :  GO  !  R2  !  R1  !  R0  !
                    * hdw  !  H/S ! DATA !  R/W !BKGND :  W/B !BD/USR! NEXT !   -  !
                    * Reg codes: R2:R1:R0
                    * 0:0:0 - Illegal, command $00 is null command
                    * 0:0:1 - not used
                    * 0:1:0 - Next Word 2,+X pre inc X by 2 and r/w next word (,X)
                    *          later r/w next will work from ADDRESS reg value not X
                    * 0:1:1 - PC
                    * 1:0:0 - D
                    * 1:0:1 - X
                    * 1:1:0 - Y
                    * 1:1:1 - SP
                    * TTAG:GO coding:
                    * 0:0 - No execution command
                    * 0:1 - Go to user program
                    * 1:0 - Trace one user instruction and return to BDM
```

```
                    * 1:1 - Tag Go command (reconfigure BKGD pin for tagging in)

ff01                STATUS              rmb1      ;Status/Control register
                    *     ! enBDM!BDACTV!  TAG ! VALID: TRACE!   -  !   -  !   -  !
                    * Exit conditions vs value written to STATUS on exit
                    *  BDM not allowed - $00
                    *          Trace 1 - $88
                    *               Go - $80
                    *           Tag Go - $A0

ff02                SHIFTER             rmb2      ;For serial data in/out

ff04                ADDRESS             rmb2      ;Address for some commands
                    * ADDRESS will be read-only on first parts but later it will
                    * be r/w so r/w next word doesn't need to use X

ff06                CCRSAVE             rmb1      ;Save user CCR value while in BDM
                    * CCRSAVE also used briefly to hold exit value for status
                    * during exit sequence to return to user code


ff20                                    orgff20   ;BDM ROM start
                    AFTER_RST             ;*****CAUTION 7.  *****CAUTION 8.
ff20 1c ff 01 80                        bset STATUS $80;Set the ENBDM bit to pass the brset
                                                  ;test below.
                                                  ;CCR immediately after rst is
                                                  ;SXHINZVC=11x1xxxx.
                                                  ;CCR after this bset is
                                                  ;SXHINZVC=11x1100x.  This is o.k.
                                                  ;because the SXI bits are not
                                                  ;affected.
                    START
                                      ;*****CAUTION 2.  *****CAUTION 9.
ff24 b7 b4                              exgt3 d   ;Save D without affecting CCR.
                                                  ;This "exg t3 d" instruction MUST
                                                  ;occur before the following
                                                  ;"tfr ccr a" instruction.
                                      ;*****CAUTION 3.
ff26 b7 20                              tfrccr a
ff28 7a ff 06                           staaCCRSAVE;Save user CCR value
ff2b b7 d3                              exgx t2   ;pc into x.  *****CAUTION 4.
ff2d 8e ff 00                           cpx#$FF00 ;Check to see if user PC overlaps BDM
                                                  ;ROM.
ff30 24 04                              bhsROM_INC;If so, increment regardless.
ff32 e7 00                              tst0,x    ;Test next opcode.  This instruction
                                                  ;affects CCR so it MUST occur AFTER
                                                  ;saving the user's CCR.
ff34 26 01                              bneRES_X_T2;if not $00, restore
ff36 08             ROM_INC             inx       ;else inc, then restore. This
                                                  ;instruction affects CCR so it MUST
                                                  ;occur AFTER saving the user's CCR.
                    RES_X_T2
ff37 b7 d3                              exgx t2   ;restore pc to temp 2
ff39 1e ff 01 80 06                     brsetSTATUS $80 INST_LOOP  ;Check if BDM allowed
ff3e 87                                 clra      ;Exit if BDM not allowed
ff3f 20 1c                              braEXIT_SEQ
                    * Above is 1 of 4 ways to exit BDM to user code.
                    INST_DONE
ff41 79 ff 00                           clrINSTR  ;clear INSTR then wait for new inst
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

```
                                              ;CAUTION 10.

                    * Top of main loop to wait for a software instruction
                                              ;*****CAUTION 1.
ff44                                  ALIGN 1   ;Make sure the following loop
                                              ;starting with ldaa is ALWAYS on an
                                              ;even boundary.
                                              ;See AR# 156 for more details.
                    INST_LOOP
ff44 b6 ff 00                         ldaaINSTR ;Wait for non-zero non-hdw command
ff47 2f fb                            bleINST_LOOP;$00 is null command
                                              ;MSB of A set (neg) is hdw command
ff49 85 18                            bita#$18  ;TAGGO,TRACE, or GO commands?
ff4b 27 2e                            beqNOT_EXE;Branch if not execution command
ff4d 81 10                            cmpa#$10  ;TRACE ---1:0--- ?  tp 4/7/95
ff4f 27 06                            beqTRACE
ff51 2b 08                            bmiGO     ;If not GO it's TAG GO
                    * Fall through from TAG_GO is 4th of 4 ways to exit to user code.
ff53 86 a0                            ldaa#$A0  ;enBDM + TAG bits in STATUS
ff55 20 06                            braEXIT_SEQ;Controlled exit (3 of 4)
                    TRACE
ff57 86 88                            ldaa#$88  ;enBDM + TRACE bits in STATUS
ff59 20 02                            braEXIT_SEQ;Controlled exit (2 of 4)
                    GO
ff5b 86 80                            ldaa#$80  ;enBDM bit only in STATUS

                    * Upon entry to EXIT_SEQ, A contains a value to be written
                    * to the STATUS register. Seq restores user info and
                    * resumes user program where it left to enter active BD mode

                    EXIT_SEQ                  ;CAUTION 10.
ff5d 79 ff 00                         clrINSTR  ;clear instruction    tp 4/6/95
ff60 f6 ff 06                         ldabCCRSAVE;re-entry value for CCR
ff63 7a ff 06                         staaCCRSAVE;will use movb to store to STATUS
ff66 b7 d3                            exgx t2   ;Swap X to Temp2 and User PC to X
ff68 7e ff 02                         stxSHIFTER;For later indirect jump
ff6b b7 d3                            exgx t2   ;Restore user X
ff6d b7 12                            tfrb ccr  ;Restore user CCR
ff6f b7 b4                            exgt3 d   ;Restore user D reg
ff71 18 0c ff 06 ff 01                movbCCRSAVE STATUS;[OrPwPO] write w/o chg to ccr

                    * Critical timing: cycle signature of above move is OrPwPO
                    * Exit timing referenced to the byte-write in cycle 4
                    * Cycle signatures of remaining instructions in exit seq
                    * are shown in the comments. ROM switch from BD ROM to
                    * user map should occur at f cycle before PPP in exit jump
                    * If TRACE, issue liufbdm at T4 of the second last P cycle
                    * of the exit jump
                    *                  O r P w P O f I f P P P
                    *                        !       !     !
ff77 05 fb ff 87                      jmp(SHIFTER-(*+4)),pc]  ;[fIfPPP] Exit to user PC

                    * In this exit jump, the I cycle is a word read of the user PC
                    * from the SHIFTER register (BD map). The PPP cycles are word
                    * fetches of user program info to fill instruction queue from
                    * user's map. The ROM switch must occur between I and PPP

                    * See also *****CAUTION 9. concerning this exit jump.
```

```
                    NOT_EXE
ff7b b7 01                          tfra b    ;Duplicate command in B
ff7d 84 07                          anda#$07  ;Strip all but 3-bit reg code
ff7f 80 02                          suba#2    ;codes 0 & 1 illegal or unused
ff81 2b be                          bmiINST_DONE;branch if A now negative
ff83 c5 20                          bitb#$20  ;Check R/W bit
ff85 26 37                          bne;COMP_GOTO;Go decode read command (was beq
                                      ;tp 3/30)

                    WAIT_DATA
ff87 f7 ff 00                       tstINSTR  ;Check for new command
ff8a 27 b8                          beqINST_LOOP;Need escape if old command aborted
ff8c 1f ff 01 10 f6                 brclrSTATUS $10 WAIT_DATA  ;Wait for data ready
ff91 c6 07                          ldab#7
ff93 12                             mul       ;B = 7*(reg_code - 1)
ff94 05 fd                          jmpb,pc   ;Calculated GOTO
                    * Each write command corresponding to reg code 2-7 takes
                    * exactly 7 bytes. For command 2 (write next word) the jump will
                    * GOTO 0,pc or the location immediately after the jump
                    * For command 7 (write SP) the jump will go to (5*7),pc
                    * Each command ends with a branch to the main command loop
                    W_NXT_WRD
ff96 fc ff 02                       lddSHIFTER;Get data to write
ff99 6c 21                          std2,+x   ;pre-inc x by 2 and store word
                    INST_DONE1
ff9b 20 a4                          braINST_DONE;Intermediate branch to loop top

                    WRITE_PC
ff9d fc ff 02                       lddSHIFTER;Get data to write
ffa0 b7 c3                          exgd t2   ;User PC in Temp2 reg
ffa2 20 9d                          braINST_DONE;Branch to loop top

                    WRITE_D
ffa4 fc ff 02                       lddSHIFTER;Get data to write
ffa7 b7 b4                          exgt3 d   ;User D in Temp3 reg (was exg d t2
                                              ;tp 3/28)
ffa9 20 96                          braINST_DONE;Branch to loop top

                    WRITE_X
ffab fe ff 02                       ldxSHIFTER;Update X register
ffae 20 91                          braINST_DONE;Branch to loop top
ffb0 a7                             nop       ;Pad to make command take 7 bytes
ffb1 a7                             nop

                    WRITE_Y
ffb2 fd ff 02                       ldySHIFTER;Update Y register
ffb5 20 8a                          braINST_DONE;Branch to loop top
ffb7 a7                             nop       ;Pad to make command take 7 bytes
ffb8 a7                             nop

                    WRITE_SP
ffb9 ff ff 02                       ldsSHIFTER;Update SP register
ffbc 20 83                          braINST_DONE;Branch to loop top
                    * No need to pad last command since we don't index past it.

                    COMP_GOTO
ffbe 48                             asla      ;x2
ffbf 48                             asla      ;A = (reg_code - 2)*4
ffc0 05 fc                          jmpa,pc   ;Calculated GOTO
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

```
                    * Each read command corresponding to reg code 2-7 takes
                    * exactly 4 bytes. For command 2 (read next word) the jump will
                    * GOTO 0,pc or the location immediately after the jump
                    * For command 7 (read SP) the jump will go to (5*4),pc
                    * Each command ends with a branch to the main command loop
                    R_NXT_WRD
ffc2 ec 21                          ldd2,+x   ;pre-inc X by 2 and read word
ffc4 20 12                          braR_COMMON;D->SHIFTER and bra loop top

                    READ_PC
ffc6 20 21                          braREAD_PC1;This command needs 4 bytes
ffc8 a7                             nop       ;Pad to make command take 4 bytes
ffc9 a7                             nop

                    READ_D
ffca b7 34                          tfrt3 d   ;User D was in Temp3
ffcc 20 0a                          braR_COMMON;D->SHIFTER and bra loop top

                    READ_X
ffce b7 54                          tfrx d    ;Requested data to D
ffd0 20 06                          braR_COMMON;D->SHIFTER and bra loop top

                    READ_Y
ffd2 b7 64                          tfry d    ;Requested data to D
ffd4 20 02                          braR_COMMON;D->SHIFTER and bra loop top

                    READ_SP
ffd6 b7 74                          tfrsp d   ;Requested data to D

                    R_COMMON
ffd8 7c ff 02                       stdSHIFTER;Requested data to SHIFTER
                    WAIT
ffdb f7 ff 00                       tst INSTR ;Check for new command    tp 3/30
ffde 18 27 ff 62                    lbeq INST_LOOP;Need escape if old command aborted
                                              ;tp 3/30
ffe2 1f ff 01 10 f4                 brclrSTATUS $10 WAIT ;Wait for data ready tp 3/30
ffe7 20 b2                          braINST_DONE1;Back to loop top

                    READ_PC1
ffe9 b7 c3                          exgd t2   ;User PC to D, junk to Temp2
ffeb 7c ff 02                       std SHIFTER;User PC to SHIFTER
ffee b7 c3                          exg d t2  ;User PC to Temp2, junk to D
fff0 20 e9                          bra WAIT  ;D->SHIFTER and bra loop top

                    FIXSP
fff2 1b 89                          leas 9,sp;Restore sp
fff4 20 a5                          braINST_DONE1;And try to resume
                                              ;*****CAUTION 5.
fff6                                zmbBDMVEC-*;All unused space must be set to
                                              ;zero.

                    *****
                    * All other normal vectors are blocked out when in BDM.  The bdmact
                    * signal goes into INT module and blocks all I and X interrupts.
                    *****CAUTION 6.

fff6                                orgBDMVEC ;BDM vectors start
fff6 ff 24          SWIV            fdbSTART  ;SWI vector (normal entry point)
fff8 ff f2          ILLOPV          fdbFIXSP  ;Illegal opcode vector
```

```
fffa ff 24          COPV                fdbSTART  ;COP watchdog error vector
fffc ff 24          CMONV               fdbSTART  ;Clock monitor error vector
fffe ff 20          RESETV              fdbAFTER_RST;Reset vector (Sgl chip special)

;********** end *************************************************************
```

14.8.9 Secured Mode BDM Firmware Listing

```
;*******************************************************************************
                        ;
                        ;   Copyright (C) 1999 by Motorola Inc.
                        ;                MTC S-CORE Design Group
                        ;             7600-C Capitol of Texas Highway
                        ;                    Austin, TX 78731
                        ;                 All rights reserved
                        ; No part of this software may be sold or distributed
                        ; in any form or by any means without the prior written
                        ; permission of Motorola, Inc.
                        ;
                        ;         MOTOROLA CONFIDENTIAL PROPRIETARY INFORMATION
                        ;
;*******************************************************************************
                        ; File:          secure_firm.s
                        ; Target:        STAR12 Version 1.5
                        ; Author:        John_Langan-RMAG10@email.sps.mot.com
                        ; Creation date: June 28, 1999
                        ; Comments: This code is contained in the secure ROM
                        ;           of the BDM.
;===============================================================================
                        ;                   VERSION HISTORY
                        ;
                        ; Ver 000       John Langan orig  July 02, 1999
                        ; update bug found by Lloyd, EERPOM size
                        ;                     spec changes Aug. 27, 1999
                        ;
                       ; Ver 001George Grimmer           26 July 2000
                       ; Enable BDM hardware commands when NVM erase verify fails,
                       ; BDM commands will remain disabled if Flash security bits = 01
;===============================================================================
                        ;
                        ; Design Strategy:
                        ;
                        ; This code determines if the FLASH and EEPROM are erased
                        ; If they are both erased, the program releases security,
                        ; else it hangs (branches to self).
                        ;

*********************************************************************************
                        *               Equates here

*********************************************************************************
001c                    MEMSIZ0   equ     $001C
0030                    PPAGE     equu    $0030
0012                    INITEE    equ     $0012
ff01                    BDMSTS    equ     $FF01
ff20                    BDMSTAR   equ     $FF20
fff6                    VECTORS   equ     $FFF6
*********************************************************************************
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

```
                        ; Code starts here.


ff80                            org     $FF80
ff80                    START   equ     *

                        ; Verify the FLASH is erased (all ones)

                        ;       Initialization

ff80 ce 00 00                   ldx     #$0000          ; needed for indexing
ff83 86 3f                      ldaa    #$3F
ff85 5a 30                      staa    PPAGE           ; start with last page
ff87 cc bf fe                   ldd     #$BFFE          ; last word in page

                        ;       We check every 128th word then change Page

ff8a ed e6              FLOOP   ldy     D,X             ; read word from FLASH
ff8c 02                         iny                     ; erased will become $0000
ff8d 26 36                      bne     FAIL            ; not blank -> done
ff8f 83 00 80                   subd    #$0080          ; point to next word
ff92 2b f6                      bmi     FLOOP           ; until we go under $8000

                        ;       On each succesive Page, we start at a different point
                        ;       such that if we only had one array we would check the
                        ;       entire array

ff94 c3 3f fe                   addd    #$3FFE          ; point toward end of next page
ff97 73 00 30                   dec     PPAGE           ; change to next lower page
ff9a 2a ee                      bpl      FLOOP          ; until we go under $00

                        ; Completed FLASH verify if we make it here

                        ; Verify the EEPROM is erased (all ones)

                        ;       Move EEPROM to $7800
                        ;       This will be $7000 if the size is 4K
                        ;       This will be $6000 if the size is 8K

ff9c 86 79                      ldaa    #$79            ;bit 0 is EEON
ff9e 5a 12                      staa    INITEE

                        ;       First, determine the size of the EEPROM

ffa0 d6 1c                      ldab    MEMSIZ0     ; size is encoded in bits 5 & 4
ffa2 c4 30                      andb    #$30        ; just the bits we need
ffa4 27 15                      beq     ECLEAR      ; no EEPROM, we're done!
ffa6 86 78                      ldaa    #$78        ; set up for 2K size
ffa8 c0 10              SLOOP   subb    #$10        ; 2K if clear after 1st subtract,
ffaa 27 03                      beq     EECHK       ;  2nd sub. is 4K, 3rd is 8K
ffac 48                         lsla                ; adjust for next size
ffad 20 f9                      bra     SLOOP

                        ;       Finally the erase verify loop
                        ;       Every ninth word is verified
                        ;       Accumulator D has already been set to the array size
                        ;       note that X still = 0 from earlier routines
```

```
ffaf 84 78              EECHK   anda    #$78        ; index D + X = last word
ffb1 ed e6              ELOOP   ldy     D,X         ; read word from EEPROM
ffb3 02                         iny                 ; erased will become $0000
ffb4 26 0f                      bne     FAIL        ; not blank -> done
ffb6 c3 00 12                   addd    #$0012      ; point to next word
ffb9 2a f6                      bpl     ELOOP       ; until we get to or under $4000

                        ; When we arive here, all is clear

ffbb 86 42              ECLEAR  ldaa    #$42        ; bit #1 is UNSEC
ffbd ce ff 01                   ldx     #BDMSTS
ffc0 6a 00                      staa    0,X    ; use instr that ends with write cycle
ffc2 06 ff 20                   jmp     BDMSTAR

                        ; Failures arrive here, forever.....

ffc5 18 0b 3f 00 30     FAIL    movb    #$3f,PPAGE
ffca f6 bf 0f                   ldab    $BF0F
ffcd ca fc                      orab    #$FC
ffcf ce ff 01                   ldx     #BDMSTS
ffd2 86 80                      ldaa    #$80
ffd4 aa 00                      oraa    0,x
ffd6 53                         decb
ffd7 27 03                      beq     BDMLOCK
ffd9 6a 00                      staa    0,x
ffdb a7                         align   1
                        BDMLOCK
ffdc a7                         nop
ffdd 20 fd                      bra     BDMLOCK

; Clear out space between here and the vectors

ffdf 00 00 00 00 00 00          zmb     VECTORS-*
     00 00 00 00 00 00
     00 00 00 00 00 00
     00 00 00 00 00

                        ; VECTORS HERE
fff6                            org     VECTORS
fff6 ff 24                      fdb     BDMSTAR+4   ; SWI
fff8 ff c5                      fdb     FAIL        ; TRAP
fffa ff 80                      fdb     START       ; COP
fffc ff 80                      fdb     START       ; CLK Monitor
fffe ff 80                      fdb     START       ; RESET

;*********** end ************************************************************
```

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Section 15 Secured Mode of Operation

This section provides a brief description of the secured mode of operation of the Core. Detailed information relating to integration issues is provided in the **STAR12 V1.5 Core Integration Guide**.

15.1 Overview

The implementation of the secured mode of operation for the Core provides for protecting the contents of internal (on-chip) memory arrays. While in secured mode the system can execute in single-chip mode or from an external memory block but the contents of the internal memory will not be accessible and all normal BDM functions will be blocked from execution. A mechanism is provided to release the system from the secured mode at which time normal operation will resume allowing the system to be reconfigured for unsecured mode.

15.1.1 Features

The secured mode of operation provides:

- Protection of internal (on-chip) Flash EEPROM contents
- Protection of internal (on-chip) EEPROM contents
- Operation in single-chip mode while secured
- Operation from external memory with internal Flash and EEPROM disabled while secured

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

15.1.2 Block Diagram

A block diagram of the Core security implementation is given in **Figure 15-1**.

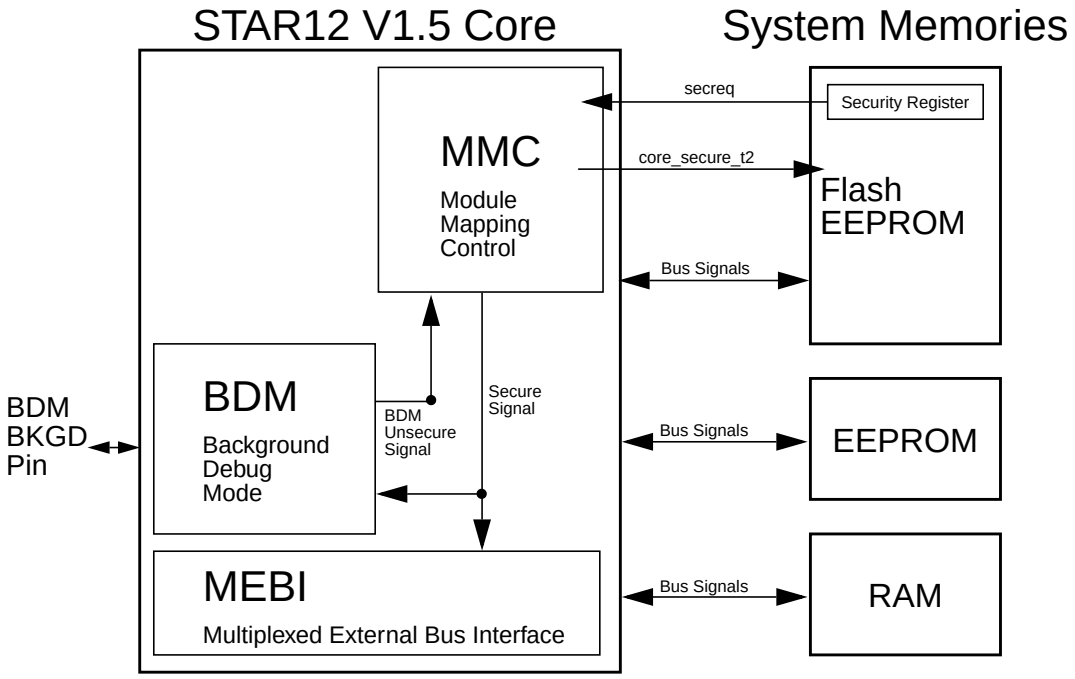


Figure 15-1 Security Implementation Block Diagram

This figure includes one example system implementation of the Core security feature. In this implementation, the Flash EEPROM block contains a security register that is programmed to the proper secured/un-secured state which generates a security request to the Core. See **15.4** for a complete description of the operation of the secured mode.

15.2 Interface Signals

The Core interface signals associated with the secured mode of operation are shown in **Table 15-1** below. The functional descriptions of the signals are provided below for completeness.

Table 15-1 Security Interface Signal Definitions

Signal Name	Type	Functional Description
core_secure_t2	O	Core secure mode signal
secreq	I	Security mode request from applicable memory

15.2.0.1 Core Secure Mode indicator (core_secure_t2)

This single bit Core output indicates that the Core is operating in secured mode.

15.2.0.2 Core Security Request (secreq)

This single bit input indicates to the Core that the system memory is in a secured state and that the Core should operate in secured mode.

15.3 Registers

There are no registers in the Core associated with the secured mode of operation. Typically, a non-volatile memory block in the system will contain a register for programming the state of system security. Please refer to the chip-level and/or memory block documentation for implementation details.

15.4 Operation

When the system is configured for secured mode of operation, it will normally operate in either normal single-chip mode or in an expanded mode executing from external memory. The conditions imposed by secured mode for each of these operating modes is discussed in the subsections that follow as well as a description of the method to unsecure the system.

15.4.1 Normal Single-Chip Mode

Normal single-chip mode will be the most common operation of a system configured for secured mode. The system functionality will appear just as an unsecured system with the exception imposed that the BDM operation will not be allowed and will be blocked. This will prevent any access to the internal non-volatile memory block contents.

15.4.2 Expanded Mode

To operate in secured mode and execute from external memory space, the system should be correctly configured for secured mode and then reset into expanded mode. The internal (on-chip) Flash EEPROM and EEPROM blocks (if applicable) will be disabled and unavailable. All BDM operation will be blocked. In addition, while in secured mode all internal visibility (IVIS) and CPU pipe (IPIPE) information will be blocked from output.

15.4.3 Unsecuring The System

To unsecure a system that is configured for secured mode, the internal (on-chip) Flash EEPROM and EEPROM must be fully erased. This can be performed using one of the following methods:

1. Reset the microcontroller into SPECIAL TEST mode, execute a program which writes the Mass Erase command sequence into the Flash and EEPROM Command registers.
2. Reset the microcontroller into SPECIAL SINGLE CHIP mode, delay while the erase test is performed by the BDM secure ROM. Send BDM commands to write the Mass Erase command sequence into the Flash and EEPROM Command registers.
3. Reset the microcontroller into SPECIAL PERIPHERAL mode, using SPM commands write the Mass Erase command sequence into the Flash and EEPROM Command registers.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

In all modes the mass erase command sequence must have the following steps:

a. Write FCLKDIV register to set the Flash clock for proper timing.
b. Write $00 to FCNFG register to select Flash block 0.
c. Write $10 to FTSTMOD register to set WRALL bit.
   (with WRALL set, all of the following writes to banked Flash registers will affect all Flash blocks.)
d. Disable Flash protection by writing the FPROT register.
e. Write any data to Flash memory space $C000-$FFFF
f. Write Mass Erase command($41) to FCMD register.
g. Clear CBIEF(bit 7) it FSTAT register.
h. Write ECLKDIV register to set the EEPROM clock for proper timing.
i. Disable protection in EEPROM by writing the EPROT register.
j. Write any data to EEPROM memory space.
k. Write Mass Erase command($41) to ECMD register.
l. Clear CBIEF(bit 7) it ESTAT register.
m. Wait until all CCIF flags are set to 1 again.

After all the CCIF flags are set to 1 again, the Flash and EEPROM have been erased. Reset the microcontroller into SPECIAL SINGLE CHIP mode. The BDM secure ROM will verify that the nonvolatile memories are erased, and then it will assert the UNSEC bit in the BDM Status register. This will cause the core_secure_t2 signal to de-assert, and the microcontroller will be unsecure. All BDM commands will be enabled and the Flash security byte may be programmed to the unsecure state by any of the following methods:

1. Send BDM commands to write to the MODE register and change to SPECIAL TEST mode, send a BDM WRITE_PC, followed by a BDM GO command to jump to a program at an external address. This external program can then program the Flash security byte to the unsecure state.
2. .Send BDM commands to directly program the Flash security byte.

   In all modes programming the security byte must have the following steps:
   a. Write FCLKDIV register to set the Flash clock for proper timing.
   b. Write $00 to FCNFG register to select Flash block 0.
   c. Disable Flash protection by writing the FPROT register.
   d. Write $FFFE to address $FF0E
   e. Write Program command($20) to FCMD register.
   f. Clear CBIEF(bit 7) it FSTAT register.
   g. Wait until Flash CCIF flag is set to 1 again.

After this Flash programming sequence is complete, the microcontroller can be reset into any mode, the Flash has been unsecured.

In normal modes, either SINGLE CHIP or EXPANDED, the microcontroller may only be unsecured by using the backdoor key access feature. This requires knowledge of the contents of the backdoor keys, which must be written to the Flash memory space at the appropriate addresses, in the correct order. In addition, in SINGLE CHIP mode the user code stored in the Flash must have a method of receiving the backdoor key from an external stimulus. This external stimulus would typically be through one of the on-chip serial ports. After the backdoor sequence has been correctly matched, the microcontroller will be

unsecured, and all Flash commands will be enabled and the Flash security byte can be programmed to the unsecure state, if desired.

Please note that if the system goes through a reset condition prior to successful configuration of unsecured mode the system will reset back into secured mode operation.

15.5 Motorola Internal Information

This subsection details information about the Core secured mode of operation that is for Motorola use only and should not be published in any form outside of Motorola.

15.5.1 BDM Secured Mode Firmware

When the Core is operating in secured mode and the system is reset into special single-chip mode, alternate BDM firmware is invoked in place of the standard BDM firmware. A listing of this secured mode firmware is given in **14.8.9** of this guide.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Appendix A Electrical Characteristics

A.1 General

This section provides electrical parametrics and electrical ratings for the IP block. If electrical characteristics are not available for a block, simulation values must be provided and identified with a warning label describing them as preliminary.

Two electrical table formats are provided as a part of this template. These can be adapted as necessary, including resizing to the width of the page

Table A-1 Parametric Table Format

Num	Characteristic	Symbol	Min	Typical	Max	Unit
1	Input High Voltage[1]	V_{IH}	0.7 (V_{DD})	—	V_{DD} + 0.3	V
2	Input Low Voltage	V_{IL}	V_{SS} – 0.3	—	0.2 (V_{DD})	V
3	Input Hysteresis	V_{HYS}	0.5	—	—	V
4	Input Leakage Current Vin = V_{DD} or V_{SS}	I_{in}	–2.5	—	2.5	µA

NOTES:
1. Notes in electricals tables are attached to the parameter or rating, rather than to the specified value.

Table A-2 Ratings Table Format

Num	Rating	Symbol	Value	Unit
1	Supply Voltage	V_{DD}	–0.3 to +6.5	V
2	Input Voltage	V_{IN}	–0.3 to +6.5	V
3	Operating Maximum Current Single pin limit	I_D	25	mA
4	Operating Maximum Current Digital Input Disruptive Current $V_{NEGCLAMP}$ = –0.3V $V_{POSCLAMP}$ = V_{DD} + 0.3V	Ii_D	–500 to +500	µA

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Appendix B Instruction Set and Commands

B.1 General

This glossary contains entries for all assembler mnemonics in alphabetical order. Each entry describes the operation of the instruction, its effect on the condition code register, and its syntax.

B.2 Glossary Notation

B.2.1 Condition Code State Notation

Table B-1 Condition Code State Notation

–	Not changed by operation
0	Cleared by operation
1	Set by operation
Δ	Set or cleared by operation
⇓	May be cleared or remain set, but not set by operation
⇑	May be set or remain cleared, but not cleared by operation
?	May be changed by operation but final state not defined
!	Used for a special purpose

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

B.2.2 Register and Memory Notation

Table B-2 Register and Memory Notation

A or *a*	Accumulator A
An	Bit n of accumulator A
B or *b*	Accumulator B
Bn	Bit n of accumulator B
D or *d*	Accumulator D
Dn	Bit n of accumulator D
X or *x*	Index register X
X_H	High byte of index register X
X_L	Low byte of index register X
Xn	Bit n of index register X
Y or *y*	Index register Y
Y_H	High byte of index register Y
Y_L	Low byte of index register Y
Yn	Bit n of index register Y
SP or *sp*	Stack pointer
SPn	Bit n of stack pointer
PC or *pc*	Program counter
PC_H	High byte of program counter
PC_L	Low byte of program counter
CCR or *c*	Condition code register
M	Address of 8-bit memory location
Mn	Bit n of byte at memory location M
Rn	Bit n of the result of an arithmetic or logical operation
In	Bit n of the intermediate result of an arithmetic or logical operation
RTN_H	High byte of return address
RTN_L	Low byte of return address
( )	Contents of

B.2.3 Address Mode Notation

Table B-3 Address Mode Notation

INH	Inherent; no operands in instruction stream
IMM	Immediate; operand immediate value in instruction stream
DIR	Direct; operand is lower byte of address from $0000 to $00FF
EXT	Operand is a 16-bit address
REL	Two's complement relative offset; for branch instructions
IDX	Indexed (no extension bytes); includes: 5-bit constant offset from X, Y, SP or PC Pre/post increment/decrement by 1–8 Accumulator A, B, or D offset
IDX1	9-bit signed offset from X, Y, SP, or PC; 1 extension byte
IDX2	16-bit signed offset from X, Y, SP, or PC; 2 extension bytes
[IDX2]	Indexed-indirect; 16-bit offset from X, Y, SP, or PC
[D, IDX]	Indexed-indirect; accumulator D offset from X, Y, SP, or PC

B.2.4 Operator Notation

Table B-4 Operator Notation

+	Add
–	Subtract
•	AND
|	OR
⊕	Exclusive OR
×	Multiply
÷	Divide
:	Concatenate
⇒	Transfer
⇔	Exchange

B.2.5 Machine Code Notation

In the **Machine Code (Hex)** column on the glossary pages, digits 0–9 and upper case letters A–F represent hexadecimal values. Pairs of lower-case letters represent 8-bit values as shown in **Table B-5**.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Table B-5 Machine Code Notation

dd	8-bit direct address from $0000 to $00FF; high byte is $00
ee	High byte of a 16-bit constant offset for indexed addressing
eb	Exchange/transfer postbyte
ff	Low eight bits of a 9-bit signed constant offset in indexed addressing, or low byte of a 16-bit constant offset in indexed addressing
hh	High byte of a 16-bit extended address
ii	8-bit immediate data value
jj	High byte of a 16-bit immediate data value
kk	Low byte of a 16-bit immediate data value
lb	Loop primitive (DBNE) postbyte
ll	Low byte of a 16-bit extended address
mm	8-bit immediate mask value for bit manipulation instructions; bits that are set indicate bits to be affected
pg	Program page or bank number used in CALL instruction
qq	High byte of a 16-bit relative offset for long branches
tn	Trap number from $30 to $39 or from $40 to $FF
rr	Signed relative offset $80 (–128) to $7F (+127) relative to the byte following the relative offset byte, or low byte of a 16-bit relative offset for long branches
xb	Indexed addressing postbyte

B.2.6 Source Form Notation

The **Source Form** column on the glossary pages gives essential information about assembler source forms. For complete information about writing source files for a particular assembler, refer to the documentation provided by the assembler vendor.

Everything in the **Source Form** column, *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 (A, B, D), are literal characters.

The groups of italic characters shown in **Table B-6** 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 *xysppc* and *oprx0_xysppc* are both valid, but the two groups *oprx0 xysppc* are not valid because there is a space between them.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table B-6 Source Form Notation

abc	Register designator for A, B, or CCR
abcdxysp	Register designator for A, B, CCR, D, X, Y, or SP
abd	Register designator for A, B, or D
abdxysp	Register designator for A, B, D, X, Y, or SP
dxysp	Register designator for D, X, Y, or SP
msk8	8-bit mask value Some assemblers require the # symbol before the mask value.
opr8i	8-bit immediate value
opr16i	16-bit immediate value
opr8a	8-bit address value used with direct address mode
opr16a	16-bit address value
oprx0_xysp	Indexed addressing postbyte code: *oprx3,–xysp* — Predecrement X , Y, or SP by 1–8 *oprx3,+xysp* — Preincrement X , Y, or SP by 1–8 *oprx3,xysp–* — Postdecrement X, Y, or SP by 1–8 *oprx3,xysp+* — Postincrement X, Y, or SP by 1–8 *oprx5,xysppc* — 5-bit constant offset from X, Y, SP, or PC *abd,xysppc* — Accumulator A, B, or D offset from X, Y, SP, or PC
oprx3	Any positive integer from 1 to 8 for pre/post increment/decrement
oprx5	Any integer from –16 to +15
oprx9	Any integer from –256 to +255
oprx16	Any integer from –32,768 to +65,535
page	8-bit value for PPAGE register Some assemblers require the # 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 the 64-Kbyte memory space
trapnum	Any 8-bit integer from $30 to $39 or from $40 to $FF
xysp	Register designator for X or Y or SP
xysppc	Register designator for X or Y or SP or PC

B.2.7 CPU Cycles Notation

The **CPU Cycles** column on the glossary pages shows how many bytes of information the CPU accesses while executing an instruction. With this information and knowledge of the type and speed of memory in the system, you can determine the execution time for any instruction in any system. Simply count the 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.

A single-letter code in represents a single CPU access cycle. An upper-case letter indicates a 16-bit access.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table B-7 CPU Cycle Notation

f	Free cycle. During an f cycle, the CPU does not use the bus. An f cycle is always one cycle of the system bus clock. An f cycle can be used by a queue controller or the background debug system to perform a single-cycle access without disturbing the CPU.
g	Read PPAGE register. A g cycle is used only in CALL instructions and is not visible on the external bus. Since PPAGE is an internal 8-bit register, a g cycle is never stretched.
I	Read indirect pointer. Indexed-indirect instructions use the 16-bit indirect pointer from memory to address the instruction operand. An I cycle is a 16-bit read that can be aligned or misaligned. An I cycle is 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. An I cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
i	Read indirect PPAGE value. An i cycle is used only in indexed-indirect CALL instructions. The 8-bit PPAGE value for the CALL destination is fetched from an indirect memory location. An i cycle is stretched only when controlled by a chip-select circuit that is programmed for slow memory.
n	Write PPAGE register. An n cycle is used only in CALL and RTC instructions to write the destination value of the PPAGE register and is not visible on the external bus. Since the PPAGE register is an internal 8-bit register, an n cycle is never stretched.
O	Optional cycle. An O cycle adjusts instruction alignment in the instruction queue. An O cycle can be a free cycle (f) or a program word access cycle (P). When the first byte of an instruction with an odd number of bytes is misaligned, the O cycle becomes a P cycle to maintain queue order. If the first byte is aligned, the O cycle is an f cycle. The $18 prebyte for a page-two opcode is treated as a special one-byte instruction. If the prebyte is misaligned, the O cycle at the beginning of the instruction becomes a P cycle to maintain queue order. If the prebyte is aligned, the O cycle is an f cycle. If the instruction has an odd number of bytes, it has a second O cycle at the end. If the first O cycle is a P cycle (prebyte misaligned), the second O cycle is an f cycle. If the first O cycle is an f cycle (prebyte aligned), the second O cycle is a P cycle. An O cycle that becomes a P cycle 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. An O cycle that becomes an f cycle is never stretched.
P	Program word access. Program information is fetched as aligned 16-bit words. A P cycle is 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. An r cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
R	16-bit data read. An R cycle is 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. An R cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
s	Stack 8-bit data. An s cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
S	Stack 16-bit data. An S cycle is 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. An S cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access. The internal RAM is designed to allow single cycle misaligned word access.

Table B-7 CPU Cycle Notation (Continued)

w	8-bit data write. A w cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
W	16-bit data write. A W cycle is 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. A W cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
u	Unstack 8-bit data. A W cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
U	Unstack 16-bit data. A U cycle is 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. A U cycle is also stretched if it corresponds to a misaligned access 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	16-bit vector fetch. Vectors are always aligned 16-bit words. A V cycle is 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. A t cycle is either a data read cycle or a free cycle, depending on the data and flow of the REVW instruction. A t cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
T	16-bit conditional read. A T cycle is either a data read cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. A T cycle is 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. A T cycle is also stretched if it corresponds to a misaligned access to a memory that is not designed for single-cycle misaligned access.
x	8-bit conditional write. An x cycle is either a data write cycle or a free cycle, depending on the data and flow of the REV or REVW instruction. An x cycle is stretched only when controlled by a chip-select circuit programmed for slow memory.
Special Notation for Branch Taken/Not Taken	
PPP/P	A short branch requires 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	A long branch requires four cycles if taken, three cycles if not taken. An O cycle is required because all long branches are page two opcodes and thus include the $18 prebyte. The prebyte is treated as a one-byte instruction. If the prebyte is misaligned, the O cycle is a P cycle; if the prebyte is aligned, the O cycle is an f cycle. As a result, both the taken and not-taken cases use one O cycle for the prebyte. In the not-taken case, the queue must advance so that execution can continue with the next instruction, and another O 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

B.3 Glossary

ABA — Add B to A — ABA

Operation (A) + (B) ⇒ A

Adds the value in B to the value in A and places the result in A. The value in B does not change. This instruction affects the H bit so it is suitable for use in BCD arithmetic operations (see DAA instruction for additional information).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: A3 • B3 | B3 • $\overline{R3}$ | $\overline{R3}$ • A3; set if there is 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 • B7 • $\overline{R7}$ | $\overline{A7}$ • $\overline{B7}$ • R7; set if the operation produces a two's complement overflow; cleared otherwise
C: A7 • B7 | B7 • $\overline{R7}$ | $\overline{R7}$ • A7; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ABA	INH	18 06	00

ABX

Add B to X
(same as LEAX B,X)

ABX

Operation (X) + (B) ⇒ X

Adds the 8-bit unsigned value in B to the value in X considering the possible carry out of the low byte of X and places the result in X. The value in B does not change.

ABX assembles as LEAX B,X. The LEAX instruction allows A, B, D, or a constant to be added to X.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ABX	IDX	1A E5	Pf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ABY

Add B to Y
(same as LEAY B,Y)

ABY

Operation (Y) + (B) ⇒ Y

Adds the 8-bit unsigned value in B to the value in Y considering the possible carry out of the low byte of Yand places the result in Y. The value in B does not change.

ABY assembles as LEAY B,Y. The LEAY instruction allows A, B, D, or a constant to be added to Y.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ABY	IDX	19 ED	Pf

ADCA Add with Carry to A ADCA

Operation

(A) + (M) + C ⇒ A
or
(A) + imm + C ⇒ A

Adds either the value in M and the C bit or an immediate value and the C bit to the value in A. Puts the result in A. This instruction affects the H bit, so it is suitable for use in BCD arithmetic operations (see DAA instruction for additional information).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: A3 • M3 | M3 • $\overline{R3}$ | $\overline{R3}$ • A3; set if there is 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 • M7 • $\overline{R7}$ | $\overline{A7}$ • $\overline{M7}$ • R7; set if the operation produces a two's complement overflow; cleared otherwise
C: A7 • M7 | M7 • $\overline{R7}$ | $\overline{R7}$ • A7; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ADCA #*opr8i*	IMM	89 ii	P
ADCA *opr8a*	DIR	99 dd	rPf
ADCA *opr16a*	EXT	B9 hh ll	rPO
ADCA *oprx0_xysppc*	IDX	A9 xb	rPf
ADCA *oprx9,xysppc*	IDX1	A9 xb ff	rPO
ADCA *oprx16,xysppc*	IDX2	A9 xb ee ff	frPP
ADCA [D,*xysppc*]	[D,IDX]	A9 xb	fIfrPf
ADCA [*oprx16,xysppc*]	[IDX2]	A9 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ADCB Add with Carry to B ADCB

Operation

(B) + (M) + C ⇒ B
or
(B) + imm + C ⇒ B

Adds either the value in M and the C bit or an immediate value and the C bit to the value in B. Puts the result in B. This instruction affects the H bit, so it is suitable for use in BCD arithmetic operations (see DAA instruction for additional information).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: B3 • M3 | M3 • $\overline{R3}$ | $\overline{R3}$ • B3; set if there is a carry from bit 3; cleared otherwise

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

Z: Set if result is $00; cleared otherwise

V: B7 • M7 • $\overline{R7}$ | $\overline{B7}$ • $\overline{M7}$ • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: B7 • M7 | M7 • $\overline{R7}$ | $\overline{R7}$ • B7; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ADCB #*opr8i*	IMM	C9 ii	P
ADCB *opr8a*	DIR	D9 dd	rPf
ADCB *opr16a*	EXT	F9 hh ll	rPO
ADCB *oprx0_xysppc*	IDX	E9 xb	rPf
ADCB *oprx9,xysppc*	IDX1	E9 xb ff	rPO
ADCB *oprx16,xysppc*	IDX2	E9 xb ee ff	frPP
ADCB [D,*xysppc*]	[D,IDX]	E9 xb	fIfrPf
ADCB [*oprx16*,*xysppc*]	[IDX2]	E9 xb ee ff	fIPrPf

ADDA Add to A ADDA

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

(A) + (M) ⇒ A
or
(A) + imm ⇒ A

Adds either the value in M or an immediate value to the value in A and places the result in A. This instruction affects the H bit, so it is suitable for use in BCD arithmetic operations (see DAA instruction for additional information).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: A3 • M3 | M3 • $\overline{R3}$ | $\overline{R3}$ • A3; set if there is 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 • M7 • $\overline{R7}$ | $\overline{A7}$ • $\overline{M7}$ • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: A7 • M7 | M7 • $\overline{R7}$ | $\overline{R7}$ • A7; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ADDA #*opr8i*	IMM	8B ii	P
ADDA *opr8a*	DIR	9B dd	rPf
ADDA *opr16a*	EXT	BB hh ll	rPO
ADDA *oprx0_xysppc*	IDX	AB xb	rPf
ADDA *oprx9,xysppc*	IDX1	AB xb ff	rPO
ADDA *oprx16,xysppc*	IDX2	AB xb ee ff	frPP
ADDA [D,*xysppc*]	[D,IDX]	AB xb	fIfrPf
ADDA [*oprx16,xysppc*]	[IDX2]	AB xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ADDB — Add to B — ADDB

Operation

(B) + (M) ⇒ B
or
(B) + imm ⇒ B

Adds either the value in M or an immediate value to the value in B and places the result in B. This instruction affects the H bit, so it is suitable for use in BCD arithmetic operations (see DAA instruction for additional information).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

H: B3 • M3 | M3 • $\overline{R3}$ | $\overline{R3}$ • B3; set if there is a carry from bit 3; cleared otherwise

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

Z: Set if result is $00; cleared otherwise

V: B7 • M7 • $\overline{R7}$ | $\overline{B7}$ • $\overline{M7}$ • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: B7 • M7 | M7 • $\overline{R7}$ | $\overline{R7}$ • B7; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ADDB #*opr8i*	IMM	CB ii	P
ADDB *opr8a*	DIR	DB dd	rPf
ADDB *opr16a*	EXT	FB hh ll	rPO
ADDB *oprx0_xysppc*	IDX	EB xb	rPf
ADDB *oprx9,xysppc*	IDX1	EB xb ff	rPO
ADDB *oprx16,xysppc*	IDX2	EB xb ee ff	frPP
ADDB [D,*xysppc*]	[D,IDX]	EB xb	fIfrPf
ADDB [*oprx16*,*xysppc*]	[IDX2]	EB xb ee ff	fIPrPf

ADDD — Add to D — ADDD

Operation

(A):(B) + (M):(M + 1) ⇒ A:B
or
(A):(B) + imm ⇒ A:B

Adds either the value in M concatenated with the value in M + 1 or an immediate value to the value in D. Puts the result in D. A is the high byte of D; B is the low byte.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $0000; cleared otherwise

V: D15 • M15 • $\overline{R15}$ | $\overline{D15}$ • $\overline{M15}$ • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: D15 • M15 | M15 • $\overline{R15}$ | $\overline{R15}$ • D15; set if there is a carry from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ADDD #*opr16i*	IMM	C3 jj kk	PO
ADDD *opr8a*	DIR	D3 dd	RPf
ADDD *opr16a*	EXT	F3 hh ll	RPO
ADDD *oprx0_xysppc*	IDX	E3 xb	RPf
ADDD *oprx9,xysppc*	IDX1	E3 xb ff	RPO
ADDD *oprx16,xysppc*	IDX2	E3 xb ee ff	fRPP
ADDD [D,*xysppc*]	[D,IDX]	E3 xb	fIfRPf
ADDD [*oprx16*,*xysppc*]	[IDX2]	E3 xb ee ff	fIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ANDA AND with A ANDA

Operation

(A) • (M) ⇒ A
or
(A) • imm ⇒ A

Performs a logical AND of either the value in M or an immediate value with the value in A. Puts the result in A.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ANDA #*opr8i*	IMM	84 ii	P
ANDA *opr8a*	DIR	94 dd	rPf
ANDA *opr16a*	EXT	B4 hh ll	rPO
ANDA *oprx0_xysppc*	IDX	A4 xb	rPf
ANDA *oprx9,xysppc*	IDX1	A4 xb ff	rPO
ANDA *oprx16,xysppc*	IDX2	A4 xb ee ff	frPP
ANDA [D,*xysppc*]	[D,IDX]	A4 xb	fIfrPf
ANDA [*oprx16,xysppc*]	[IDX2]	A4 xb ee ff	fIPrPf

ANDB AND with B ANDB

Operation

(B) • (M) ⇒ B
or
(B) • imm ⇒ B

Performs a logical AND of either the value in M or an immediate value with the value in B. Puts the result in B.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ANDB #*opr8i*	IMM	C4 ii	P
ANDB *opr8a*	DIR	D4 dd	rPf
ANDB *opr16a*	EXT	F4 hh ll	rPO
ANDB *oprx0_xysppc*	IDX	E4 xb	rPf
ANDB *oprx9,xysppc*	IDX1	E4 xb ff	rPO
ANDB *oprx16,xysppc*	IDX2	E4 xb ee ff	frPP
ANDB [D,*xysppc*]	[D,IDX]	E4 xb	fIfrPf
ANDB [*oprx16*,*xysppc*]	[IDX2]	E4 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ANDCC AND with CCR ANDCC

Operation (CCR) • imm ⇒ CCR

Performs a logical AND of an immediate value and the value in the CCR. Puts the result in the CCR.

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

CCR Effects

S	X	H	I	N	Z	V	C
⇓	⇓	⇓	⇓	⇓	⇓	⇓	⇓

All CCR bits: Clear if 0 before operation or if corresponding bit in mask is 0

Code and CPU Cycles

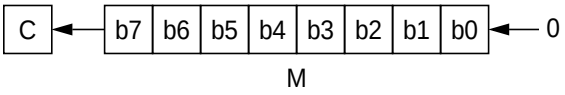
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ANDCC #*opr8i*	IMM	10 ii	P

ASL

Arithmetic Shift Left M
(same as LSL)

ASL

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASL *opr16a*	EXT	78 hh ll	rPw0
ASL *oprx0_xysppc*	IDX	68 xb	rPw
ASL *oprx9,xysppc*	IDX1	68 xb ff	rPw0
ASL *oprx16,xysppc*	IDX2	68 xb ee ff	frPwP
ASL [D,*xysppc*]	[D,IDX]	68 xb	fIfrPw
ASL [*oprx16,xysppc*]	[IDX2]	68 xb ee ff	fIPrPw

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

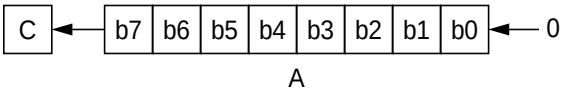
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ASLA

Arithmetic Shift Left A
(same as LSLA)

ASLA

Operation



Shifts all bits of A one bit position to the left. Bit 0 is loaded with a 0. The C bit is loaded from the most significant bit of A.

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

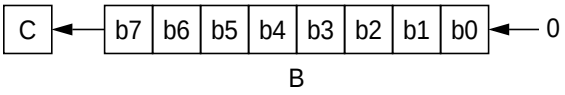
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASLA	INH	48	0

ASLB

Arithmetic Shift Left B
(same as LSLB)

ASLB

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASLB	INH	58	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

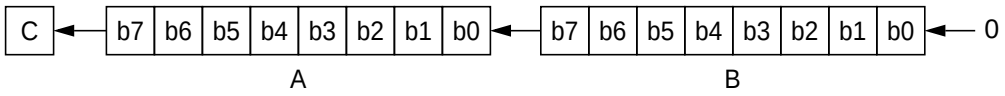
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ASLD

Arithmetic Shift Left D
(same as LSLD)

ASLD

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

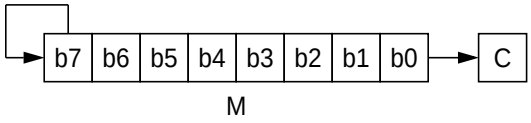
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASLD	INH	59	0

ASR

Arithmetic Shift Right M

ASR

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

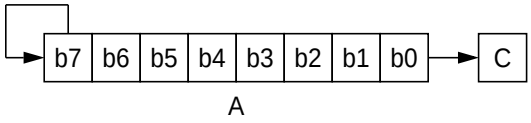
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASR *opr16a*	EXT	77 hh ll	rPw0
ASR *oprx0_xysppc*	IDX	67 xb	rPw
ASR *oprx9,xysppc*	IDX1	67 xb ff	rPw0
ASR *oprx16,xysppc*	IDX2	67 xb ee ff	frPwP
ASR [D,*xysppc*]	[D,IDX]	67 xb	fIfrPw
ASR [*oprx16,xysppc*]	[IDX2]	67 xb ee ff	fIPrPw

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ASRA — Arithmetic Shift Right A — ASRA

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

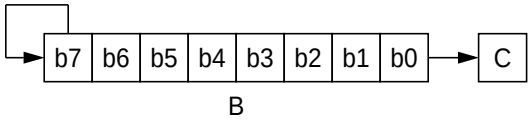
C: A0; set if the LSB of A was set before the shift; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASRA	INH	47	0

ASRB Arithmetic Shift Right B ASRB

Operation



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

CCR Effects

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 ⊕ C; set if:
N is set and C is cleared after the shift, or
N is cleared and C is set after the shift; cleared otherwise

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ASRB	INH	57	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BCC

Branch if C Clear
(same as BHS)

BCC

Operation

If C = 0, then (PC) + $0002 + rel ⇒ PC

Tests the C bit and branches if C = 0.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BCC *rel8*	REL	24 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BCC/BHS	24	(R) ≥ (M) or (B) ≥ (A)	BCS/BLO	25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	
BGE	2C	(R) ≥ (M) or (B) ≥ (A)	BLT	2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	

BCLR Clear Bit(s) in M BCLR

Operation (M) • ($\overline{\text{mask byte}}$) ⇒ M

Performs a logical AND of the value in M and the complement of a mask byte contained in the instruction. Puts the result in M. Bits in M that correspond to 1s in the mask byte are cleared. No other bits in M change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode[1]	Machine Code (Hex)	CPU Cycles
BCLR *opr8a*, *msk8*	DIR	4D dd mm	rPwO
BCLR *opr16a*, *msk8*	EXT	1D hh ll mm	rPwP
BCLR *oprx0_xysppc*, *msk8*	IDX	0D xb mm	rPwO
BCLR *oprx9*,*xysppc*, *msk8*	IDX1	0D xb ff mm	rPwP
BCLR *oprx16*,*xysppc*, *msk8*	IDX2	0D xb ee ff mm	frPwPO

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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BCS

Branch if C Set
(same as BLO)

BCS

Operation

If C = 1, then (PC) + $0002 + rel ⇒ PC

Tests the C bit and branches if C = 1.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BCS *rel8*	REL	25 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BCS/BLO	25	(R) < (M) or (B) < (A)	BCC/BHS	24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	
BLT	2D	(R) < (M) or (B) < (A)	BGE	2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	

BEQ Branch if Equal BEQ

Operation

If Z = 1, then (PC) + $0002 + rel ⇒ PC

Tests the Z bit and branches if Z = 1.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BEQ *rel8*	REL	27 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BEQ	27	(R) = (M) or (R) = zero	BNE	26	(R) ≠ (M) or (R) ≠ zero	Signed, unsigned or simple
		Z = 1			Z = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BGE Branch if Greater Than or Equal to Zero BGE

Operation

If N ⊕ V = 0, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BGE *rel8*	REL	2C rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BGE	2C	(R) ≥ (M) or (B) ≥ (A)	BLT	2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	
BHS/BCC	24	(R) ≥ (M) or (B) ≥ (A)	BLO/BCS	25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	

BGND Enter Background Debug Mode BGND

Operation

(PC) ⇒ TMP2
BDM vector ⇒ PC

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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BGND	INH	00	VfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BGT Branch if Greater Than Zero BGT

Operation

If Z | (N ⊕ V) = 0, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BGT *rel8*	REL	2E `rr`	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BGT	2E	(R) > (M) or (B) > (A)	BLE	2F	(R) ≤ (M) or (B) ≤ (A)	Signed
		Z | (N ⊕ V) = 0			Z | (N ⊕ V) = 1	
BHI	22	(R) > (M) or (B) > (A)	BLS	23	(R) ≤ (M) or (B) ≤ (A)	Unsigned
		C | Z = 0			C | Z = 1	

BHI Branch if Higher BHI

Operation

If C | Z = 0, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BHI *rel8*	REL	22 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BHI	22	(R) > (M) or (B) > (A)	BLS	23	(R) ≤ (M) or (B) ≤ (A)	Unsigned
		C | Z = 0			C | Z = 1	
BGT	2E	(R) > (M) or (B) > (A)	BLE	2F	(R) ≤ (M) or (B) ≤ (A)	Signed
		Z | (N ⊕ V) = 0			Z | (N ⊕ V) = 1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BHS

Branch if Higher or Same
(same as BCC)

BHS

Operation

If C = 0, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BHS *rel8*	REL	24 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BHS/BCC	24	(R) ≥ (M) or (B) ≥ (A)	BLO/BCS	25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	
BGE	2C	(R) ≥ (M) or (B) ≥ (A)	BLT	2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	

BITA Bit Test A BITA

Operation

(A) • (M)
or
(A) • imm

Performs a logical AND of either the value in M or an immediate value with the value in A. CCR bits reflect the result. The values in A and M do not change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BITA #*opr8i*	IMM	85 ii	P
BITA *opr8a*	DIR	95 dd	rPf
BITA *opr16a*	EXT	B5 hh ll	rPO
BITA *oprx0_xysppc*	IDX	A5 xb	rPf
BITA *oprx9,xysppc*	IDX1	A5 xb ff	rPO
BITA *oprx16,xysppc*	IDX2	A5 xb ee ff	frPP
BITA [D,*xysppc*]	[D,IDX]	A5 xb	fIfrPf
BITA [*oprx16,xysppc*]	[IDX2]	A5 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BITB — Bit Test B — BITB

Operation

(B) • (M)
or
(B) • imm

Performs a logical AND of either the value in M or an immediate value with the value in B. CCR bits reflect the result. The values in B and M do not change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BITB #*opr8i*	IMM	C5 ii	P
BITB *opr8a*	DIR	D5 dd	rPf
BITB *opr16a*	EXT	F5 hh ll	rPO
BITB *oprx0_xysppc*	IDX	E5 xb	rPf
BITB *oprx9,xysppc*	IDX1	E5 xb ff	rPO
BITB *oprx16,xysppc*	IDX2	E5 xb ee ff	frPP
BITB [D,*xysppc*]	[D,IDX]	E5 xb	fIfrPf
BITB [*oprx16*,*xysppc*]	[IDX2]	E5 xb ee ff	fIPrPf

BLE Branch if Less Than or Equal to Zero BLE

Operation

If Z | (N ⊕ V) = 1, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Object Code	CPU Cycles
BLE *rel8*	REL	2F `rr`	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BLE	2F	(R) ≤ (M) or (B) ≤ (A)	BGT	2E	(R) > (M) or (B) > (A)	Signed
		Z | (N ⊕ V) = 1			Z | (N ⊕ V) = 0	
BLS	23	(R) ≤ (M) or (B) ≤ (A)	BHI	22	(R) > (M) or (B) > (A)	Unsigned
		C | Z = 1			C | Z = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BLO

Branch if Lower
(same as BCS)

BLO

Operation

If C = 1, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BLO *rel8*	REL	25 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BLO/BCS	25	(R) < (M) or (B) < (A)	BHS/BCC	24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	
BLT	2D	(R) < (M) or (B) < (A)	BGE	2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	

BLS Branch if Lower or Same BLS

Operation

If C | Z = 1, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BLS *rel8*	REL	23 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BLS	23	(R) ≤ (M) or (B) ≤ (A)	BHI	22	(R) > (M) or (B) > (A)	Unsigned
		C | Z = 1			C | Z = 0	
BLE	2F	(R) ≤ (M) or (B) ≤ (A)	BGT	2E	(R) > (M) or (B) > (A)	Signed
		Z | (N ⊕ V) = 1			Z | (N ⊕ V) = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BLT — Branch if Less Than Zero — BLT

Operation

If N ⊕ V = 1, then (PC) + $0002 + rel ⇒ PC

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

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BLT *rel8*	REL	2D rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BLT	2D	(R) < (M) or (B) < (A)	BGE	2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	
BLO/BCS	25	(R) < (M) or (B) < (A)	BHS/BCC	24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	

BMI Branch if Minus BMI

Operation

If N = 1, then (PC) + $0002 + rel ⇒ PC

Tests the N bit and branches if N = 1.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BMI *rel8*	REL	2B `rr`	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BMI	2B	Negative N = 1	BPL	2A	Positive N = 0	Simple

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BNE — Branch if Not Equal to Zero — BNE

Operation

If Z = 0, then (PC) + $0002 + rel ⇒ PC

Tests the Z bit and branches if Z = 0.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BNE *rel8*	REL	26 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BNE	26	(R) ≠ (M) or (R) ≠ zero	BEQ	27	(R) = (M) or (R) = zero	Signed, unsigned, or simple
		Z = 0			Z = 1	

BPL Branch if Plus BPL

Operation

If N = 0, then (PC) + $0002 + rel ⇒ PC

Tests the N bit and branches if N = 0.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	Source Form
BPL *rel8*	REL	2A `rr`	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BPL	2A	Positive N = 0	BMI	2B	Negative N = 1	Simple

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BRA Branch Always BRA

Operation

(PC) + $0002 + rel ⇒ PC

Branches unconditionally.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

Execution time is longer when a conditional branch is taken than when it is not taken, 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BRA *rel8*	REL	20 rr	PPP

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BRA	20	Always	BRN	21	Never	Simple

BRCLR Branch if Bit(s) Clear BRCLR

Operation

If (M) • (mask byte) = 0, then (PC) + $0002 + rel ⇒ PC

Performs a logical AND of the value in M and the mask value supplied with the instruction. Branches if all the 0s in M correspond to 1s in the mask byte.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BRCLR *opr8a*, *msk8*, *rel8*	DIR	4F dd mm rr	rPPP
BRCLR *opr16a*, *msk8*, *rel8*	EXT	1F hh ll mm rr	rfPPP
BRCLR *oprx0_xysppc*, *msk8*, *rel8*	IDX	0F xb mm rr	rPPP
BRCLR *oprx9,xysppc*, *msk8*, *rel8*	IDX1	0F xb ff mm rr	rfPPP
BRCLR *oprx16,xysppc*, *msk8*, *rel8*	IDX2	0F xb ee ff mm rr	PrfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BRN Branch Never BRN

Operation

(PC) + $0002 ⇒ PC

Never branches. BRN is effectively a 2-byte NOP that requires one cycle. BRN is included in the instruction set to provide a complement to the BRA instruction. BRN 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BRN *rel8*	REL	21 rr	P

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BRN	21	Never	BRA	20	Always	Simple

BRSET Branch if Bit(s) Set BRSET

Operation

If ($\overline{M}$) • (mask byte) = 0, then (PC) + $0002 + rel ⇒ PC

Performs a logical AND of the value of $\overline{M}$ and the mask value supplied with the instruction. Branches if all the ones in $\overline{M}$ correspond to ones in the mask byte.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BRSET *opr8a*, *msk8*, *rel8*	DIR	4E dd mm rr	rPPP
BRSET *opr16a*, *msk8*, *rel8*	EXT	1E hh ll mm rr	rfPPP
BRSET *oprx0_xysppc*, *msk8, rel8*	IDX	0E xb mm rr	rPPP
BRSET *oprx9,xysppc*, *msk8*, *rel8*	IDX1	0E xb ff mm rr	rfPPP
BRSET *oprx16,xysppc*, *msk8*, *rel8*	IDX2	0E xb ee ff mm rr	PrfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BSET — Set Bit(s) in M — BSET

Operation

(M) | (mask byte) ⇒ M

Performs a logical OR of the value in M and a mask byte contained in the instruction. Puts the result in M. Bits in M that correspond to 1s in the mask are set. No other bits in M change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BSET *opr8a*, *msk8*	DIR	4C dd mm	rPwO
BSET *opr16a*, *msk8*	EXT	1C hh ll mm	rPwP
BSET *oprx0_xysppc*, *msk8*	IDX	0C xb mm	rPwO
BSET *oprx9*,*xysppc*, *msk8*	IDX1	0C xb ff mm	rPwP
BSET *oprx16*,*xysppc*, *msk8*	IDX2	0C xb ee ff mm	frPwPO

BSR Branch to Subroutine BSR

Operation

(SP) – $0002 ⇒ SP
RTN_H:RTN_L ⇒ M_{SP}:$M_{SP + 1}$
(PC) + $0002 + rel ⇒ PC

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 byte of the return address).

Branches to a location determined by the branch offset.

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

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BSR *rel8*	REL	07 rr	SPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

BVC — Branch if V Clear — BVC

Operation

If V = 0, then (PC) + $0002 + rel ⇒ PC

Tests the V 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.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BVC *rel8*	REL	28 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BRN	21	Never	BRA	20	Always	Simple

BVS Branch if V Set BVS

Operation

If V = 1, then (PC) + $0002 + rel ⇒ PC

Tests the V bit and branches if V = 1. BVS causes a branch when a previous operation on two's complement 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.

Rel is an 8-bit two's complement offset for branching forward or backward in memory. Branching range is $80 to $7F (–128 to 127) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
BVS *rel8*	REL	29 rr	PPP (branch) P (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
BVS	29	No overflow V = 1	BVC	28	Overflow V = 1	Simple

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CALL Call Subroutine in Expanded Memory CALL

Operation

(SP) – $0002 ⇒ SP
RTN_H:RTN_L ⇒ M_{SP}:$M_{SP + 1}$
(SP) – $0001 ⇒ SP
(PPAGE) ⇒ M_{SP}
new page value ⇒ PPAGE
Subroutine address ⇒ PC

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 SP by two, allowing the two return address bytes to be stacked.

Stacks the return address; SP points to the high byte of the return address.

Decrements SP by one, allowing the current PPAGE value to be stacked.

Stacks the value in 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 PPAGE value are fetched in the order M high byte, M low byte, and new PPAGE value.

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

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CALL *opr16a*, *page*	EXT	4A hh ll pg	gnSsPPP
CALL *oprx0_xysppc*, *page*	IDX	4B xb pg	gnSsPPP
CALL *oprx9*,*xysppc*, *page*	IDX1	4B xb ff pg	gnSsPPP
CALL *oprx16*,*xysppc*, *page*	IDX2	4B xb ee ff pg	fgnSsPPP
CALL [D,*xysppc*]	[D,IDX]	4B xb	fIignSsPPP
CALL [*oprx16*,*xysppc*]	[IDX2]	4B xb ee ff	fIignSsPPP

CBA Compare B to A CBA

Operation

(A) – (B)

Compares the value in A with the value in B. Condition code bits affected by the comparison can be used for conditional branches. The values in A and B do not change.

CCR Effects

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 • $\overline{B7}$ • $\overline{R7}$ | $\overline{A7}$ • B7 • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • B7 | B7 • R7 | R7 | $\overline{A7}$; set if there is a borrow from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CBA	INH	18 17	00

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CLC

Clear C
(same as ANDCC #$FE)

CLC

Operation $0 \Rightarrow$ C bit

Clears the C bit. CLC assembles as ANDCC #$FE.

CLC can be used to initialize the C bit prior to a shift or rotate instruction affecting the C bit.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	0

C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLC	IMM	10 FE	P

CLI

Clear I
(same as ANDCC #$EF)

CLI

Operation $0 \Rightarrow$ I bit

Clears the I bit. CLI assembles as ANDCC #$EF.

Clearing the I bit enables interrupts. There is a one-cycle bus clock delay in the clearing mechanism. If interrupts were previously disabled, the next instruction after a CLI is always executed, even if there was an interrupt pending prior to execution of the CLI instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	0	–	–	–	–

I: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLI	IMM	10 EF	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CLR — Clear M — CLR

Operation

$\$00 \Rightarrow M$

Clears all bits in M.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: Cleared
Z: Set
V: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLR *opr16a*	EXT	79 hh ll	PwO
CLR *oprx0_xysppc*	IDX	69 xb	Pw
CLR *oprx9*,*xysppc*	IDX1	69 xb ff	PwO
CLR *oprx16*,*xysppc*	IDX2	69 xb ee ff	PwP
CLR [D,*xysppc*]	[D,IDX]	69 xb	PIfw
CLR [*oprx16*,*xysppc*]	[IDX2]	69 xb ee ff	PIPw

CLRA Clear A CLRA

Operation $\$00 \Rightarrow A$

Clears all bits in A.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: Cleared
Z: Set
V: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLRA	INH	87	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CLRB Clear B CLRB

Operation $00 ⇒ B

Clears all bits in B.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: Cleared
Z: Set
V: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLRB	INH	C7	0

CLV

Clear V
(same as ANDCC #$FD)

CLV

Operation $0 \Rightarrow$ V bit

Clears the V bit. CLV assembles as ANDCC #$FD.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	0	–

V: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CLV	IMM	10 FD	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CMPA Compare A CMPA

Operation

(A) – (M)
or
(A) – imm

Compares the value in A to either the value in M or an immediate value. CCR bits reflect the result. The values in A and M do not change.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if there is a borrow from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CMPA #*opr8i*	IMM	81 ii	P
CMPA *opr8a*	DIR	91 dd	rPf
CMPA *opr16a*	EXT	B1 hh ll	rPO
CMPA *oprx0_xysppc*	IDX	A1 xb	rPf
CMPA *oprx9,xysppc*	IDX1	A1 xb ff	rPO
CMPA *oprx16,xysppc*	IDX2	A1 xb ee ff	frPP
CMPA [D,*xysppc*]	[D,IDX]	A1 xb	fIfrPf
CMPA [*oprx16*,*xysppc*]	[IDX2]	A1 xb ee ff	fIPrPf

CMPB Compare B CMPB

Operation

(B) – (M)
or
(B) – imm

Compares the value in B to either the value in M or an immediate value. CCR bits reflect the result. The values in B and M do not change.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

V: B7 • $\overline{M7}$ • $\overline{R7}$ | $\overline{B7}$ • M7 • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{B7}$ • M7 | M7 • R7 | R7 • $\overline{B7}$; set if there is a borrow from the MSB of the result; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CMPB #*opr8i*	IMM	C1 ii	P
CMPB *opr8a*	DIR	D1 dd	rPf
CMPB *opr16a*	EXT	F1 hh ll	rPO
CMPB *oprx0_xysppc*	IDX	E1 xb	rPf
CMPB *oprx9,xysppc*	IDX1	E1 xb ff	rPO
CMPB *oprx16,xysppc*	IDX2	E1 xb ee ff	frPP
CMPB [D,*xysppc*]	[D,IDX]	E1 xb	fIfrPf
CMPB [*oprx16,xysppc*]	[IDX2]	E1 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

COM — Complement M — COM

Operation

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

Replaces the value in M with its one's complement. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two's complement values, all signed branches are available.

CCR Effects

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: Cleared
C: Set for M6800 compatibility

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
COM *opr16a*	EXT	71 hh ll	rPw0
COM *oprx0_xysppc*	IDX	61 xb	rPw
COM *oprx9,xysppc*	IDX1	61 xb ff	rPw0
COM *oprx16,xysppc*	IDX2	61 xb ee ff	frPwP
COM [D,*xysppc*]	[D,IDX]	61 xb	fIfrPw
COM [*oprx16,xysppc*]	[IDX2]	61 xb ee ff	fIPrPw

COMA Complement A COMA

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

Replaces the value in A with its one's complement. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two's complement values, all signed branches are available.

CCR Effects

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: Cleared
C: Set for M6800 compatibility

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
COMA	INH	41	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

COMB Complement B COMB

Operation

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

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

CCR Effects

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: Cleared
C: Set for M6800 compatibility

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
COMB	INH	51	0

CPD — Compare D — CPD

Operation

(A):(B) – (M):(M + 1)
or
(A:B) – imm

Compares the value in D to either the value in M:M + 1 or an immediate value. CCR bits reflect the result. The values in D and M:M + 1 do not change.

CCR Effects

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 • $\overline{M15}$ • $\overline{R15}$ | $\overline{D15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if the absolute value of (M:M + 1) is larger than the absolute value of (D); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CPD #*opr16i*	IMM	8C jj kk	PO
CPD *opr8a*	DIR	9C dd	RPf
CPD *opr16a*	EXT	BC hh ll	RPO
CPD *oprx0_xysppc*	IDX	AC xb	RPf
CPD *oprx9*,*xysppc*	IDX1	AC xb ff	RPO
CPD *oprx16*,*xysppc*	IDX2	AC xb ee ff	fRPP
CPD [D,*xysppc*]	[D,IDX]	AC xb	fIfRPf
CPD [*oprx16*,*xysppc*]	[IDX2]	AC xb ee ff	fIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CPS — Compare SP — CPS

Operation

(SP) – (M):(M + 1)
or
(SP) – imm

Compares the value in SP to either the value in M:M + 1 or an immediate value. CCR bits reflect the result. The values in SP and M:M + 1 do not change.

CCR Effects

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: SP15 • $\overline{M15}$ • $\overline{R15}$ | $\overline{SP15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{SP15}$ • M15 | M15 • R15 | R15 • $\overline{SP15}$; set if the absolute value of (M:M + 1) is larger than the absolute value of (SP); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CPS #*opr16i*	IMM	8F jj kk	PO
CPS *opr8a*	DIR	9F dd	RPf
CPS *opr16a*	EXT	BF hh ll	RPO
CPS *oprx0_xysppc*	IDX	AF xb	RPf
CPS *oprx9*,*xysppc*	IDX1	AF xb ff	RPO
CPS *oprx16*,*xysppc*	IDX2	AF xb ee ff	fRPP
CPS [D,*xysppc*]	[D,IDX]	AF xb	fIfRPf
CPS [*oprx16*,*xysppc*]	[IDX2]	AF xb ee ff	fIPRPf

CPX — Compare X — CPX

Operation

(X) – (M):(M + 1)
or
(X) – imm

Compares the value in X to either the value in M:M + 1 or an immediate value. CCR bits reflect the result. The values in X and M:M + 1 do not change.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $0000; cleared otherwise

V: X15 • $\overline{M15}$ • $\overline{R15}$ | $\overline{X15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{X15}$ • M15 | M15 • R15 | R15 • $\overline{X15}$; set if the absolute value of (M:M + 1) is larger than the absolute value of (X); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CPX #*opr16i*	IMM	8E jj kk	PO
CPX *opr8a*	DIR	9E dd	RPf
CPX *opr16a*	EXT	BE hh ll	RPO
CPX *oprx0_xysppc*	IDX	AE xb	RPf
CPX *oprx9*,*xysppc*	IDX1	AE xb ff	RPO
CPX *oprx16*,*xysppc*	IDX2	AE xb ee ff	fRPP
CPX [D,*xysppc*]	[D,IDX]	AE xb	fIfRPf
CPX [*oprx16*,*xysppc*]	[IDX2]	AE xb ee ff	fIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

CPY — Compare Y — CPY

Operation

(Y) – (M):(M + 1)
or
(Y) – imm

Compares the value in Y to either the value in M:M + 1 or an immediate value. CCR bits reflect the result. The values in Y and M:M + 1 do not change.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $0000; cleared otherwise

V: Y15 • $\overline{M15}$ • $\overline{R15}$ | $\overline{Y15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{Y15}$ • M15 | M15 • R15 | R15 • $\overline{Y15}$; set if the absolute value of (M:M + 1) is larger than the absolute value of (Y); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
CPY #*opr16i*	IMM	8D jj kk	PO
CPY *opr8a*	DIR	9D dd	RPf
CPY *opr16a*	EXT	BD hh ll	RPO
CPY *oprx0_xysppc*	IDX	AD xb	RPf
CPY *oprx9*,*xysppc*	IDX1	AD xb ff	RPO
CPY *oprx16*,*xysppc*	IDX2	AD xb ee ff	fRPP
CPY [D,*xysppc*]	[D,IDX]	AD xb	fIfRPf
CPY [*oprx16*,*xysppc*]	[IDX2]	AD xb ee ff	fIPRPf

DAA Decimal Adjust A for BCD DAA

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

DAA adjusts the value in A and the state of the C bit to represent the correct binary-coded-decimal (BCD) sum and the associated carry when a BCD calculation is performed. To execute DAA, the value in A, the state of the C bit, and the state of the H bit must all be the result of performing an ABA, ADD, or ADC on BCD operands, with or without an initial carry.

The table below shows DAA operation for all legal combinations of input operands. The first four columns represent the results of ABA, ADC, or ADD operations on BCD operands. The correction factor in the fifth column 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.

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

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

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

Z: Set if result is $00; cleared otherwise

C: Represents BCD carry

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DAA	INH	18 07	Ofo

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

DBEQ Decrement and Branch if Equal to Zero DBEQ

Operation

(counter) – 1 ⇒ counter
If (counter) = 0, then (PC) + $0003 + rel ⇒ PC

Subtracts one from the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register reaches zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DBEQ *abdxysp*, *rel9*	REL (9-bit)	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding				
Source Form	**Postbyte[1]**	**Object Code**	**Counter Register**	**Offset**
DBEQ A, *rel9*	0000 X000	`04 00 rr`	A	Positive
DBEQ B, *rel9*	0000 X001	`04 01 rr`	B	
DBEQ D, *rel9*	0000 X100	`04 04 rr`	D	
DBEQ X, *rel9*	0000 X101	`04 05 rr`	X	
DBEQ Y, *rel9*	0000 X110	`04 06 rr`	Y	
DBEQ SP, *rel9*	0000 X111	`04 07 rr`	SP	
DBEQ A, *rel9*	0001 X000	`04 10 rr`	A	Negative
DBEQ B, *rel9*	0001 X001	`04 11 rr`	B	
DBEQ D, *rel9*	0001 X100	`04 14 rr`	D	
DBEQ X, *rel9*	0001 X101	`04 15 rr`	X	
DBEQ Y, *rel9*	0001 X110	`04 16 rr`	Y	
DBEQ SP, *rel9*	0001 X111	`04 17 rr`	SP	

NOTES:
1. Bits 7:6:5 select DBEQ or DBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

DBNE Decrement and Branch if Not Equal to Zero DBNE

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

(counter) – 1 ⇒ counter
If (counter) not = 0, then (PC) + $0003 + rel ⇒ PC

Subtracts one from the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register does not reach zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DBNE *abdxysp*, *rel9*	REL (9-bit)	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding				
Source Form	**Postbyte[1]**	**Object Code**	**Counter Register**	**Offset**
DBNE A, *rel9*	0010 X000	`04 20 rr`	A	Positive
DBNE B, *rel9*	0010 X001	`04 21 rr`	B	
DBNE D, *rel9*	0010 X100	`04 24 rr`	D	
DBNE X, *rel9*	0010 X101	`04 25 rr`	X	
DBNE Y, *rel9*	0010 X110	`04 26 rr`	Y	
DBNE SP, *rel9*	0010 X111	`04 27 rr`	SP	
DBNE A, *rel9*	0011 X000	`04 30 rr`	A	Negative
DBNE B, *rel9*	0011 X001	`04 31 rr`	B	
DBNE D, *rel9*	0011 X100	`04 34 rr`	D	
DBNE X, *rel9*	0011 X101	`04 35 rr`	X	
DBNE Y, *rel9*	0011 X110	`04 36 rr`	Y	
DBNE SP, *rel9*	0011 X111	`04 37 rr`	SP	

NOTES:
1. Bits 7:6:5 select DBEQ or DBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

DEC — Decrement M — DEC

Operation

(M) – $01 ⇒ M

Subtracts one from the value in M. The N, Z, and V bits are set or cleared by the operation. The C bit is not affected by the operation, allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Effects

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 operation produces a two's complement overflow (if and only if (M) was $80 before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DEC *opr16a*	EXT	73 hh ll	rPw0
DEC *oprx0_xysppc*	IDX	63 xb	rPw
DEC *oprx9,xysppc*	IDX1	63 xb ff	rPw0
DEC *oprx16,xysppc*	IDX2	63 xb ee ff	frPwP
DEC [D,*xysppc*]	[D,IDX]	63 xb	fIfrPw
DEC [*oprx16,xysppc*]	[IDX2]	63 xb ee ff	fIPrPw

DECA — Decrement A — DECA

Operation (A) – $01 ⇒ A

Subtracts one from the value in A. The N, Z, and V bits are set or cleared by the operation. The C bit is not affected by the operation, allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Effects

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 operation produces a two's complement overflow (if and only if (A) was $80 before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DECA	INH	43	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

DECB Decrement B DECB

Operation

(B) – $01 ⇒ B

Subtracts one from the value in B. The N, Z, and V bits are set or cleared by the operation. The C bit is not affected by the operation, allowing the DEC instruction to be used as a loop counter in multiple-precision computations.

CCR Effects

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 operation produces a two's complement overflow (if and only if (B) was $80 before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DECB	INH	53	0

DES

Decrement SP
(same as LEAS –1,SP)

DES

Operation (SP) – $0001 ⇒ SP

Subtracts one from SP. DES assembles as LEAS –1,SP. DES does not affect condition code bits as DEX and DEY do.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DES	IDX	1B 9F	Pf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

DEX — Decrement X — DEX

Operation (X) – $0001 ⇒ X

Subtracts one from X. The Z bit reflects the result. The LEAX –1,X instruction does the same thing as DEX, but without affecting the Z bit.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DEX	INH	09	0

DEY

Decrement Y

DEY

Operation (Y) – $0001 ⇒ Y

Subtracts one from Y. The Z bit reflects the result. The LEAY –1,Y instruction does the same thing as DEY, but without affecting the Z bit.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
DEY	INH	03	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EDIV Extended Divide, Unsigned EDIV

Operation

(Y):(D) ÷ (X) ⇒ Y; remainder ⇒ D

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. Division by zero has no effect, except that the states of the N, Z, and V bits are undefined.

CCR Effects

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 0
Z: Set if result is $0000; cleared otherwise; undefined after overflow or division by 0
V: Set if the result is greater than $FFFF; cleared otherwise; undefined after division by 0
C: Set if divisor is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EDIV	INH	11	fffffffff0

EDIVS Extended Divide, Signed EDIVS

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

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. Division by zero has no effect, except that the C bit is set and the states of the N, Z, and V bits are undefined.

CCR Effects

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 0
Z: Set if result is $0000; cleared otherwise; undefined after overflow or division by 0
V: Set if the result is greater than $7FFF or less than $8000; cleared otherwise; undefined after division by 0
C: Set if divisor is $0000; cleared otherwise; indicates division by 0

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EDIVS	INH	18 14	Offffffffff0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EMACS Extended Multiply and Accumulate, Signed EMACS

Operation

$(M_X):(M_{X+1}) \times (M_Y):(M_{Y+1}) + (M):(M+1):(M+2):(M+3) \Rightarrow M+1:M+2:M+3$

Multiplies two 16-bit values. Adds the 32-bit product to the value in a 32-bit accumulator in memory. EMACS is a signed integer operation. All operands and results are located in memory. X must point to the high byte of the first source operand, and Y must point to the high byte of the second source operand. An extended address supplied with the instruction must point to the most significant byte of the 32-bit result.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result, R31, is set; cleared otherwise

Z: Set if result is $00000000; cleared otherwise

V: M31 • I31 • $\overline{R31}$ | $\overline{M31}$ • $\overline{I31}$ • R31; set if result is greater than $7FFFFFFF (+ overflow) or less than $80000000 (– underflow); indicates two's complement overflow

C: M15 • I15 | I15 • $\overline{R15}$ | $\overline{R15}$ • M15; set if there is a carry from bit 15 of the result, R15; cleared otherwise; indicates a carry from low word to high word of the result

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMACS *opr16a*[1]	Special	18 12 hh ll	ORROfffRRfWWP

NOTES:

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

EMAXD Extended Maximum in D EMAXD

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

MAX [(D), (M):(M + 1)] ⇒ D

Subtracts an unsigned 16-bit value in M:M + 1 from an unsigned 16-bit value in D to determine which is larger. Puts the larger value in D. If the values are equal, the Z bit is set. If the value in M:M + 1 is larger, the C bit is set when the value in M:M + 1 replaces the value in D. If the value in D is larger, the C bit is cleared.

EMAXD accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate finding the largest value in a list of values.

CCR Effects

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 • $\overline{M15}$ • $\overline{R15}$ | $\overline{D15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if (M):(M + 1) is larger than (D); cleared otherwise

Condition code bits reflect internal subtraction: R = (D) – (M):(M + 1).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMAXD *oprx0_xysppc*	IDX	18 1A xb	ORPf
EMAXD *oprx9,xysppc*	IDX1	18 1A xb ff	ORPO
EMAXD *oprx16,xysppc*	IDX2	18 1A xb ee ff	OfRPP
EMAXD [D,*xysppc*]	[D,IDX]	18 1A xb	OfIfRPf
EMAXD [*oprx16,xysppc*]	[IDX2]	18 1A xb ee ff	OfIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EMAXM Extended Maximum in M EMAXM

Operation

MAX [(D), (M):(M + 1)] ⇒ M:M + 1

Subtracts an unsigned 16-bit value in M:M + 1 from an unsigned 16-bit value in D to determine which is larger. Puts the larger value in M:M + 1. If the values are equal, the Z bit is set. If the value in M:M + 1 is larger, the C bit is set. If the value in D is larger, the C bit is cleared when the value in D replaces the value in M:M + 1.

EMAXM accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate controlling the values in a list of values.

CCR Effects

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 • $\overline{M15}$ • $\overline{R15}$ | $\overline{D15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if (M):(M + 1) is larger than (D); cleared otherwise

Condition code bits reflect internal subtraction: R = (D) – (M):(M + 1).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMAXM *oprx0_xysppc*	IDX	`18 1E xb`	`ORPW`
EMAXM *oprx9*,*xysppc*	IDX1	`18 1E xb ff`	`ORPWO`
EMAXM *oprx16*,*xysppc*	IDX2	`18 1E xb ee ff`	`OfRPWP`
EMAXM [D,*xysppc*]	[D,IDX]	`18 1E xb`	`OfIfRPW`
EMAXM [*oprx16*,*xysppc*]	[IDX2]	`18 1E xb ee ff`	`OfIPRPW`

EMIND Extended Minimum in D EMIND

Operation

MIN [(D), (M):(M + 1)] ⇒ D

Subtracts an unsigned 16-bit value in M:M + 1 from an unsigned 16-bit value in D to determine which is larger. Puts the smaller value in D. If the values are equal, the Z bit is set. If the value in M:M + 1 is larger, the C bit is set. If the value in D is larger, the C bit is cleared when the value in M:M + 1 replaces the value in D.

EMIND accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate finding the smallest value in a list of values.

CCR Effects

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 • $\overline{M15}$ • $\overline{R15}$ | $\overline{D15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if (M):(M + 1) is larger than (D); cleared otherwise

Condition code bits reflect internal subtraction: R = (D) – (M):(M + 1).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMIND *oprx0_xysppc*	IDX	18 1B xb	ORPf
EMIND *oprx9*,*xysppc*	IDX1	18 1B xb ff	ORPO
EMIND *oprx16*,*xysppc*	IDX2	18 1B xb ee ff	OfRPP
EMIND [D,*xysppc*]	[D,IDX]	18 1B xb	OfIfRPf
EMIND [*oprx16*,*xysppc*]	[IDX2]	18 1B xb ee ff	OfIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EMINM Extended Minimum in M EMINM

Operation

MIN [(D), (M):(M + 1)] ⇒ M:M + 1

Subtracts an unsigned 16-bit value in M:M + 1 from an unsigned 16-bit value in D to determine which is larger. Puts the smaller value in M:M + 1. If the values are equal, the Z bit is set. If the value in M:M + 1 is larger, the C bit is set when the value in D replaces the value in M:M + 1. If the value in D is larger, the C bit is cleared.

EMINM accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate finding the smallest value in a list of values.

CCR Effects

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 • $\overline{M15}$ • $\overline{R15}$ | $\overline{D15}$ • M15 • R15; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if (M):(M + 1) is larger than (D); cleared otherwise

Condition code bits reflect internal subtraction: R = (D) – (M):(M + 1).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMINM *oprx0_xysppc*	IDX	`18 1F xb`	`ORPW`
EMINM *oprx9,xysppc*	IDX1	`18 1F xb ff`	`ORPWO`
EMINM *oprx16,xysppc*	IDX2	`18 1F xb ee ff`	`OfRPWP`
EMINM [D,*xysppc*]	[D,IDX]	`18 1F xb`	`OfIfRPW`
EMINM [*oprx16,xysppc*]	[IDX2]	`18 1F xb ee ff`	`OfIPRPW`

EMUL Extended Multiply, Unsigned EMUL

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

Multiplies an unsigned 16-bit value in D by an unsigned 16-bit value in Y. Puts the high 16-bits of the unsigned 32-bit result in Y and the low 16-bits of the result in D.

The C bit can be used to round the low 16 bits of the result.

CCR Effects

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMUL	INH	13	ff0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EMULS Extended Multiply, Signed EMULS

Operation

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

Multiplies a signed 16-bit value in D by a signed 16-bit value in Y. Puts the high 16 bits of the 32-bit signed result in Y and the low 16 bits of the result in D.

The C bit can be used to round the low 16 bits of the result.

CCR Effects

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EMULS	INH	18 13	`0f0` `0ff0`[1]

NOTES:
1. EMULS has an extra free cycle if it is followed by another page two instruction.

EORA Exclusive OR A EORA

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

(A) ⊕ (M) ⇒ A
or
(A) ⊕ imm ⇒ A

Performs a logical exclusive OR of the value in A and either the value in M or an immediate value. Puts the result in A.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EORA #*opr8i*	IMM	88 ii	P
EORA *opr8a*	DIR	98 dd	rPf
EORA *opr16a*	EXT	B8 hh ll	rPO
EORA *oprx0_xysppc*	IDX	A8 xb	rPf
EORA *oprx9,xysppc*	IDX1	A8 xb ff	rPO
EORA *oprx16,xysppc*	IDX2	A8 xb ee ff	frPP
EORA [D,*xysppc*]	[D,IDX]	A8 xb	fIfrPf
EORA [*oprx16*,*xysppc*]	[IDX2]	A8 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EORB Exclusive OR B with M EORB

Operation

(B) ⊕ (M) ⇒ B
or
(B) ⊕ imm ⇒ B

Performs a logical exclusive OR of the value in B and either the value in M or an immediate value. Puts the result in B.

CCR Effects

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: Cleared

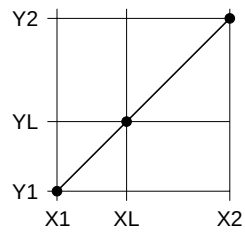
Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EORB #*opr8i*	IMM	C8 ii	P
EORB *opr8a*	DIR	D8 dd	rPf
EORB *opr16a*	EXT	F8 hh ll	rPO
EORB *oprx0_xysppc*	IDX	E8 xb	rPf
EORB *oprx9,xysppc*	IDX1	E8 xb ff	rPO
EORB *oprx16,xysppc*	IDX2	E8 xb ee ff	frPP
EORB [D,*xysppc*]	[D,IDX]	E8 xb	fIfrPf
EORB [*oprx16*,*xysppc*]	[IDX2]	E8 xb ee ff	fIPrPf

ETBL Extended Table Lookup and Interpolate ETBL

Operation (M):(M + 1) + [(B) × ((M + 2):(M + 3) – (M):(M + 1))] ⇒ D

Linearly interpolates and stores in D one of 256 values between a pair of data entries, Y1 and Y2, in a lookup table. Data entries represent y coordinates of line segment endpoints. Table entries and the interpolated results are 16-bit values.



Before executing ETBL, point an indexing register at the Y1 value closest to but less than or equal to the Y value to interpolate. Point to Y1 using any indexed addressing mode except indirect, 9-bit offset, and 16-bit offset. The next table entry after Y1 is Y2. Load B with a binary fraction (radix point to the left of the MSB) representing the ratio:

(XL – X1) ÷ (X2 – X1)
where
X1 = Y1 and X2 = Y2
XL is the x coordinate of the value to interpolate

The 16-bit unrounded result, YL, is calculated using the expression:
YL = Y1 + [(B) × (Y2 – Y1)]
where
Y1 = 16-bit data entry pointed to by effective address
Y2 = 16-bit data entry pointed to by the effective address plus two

The 24-bit intermediate value (B) × (Y2 – Y1) has a radix point between bits 7 and 8.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is $0000; cleared otherwise
C: Set if result can be rounded up; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ETBL *oprx0_xysppc*	IDX	18 3F xb	ORRfffffP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

EXG — Exchange Register Contents — EXG

Operation

(r1) ⇔ (r2) when r1 and r2 are the same size
$00:(r1) ⇒ (r2) when r1 is 8 bits and r2 is 16 bits
($r1_L$) ⇔ (r2) when r1 is 16 bits and r2 is 8 bits

See the table on the next page.

Exchanges the values between a source register A, B, CCR, D, X, Y, or SP and a destination register A, B, CCR, D, X, Y, or SP. Exchanges involving TMP2 and TMP3 are reserved for Motorola use.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

or

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

CCR bits affected only when the CCR is the destination register. The X 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 X can only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
EXG *abcdxysp*,*abcdxysp*	INH	B7 eb	P

EXG

Exchange Register Contents
(continued)

EXG

Exchange Postbyte (eb) Coding							
Source Form	**Postbyte**	**Object Code**	**Exchange**	**Source Form**	**Postbyte**	**Object Code**	**Exchange**
EXG A,A	1000 X000	B7 80	A⇔A	EXG B,A	1100 X000	B7 C0	B⇒A, A⇒B
EXG A,B	1000 X001	B7 81	A⇔B	EXG B,B	1100 X001	B7 C1	B⇒B, $FF⇒A
EXG A,CCR	1000 X010	B7 82	A⇔CCR	EXG B,CCR	1100 X010	B7 C2	B⇒CCR, $FF:CCR⇒D
EXG A,TMP2	1000 X011	B7 83	$00:A⇒TMP2, TMP2$_L$⇒A	EXG D,TMP2	1100 X011	B7 C3	D⇔TMP2
EXG A,D	1000 X100	B7 84	$00:A⇒D	EXG D,D	1100 X100	B7 C4	D⇔D
EXG A,X	1000 X101	B7 85	$00:A⇒X, X_L⇒A	EXG D,X	1100 X101	B7 C5	D⇔X
EXG A,Y	1000 X110	B7 86	$00:A⇒Y, Y_L⇒A	EXG D,Y	1100 X110	B7 C6	D⇔Y
EXG A,SP	1000 X111	B7 87	$00:A⇒SP, SP_L⇒A	EXG D,SP	1100 X111	B7 C7	D⇔SP
EXG B,A	1001 X000	B7 90	B⇔A	EXG X,A	1101 X000	B7 D0	X_L⇒A, $00:A⇒X
EXG B,B	1001 X001	B7 91	B⇔B	EXG X,B	1101 X001	B7 D1	X_L⇒B, $FF:B⇒X
EXG B,CCR	1001 X010	B7 92	B⇔CCR	EXG X,CCR	1101 X010	B7 D2	X_L⇒CCR, $FF:CCR⇒X
EXG B,TMP2	1001 X011	B7 93	$00:B⇒TMP2, TMP2$_L$⇒B	EXG X,TMP2	1101 X011	B7 D3	X⇔TMP2
EXG B,D	1001 X100	B7 94	$00:B⇒D	EXG X,D	1101 X100	B7 D4	X⇔D
EXG B,X	1001 X101	B7 95	$00:B⇒X, X_L⇒B	EXG X,X	1101 X101	B7 D5	X⇔X
EXG B,Y	1001 X110	B7 96	$00:B⇒Y, Y_L⇒B	EXG X,Y	1101 X110	B7 D6	X⇔Y
EXG B,SP	1001 X111	B7 97	$00:B⇒SP, SP_L⇒B	EXG X,SP	1101 X111	B7 D7	X⇔SP
EXG CCR,A	1010 X000	B7 A0	CCR⇔A	EXG Y,A	1110 X000	B7 E0	Y_L⇒A, $00:A⇒Y
EXG CCR,B	1010 X001	B7 A1	CCR⇔B	EXG Y,B	1110 X001	B7 E1	Y_L⇒B, $FF:B⇒Y
EXG CCR,CCR	1010 X010	B7 A2	CCR⇔CCR	EXG Y,CCR	1110 X010	B7 E2	Y_L⇒CCR, $FF:CCR⇒Y
EXG CCR,TMP2	1010 X011	B7 A3	$00:CCR⇒TMP2, TMP2$_L$⇒CCR	EXG Y,TMP2	1110 X011	B7 E3	Y⇔TMP2
EXG CCR,D	1010 X100	B7 A4	$00:CCR⇒D	EXG Y,D	1110 X100	B7 E4	Y⇔D
EXG CCR,X	1010 X101	B7 A5	$00:CCR⇒X, X_L⇒CCR	EXG Y,X	1110 X101	B7 E5	Y⇔X
EXG CCR,Y	1010 X110	B7 A6	$00:CCR⇒Y, Y_L⇒CCR	EXG Y,Y	1110 X110	B7 E6	Y⇔Y
EXG CCR,SP	1010 X111	B7 A7	$00:CCR⇒SP, SP_L⇒CCR	EXG Y,SP	1110 X111	B7 E7	Y⇔SP
EXG TMP3,A	1011 X000	B7 B0	TMP3$_L$⇒A, $00:A⇒TMP3	EXG SP,A	1111 X000	B7 F0	SP_L⇒A, $00:A⇒SP
EXG TMP3,B	1011 X001	B7 B1	TMP3$_L$⇒B, $FF:B⇒TMP3	EXG SP,B	1111 X001	B7 F1	SP_L⇒B, $FF:B⇒SP
EXG TMP3,CCR	1011 X010	B7 B2	TMP3$_L$⇒CCR, $FF:CCR⇒TMP3	EXG SP,CCR	1111 X010	B7 F2	SP_L⇒CCR, $FF:CCR⇒SP
EXG TMP3,TMP2	1011 X011	B7 B3	TMP3⇔TMP2	EXG SP,TMP2	1111 X011	B7 F3	SP⇔TMP2
EXG TMP3,D	1011 X100	B7 B4	TMP3⇔D	EXG SP,D	1111 X100	B7 F4	SP⇔D
EXG TMP3,X	1011 X101	B7 B5	TMP3⇔X	EXG SP,X	1111 X101	B7 F5	SP⇔X
EXG TMP3,Y	1011 X110	B7 B6	TMP3⇔Y	EXG SP,Y	1111 X110	B7 F6	SP⇔Y
EXG TMP3,SP	1011 X111	B7 B7	TMP3⇔SP	EXG SP,SP	1111 X111	B7 F7	SP⇔SP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

FDIV — Fractional Divide — FDIV

Operation

(D) ÷ (X) ⇒ X, remainder ⇒ D

Divides an unsigned 16-bit numerator in D by an unsigned 16-bit denominator in X. Puts the unsigned 16-bit quotient in X and the 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 0, 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 x 16-bit integer division. The result is interpreted as a binary-weighted fraction, which resulted from the division of a 16-bit integer by a larger 16-bit integer. A result of $0001 corresponds to 0.000015, and $FFFF corresponds to 0.9998. The remainder of an IDIV instruction can be resolved into a binary-weighted fraction by an FDIV instruction. The remainder of an FDIV instruction can be resolved into the next 16 bits of binary-weighted fraction by another FDIV instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	Δ	Δ

Z: Set if quotient is $0000; cleared otherwise

V: Set if the denominator X is less than or equal to the numerator D; cleared otherwise

C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet \ldots \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$; set if denominator is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
FDIV	INH	18 11	Offfffffff0

IBEQ Increment and Branch if Equal to Zero IBEQ

Operation

(counter) + 1 ⇒ counter
If (counter) = 0, then (PC) + $0003 + rel ⇒ PC

Adds one to the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register reaches zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
IBEQ *abdxysp*, *rel9*	REL	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding

Source Form	Postbyte[1]	Object Code	Counter Register	Offset
IBEQ A, *rel9*	1000 X000	`04 80 rr`	A	Positive
IBEQ B, *rel9*	1000 X001	`04 81 rr`	B	
IBEQ D, *rel9*	1000 X100	`04 84 rr`	D	
IBEQ X, *rel9*	1000 X101	`04 85 rr`	X	
IBEQ Y, *rel9*	1000 X110	`04 86 rr`	Y	
IBEQ SP, *rel9*	1000 X111	`04 87 rr`	SP	
IBEQ A, *rel9*	1001 X000	`04 90 rr`	A	Negative
IBEQ B, *rel9*	1001 X001	`04 91 rr`	B	
IBEQ D, *rel9*	1001 X100	`04 94 rr`	D	
IBEQ X, *rel9*	1001 X101	`04 95 rr`	X	
IBEQ Y, *rel9*	1001 X110	`04 96 rr`	Y	
IBEQ SP, *rel9*	1001 X111	`04 97 rr`	SP	

NOTES:
1. Bits 7:6:5 select IBEQ or IBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

IBNE Increment and Branch if Not Equal to Zero IBNE

Operation

(counter) + 1 ⇒ counter
If (counter) ≠ 0, then (PC) + $0003 + rel ⇒ PC

Adds one to the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register does not reach zero.Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
IBNE *abdxysp*, *rel9*	REL	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding				
Source Form	**Postbyte[1]**	**Object Code**	**Counter Register**	**Offset**
IBNE A, *rel9*	1010 X000	`04 A0 rr`	A	Positive
IBNE B, *rel9*	1010 X001	`04 A1 rr`	B	
IBNE D, *rel9*	1010 X100	`04 A4 rr`	D	
IBNE X, *rel9*	1010 X101	`04 A5 rr`	X	
IBNE Y, *rel9*	1010 X110	`04 A6 rr`	Y	
IBNE SP, *rel9*	1010 X111	`04 A7 rr`	SP	
IBNE A, *rel9*	1011 X000	`04 B0 rr`	A	Negative
IBNE B, *rel9*	1011 X001	`04 B1 rr`	B	
IBNE D, *rel9*	1011 X100	`04 B4 rr`	D	
IBNE X, *rel9*	1011 X101	`04 B5 rr`	X	
IBNE Y, *rel9*	1011 X110	`04 B6 rr`	Y	
IBNE SP, *rel9*	1011 X111	`04 B7 rr`	SP	

NOTES:
1. Bits 7:6:5 select IBEQ or IBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

IDIV — Integer Divide, Unsigned — IDIV

Operation (D) ÷ (X) ⇒ X; remainder ⇒ D

Divides an unsigned 16-bit dividend in D by an unsigned 16-bit divisor in X. Puts the unsigned 16-bit quotient in X and the 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 0, the quotient is set to $FFFF, and the remainder is indeterminate.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	0	Δ

Z: Set if quotient is $0000; cleared otherwise
V: Cleared
C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12}$ ... $\bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$; set if denominator is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
IDIV	INH	18 10	Offfffffff0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

IDIVS Integer Divide, Signed IDIVS

Operation

(D) ÷ (X) ⇒ X; remainder ⇒ D

Divides a signed 16-bit dividend in D by a signed 16-bit divisor in X. Puts the signed 16-bit quotient in X and the signed 16-bit remainder in D. If division by 0 is attempted, the values in D and X do not change, but the N, Z, and V bits are undefined.

Other than division by 0, which is not legal and sets the C bit, the only overflow case is:

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

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

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of quotient is set; cleared otherwise; undefined after overflow or division by 0

Z: Set if quotient is $0000; cleared otherwise; undefined after overflow or division by 0

V: Set if the quotient is greater than $7FFF or less than $8000; cleared otherwise; undefined after division by 0

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
IDIVS	INH	18 15	Offfffffff0

INC — Increment M — INC

Operation

(M) + $01 ⇒ M

Adds one to the value in M. The N, Z, and V bits reflect the result of the operation. The C bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Effects

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 the operation produces a two's complement overflow (if and only if (M) was $7F before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INC *opr16a*	EXT	72 hh ll	rPw0
INC *oprx0_xysppc*	IDX	62 xb	rPw
INC *oprx9*,*xysppc*	IDX1	62 xb ff	rPw0
INC *oprx16*,*xysppc*	IDX2	62 xb ee ff	frPwP
INC [D,*xysppc*]	[D,IDX]	62 xb	fIfrPw
INC [*oprx16*,*xysppc*]	[IDX2]	62 xb ee ff	fIPrPw

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

INCA — Increment A — INCA

Operation

(A) + $01 ⇒ A

Adds one to the value in A. The N, Z and V bits reflect the result of the operation. The C bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Effects

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 the operation produces a two's complement overflow (if and only if (A) was $7F before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INCA	INH	42	0

INCB Increment B INCB

Operation

(B) + $01 ⇒ B

Adds one to the value in B. The N, Z and V bits reflect the result of the operation. The C bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Effects

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 the operation produces a two's complement overflow (if and only if (B) was $7F before the operation); cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INCB	INH	52	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

INS

Increment SP
(same as LEAS 1,SP)

INS

Operation (SP) + $0001 ⇒ SP

Adds one to SP. INS assembles as LEAS 1,SP. INS does not affect condition code bits as INX and INY instructions do.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INS	IDX	1B 81	Pf

INX

Increment X

INX

Operation (X) + $0001 ⇒ X

Adds one to X. LEAX 1,X can produce the same result but LEAX does not affect the Z bit. Although the LEAX instruction is more flexible, INX requires only one byte of object code.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INX	INH	08	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

INY — Increment Y — INY

Operation (Y) + $0001 ⇒ Y

Adds one to Y. LEAY 1,Y can produce the same result but LEAY does not affect the Z bit. Although the LEAY instruction is more flexible, INY requires only one byte of object code.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is $0000; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
INY	INH	02	0

JMP

Jump

JMP

Operation

Subroutine address ⇒ PC

Jumps to the instruction stored at the effective address. The effective address is obtained according to the rules for extended or indexed addressing.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
JMP *opr16a*	EXT	06 hh ll	PPP
JMP *oprx0_xysppc*	IDX	05 xb	PPP
JMP *oprx9*,*xysppc*	IDX1	05 xb ff	PPP
JMP *oprx16*,*xysppc*	IDX2	05 xb ee ff	fPPP
JMP [D,*xysppc*]	[D,IDX]	05 xb	fIfPPP
JMP [*oprx16*,*xysppc*]	[IDX2]	05 xb ee ff	fIfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

JSR — Jump to Subroutine — JSR

Operation

(SP) – $0002 ⇒ SP
RTN_H:RTN_L ⇒ (M_{SP}):($M_{SP + 1}$)

Subroutine address ⇒ PC

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

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

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Jumps to the location determined by the effective address.

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

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
JSR *opr8a*	DIR	17 dd	SPPP
JSR *opr16a*	EXT	16 hh ll	SPPP
JSR *oprx0_xysppc*	IDX	15 xb	PPPS
JSR *oprx9*,*xysppc*	IDX1	15 xb ff	PPPS
JSR *oprx16*,*xysppc*	IDX2	15 xb ee ff	fPPPS
JSR [D,*xysppc*]	[D,IDX]	15 xb	fIfPPPS
JSR [*oprx16*,*xysppc*]	[IDX2]	15 xb ee ff	fIfPPPS

LBCC

Long Branch if C Clear
(same as LBHS)

LBCC

Operation If C = 0, then (PC) + $0004 + rel ⇒ PC

Tests the C bit and branches if C = 0.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBCC *rel16*	REL	18 24 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBCC/LBHS	18 24	(R) ≥ (M) or (B) ≥ (A)	LBCS/LBLO	18 25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	
LBGE	2C	(R) ≥ (M) or (B) ≥ (A)	LBLT	18 2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBCS

Long Branch if C Set
(same as LBLO)

LBCS

Operation If C = 1, then (PC) + $0004 + rel ⇒ PC

Tests the C bit and branches if C = 1.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBCS *rel16*	REL	18 25 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBCS/LBLO	18 25	(R) < (M) or (B) < (A)	LBCC/LBHS	18 24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	
LBLT	18 2D	(R) < (M) or (B) < (A)	LBGE	18 2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	

LBEQ Long Branch if Equal LBEQ

Operation

If Z = 1, (PC) + $0004 + rel ⇒ PC

Tests the Z bit and branches if Z = 1.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBEQ *rel16*	REL	18 27 qq rr	OPPP (no branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBEQ	18 27	(R) = (M) or (R) = zero	LBNE	18 26	(R) ≠ (M) or (R) ≠ zero	Signed, unsigned or simple
		Z = 1			Z = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBGE — Long Branch if Greater Than or Equal to Zero — LBGE

Operation

If N ⊕ V = 0, (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBGE *rel16*	REL	18 2C qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBGE	18 2C	(R) ≥ (M) or (B) ≥ (A)	LBLT	18 2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	
LBHS/LBCC	18 24	(R) ≥ (M) or (B) ≥ (A)	LBLO/LBCS	18 25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	

LBGT Long Branch if Greater Than Zero LBGT

Operation If Z | (N ⊕ V) = 0, then (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBGT *rel16*	REL	18 2E qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBGT	18 2E	(R) > (M) or (B) > (A)	LBLE	18 2F	(R) ≤ (M) or (B) ≤ (A)	Signed
		Z | (N ⊕ V) = 0			Z | (N ⊕ V) = 1	
LBHI	18 22	(R) > (M) or (B) > (A)	LBLS	18 23	(R) ≤ (M) or (B) ≤ (A)	Unsigned
		C | Z = 0			C | Z = 1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBHI Long Branch if Higher LBHI

Operation

If C | Z = 0, then (PC) + $0004 + rel ⇒ PC

LBHI can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than the value in M. After CBA or SBA, the branch occurs if the value in B is greater than the value in A.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBHI *rel16*	REL	18 22 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBHI	18 22	(R) > (M) or (B) > (A)	LBLS	18 23	(R) ≤ (M) or (B) ≤ (A)	Unsigned
		C | Z = 0			C | Z = 1	
LBGT	18 2E	(R) > (M) or (B) > (A)	LBLE	18 2F	(R) ≤ (M) or (B) ≤ (A)	Signed
		Z | (N ⊕ V) = 0			Z | (N ⊕ V) = 1	

LBHS

Long Branch if Higher or Same
(same as LBCC)

LBHS

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

If C = 0, then (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBHS *rel16*	REL	18 24 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBHS/LBCC	18 24	(R) ≥ (M) or (B) ≥ (A)	LBLO/LBCS	18 25	(R) < (M) or (B) < (A)	Unsigned
		C = 0			C = 1	
LBGE	18 2C	(R) ≥ (M) or (B) ≥ (A)	LBLT	18 2D	(R) < (M) or (B) < (A)	Signed
		N ⊕ V = 0			N ⊕ V = 1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBLE Long Branch if Less Than or Equal to Zero LBLE

Operation

If Z | (N ⊕ V) = 1, then (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBLE *rel16*	REL	18 2F qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBLE	18 2F	(R) ≤ (M) or (B) ≤ (A)	LBGT	18 2E	(R) > (M) or (B) > (A)	Signed
		Z | (N ⊕ V) = 1			Z | (N ⊕ V) = 0	
LBLS	18 23	(R) ≤ (M) or (B) ≤ (A)	LBHI	18 22	(R) > (M) or (B) > (A)	Unsigned
		C | Z = 1			C | Z = 0	

LBLO

Long Branch if Lower
(same as LBCS)

LBLO

Operation

If C = 1, then (PC) + $0004 + rel ⇒ PC

LBLO can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBLO *rel16*	REL	18 25 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBLO/LBCS	18 25	(R) < (M) or (B) < (A)	LBHS/LBCC	18 24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	
LBLT	18 2D	(R) < (M) or (B) < (A)	LBGE	18 2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBLS Long Branch if Lower or Same LBLS

Operation

If C | Z = 1, then (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBLS *rel16*	REL	18 23 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBLS	18 23	(R) ≤ (M) or (B) ≤ (A)	LBHI	18 22	(R) > (M) or (B) > (A)	Unsigned
		C | Z = 1			C | Z = 0	
LBLE	18 2F	(R) ≤ (M) or (B) ≤ (A)	LBGT	18 2E	(R) > (M) or (B) > (A)	Signed
		Z | (N ⊕ V) = 1			Z | (N ⊕ V) = 0	

LBLT Long Branch if Less Than Zero LBLT

Operation If N ⊕ V = 1, (PC) + $0004 + rel ⇒ PC

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

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBLT *rel16*	REL	18 2D qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBLT	18 2D	(R) < (M) or (B) < (A)	LBGE	18 2C	(R) ≥ (M) or (B) ≥ (A)	Signed
		N ⊕ V = 1			N ⊕ V = 0	
LBLO/LBCS	18 25	(R) < (M) or (B) < (A)	LBHS/LBCC	18 24	(R) ≥ (M) or (B) ≥ (A)	Unsigned
		C = 1			C = 0	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBMI Long Branch if Minus LBMI

Operation

If N = 1, then (PC) + $0004 + rel ⇒ PC

Tests the N bit and branches if N = 1.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBMI *rel16*	REL	18 2B qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBMI	18 2B	Negative N = 1	LBPL	18 2A	Positive N = 0	Simple

LBNE Long Branch if Not Equal to Zero LBNE

Operation

If Z = 0, then (PC) + $0004 + rel ⇒ PC

Tests the Z bit and branches if Z = 0.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBNE *rel16*	REL	18 26 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBNE	18 26	(R) ≠ (M) or (R) ≠ zero	LBEQ	18 27	(R) = (M) or (R) = zero	Signed, unsigned, or simple
		Z = 0			Z = 1	

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBPL Long Branch if Plus LBPL

Operation

If N = 0, then (PC) + $0004 + rel ⇒ PC

Tests the N bit and branches if N = 0.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBPL *rel16*	REL	18 2A qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBPL	18 2A	Positive N = 0	LBMI	18 2B	Negative N = 1	Simple

LBRA Long Branch Always LBRA

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation (PC) + $0004 + rel ⇒ PC

Branches unconditionally.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBRA *rel16*	REL	18 20 qq rr	OPPP

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBRA	18 20	Always	LBRN	18 21	Never	Simple

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBRN Long Branch Never LBRN

Operation (PC) + $0004 ⇒ PC

Never branches. LBRN is effectively a 4-byte NOP that requires three cycles. LBRN is included in the instruction set to provide a complement to the LBRA instruction. LBRN 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBRN *rel16*	REL	18 21 qq rr	OPO

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBRN	18 21	Never	LBRA	18 20	Always	Simple

LBVC Long Branch if V Clear LBVC

Operation

If V = 0, then (PC) + $0004 + rel ⇒ PC

Tests the V 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.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBVC *rel16*	REL	18 28 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBVC	18 28	No overflow V = 0	LBVS	18 29	Overflow V = 1	Simple

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LBVS Long Branch if V Set LBVS

Operation

If V = 1, then (PC) + $0004 + rel ⇒ PC

Tests the V bit and branches if V = 1. LBVS causes a branch when a previous operation on two's complement 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.

Rel is a 16-bit two's complement offset for branching forward or backward in memory. Branching range is $8000 to $7FFF (–32768 to 32767) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LBVS *rel16*	REL	18 29 qq rr	OPPP (branch) OPO (no branch)

Branch			Complementary Branch			Comment
Mnemonic	Opcode	Test	Mnemonic	Opcode	Test	
LBVS	18 29	Overflow V = 1	LBVC	18 28	No overflow V = 0	Simple

LDAA Load A LDAA

Operation

(M) ⇒ A
or
imm ⇒ A

Loads A with either the value in M or an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDAA #*opr8i*	IMM	86 ii	P
LDAA *opr8a*	DIR	96 dd	rPf
LDAA *opr16a*	EXT	B6 hh ll	rPO
LDAA *oprx0_xysppc*	IDX	A6 xb	rPf
LDAA *oprx9,xysppc*	IDX1	A6 xb ff	rPO
LDAA *oprx16,xysppc*	IDX2	A6 xb ee ff	frPP
LDAA [D,*xysppc*]	[D,IDX]	A6 xb	fIfrPf
LDAA [*oprx16,xysppc*	[IDX2]	A6 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LDAB Load B LDAB

Operation

(M) ⇒ B
or
imm ⇒ B

Loads B with either the value in M or an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDAB #*opr8i*	IMM	C6 ii	P
LDAB *opr8a*	DIR	D6 dd	rPf
LDAB *opr16a*	EXT	F6 hh ll	rPO
LDAB *oprx0_xysppc*	IDX	E6 xb	rPf
LDAB *oprx9,xysppc*	IDX1	E6 xb ff	rPO
LDAB *oprx16,xysppc*	IDX2	E6 xb ee ff	frPP
LDAB [D,*xysppc*]	[D,IDX]	E6 xb	fIfrPf
LDAB [*oprx16,xysppc*]	[IDX2]	E6 xb ee ff	fIPrPf

LDD Load D LDD

Operation

(M):(M + 1) ⇒ A:B
or
imm ⇒ A:B

Loads A with the value in M and loads B with the value in M:M + 1 or loads A:B with an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDD #*opr16i*	IMM	CC jj kk	PO
LDD *opr8a*	DIR	DC dd	RPf
LDD *opr16a*	EXT	FC hh ll	RPO
LDD *oprx0_xysppc*	IDX	EC xb	RPf
LDD *oprx9*,*xysppc*	IDX1	EC xb ff	RPO
LDD *oprx16*,*xysppc*	IDX2	EC xb ee ff	fRPP
LDD [D,*xysppc*]	[D,IDX]	EC xb	fIfRPf
LDD [*oprx16*,*xysppc*]	[IDX2]	EC xb ee ff	fIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LDS — Load SP — LDS

Operation

(M):(M + 1) ⇒ SP
or
imm ⇒ SP

Loads the high byte of SP with the value in M and the low byte with the value in M + 1 or loads SP with an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDS #*opr16i*	IMM	CF jj kk	PO
LDS *opr8a*	DIR	DF dd	RPf
LDS *opr16a*	EXT	FF hh ll	RPO
LDS *oprx0_xysppc*	IDX	EF xb	RPf
LDS *oprx9*,*xysppc*	IDX1	EF xb ff	RPO
LDS *oprx16*,*xysppc*	IDX2	EF xb ee ff	fRPP
LDS [D,*xysppc*]	[D,IDX]	EF xb	fIfRPf
LDS [*oprx16*,*xysppc*]	[IDX2]	EF xb ee ff	fIPRPf

LDX — Load X — LDX

Operation

(M):(M + 1) ⇒ X
or
imm ⇒ X

Loads the high byte of X with value in M and low byte with the value in M + 1 or loads X with an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDX #*opr16i*	IMM	CE jj kk	PO
LDX *opr8a*	DIR	DE dd	RPf
LDX *opr16a*	EXT	FE hh ll	RPO
LDX *oprx0_xysppc*	IDX	EE xb	RPf
LDX *oprx9*,*xysppc*	IDX1	EE xb ff	RPO
LDX *oprx16*,*xysppc*	IDX2	EE xb ee ff	fRPP
LDX [D,*xysppc*]	[D,IDX]	EE xb	fIfRPf
LDX [*oprx16*,*xysppc*]	[IDX2]	EE xb ee ff	fIPRPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LDY — Load Y — LDY

Operation

(M):(M + 1) ⇒ Y
or
imm ⇒ Y

Loads the high byte of Y with the value in M and the low byte with the value in M + 1 or loads Y with an immediate value.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LDY *#opr16i*	IMM	CD jj kk	PO
LDY *opr8a*	DIR	DD dd	RPf
LDY *opr16a*	EXT	FD hh ll	RPO
LDY *oprx0_xysppc*	IDX	ED xb	RPf
LDY *oprx9,xysppc*	IDX1	ED xb ff	RPO
LDY *oprx16,xysppc*	IDX2	ED xb ee ff	fRPP
LDY [D,*xysppc*]	[D,IDX]	ED xb	fIfRPf
LDY [*oprx16,xysppc*]	[IDX2]	ED xb ee ff	fIPRPf

LEAS Load Effective Address into SP LEAS

Operation

Effective address ⇒ SP

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.

LEAS does not alter condition code bits. This allows stack modification without disturbing CCR bits changed by recent arithmetic operations.

When SP is the indexing register, a predecrement or preincrement LEAS loads SP with the changed value. A postdecrement or postincrement LEAS does not affect the value in SP.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LEAS *oprx0_xysppc*	IDX	1B xb	Pf
LEAS *oprx9,xysppc*	IDX1	1B xb ff	P0
LEAS *oprx16,xysppc*	IDX2	1B xb ee ff	PP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LEAX — Load Effective Address into X — LEAX

Operation

Effective address ⇒ X

Loads 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.

When X is the indexing register, a predecrement or preincrement LEAX loads X with the changed value. A postdecrement or postincrement LEAX does not affect the value in X.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LEAX *oprx0_xysppc*	IDX	1A xb	Pf
LEAX *oprx9,xysppc*	IDX1	1A xb ff	P0
LEAX *oprx16,xysppc*	IDX2	1A xb ee ff	PP

LEAY Load Effective Address into Y LEAY

Operation

Effective address ⇒ Y

Loads 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.

When Y is the indexing register, a predecrement or preincrement LEAY loads Y with the changed value. A postdecrement or postincrement LEAY does not affect the value in Y.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LEAY *oprx0_xysppc*	IDX	19 xb	Pf
LEAY *oprx9,xysppc*	IDX1	19 xb ff	P0
LEAY *oprx16,xysppc*	IDX2	19 xb ee ff	PP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

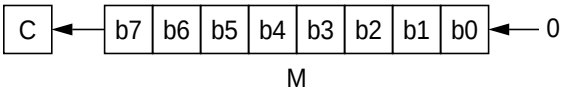
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LSL

Logical Shift Left M
(same as ASL)

LSL

Operation



Shifts all bits of the M one place to the left. Loads bit 0 with 0. Loads the C bit from the most significant bit of M.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

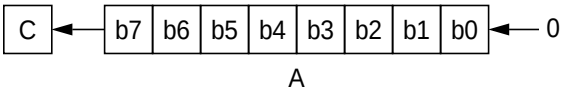
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSL *opr16a*	EXT	78 hh ll	rPw0
LSL *oprx0_xysppc*	IDX	68 xb	rPw
LSL *oprx9*,*xysppc*	IDX1	68 xb ff	rPw0
LSL *oprx16*,*xysppc*	IDX2	68 xb ee ff	frPPw
LSL [D,*xysppc*]	[D,IDX]	68 xb	fIfrPw
LSL [*oprx16*,*xysppc*]	[IDX2]	68 xb ee ff	fIPrPw

LSLA

Logical Shift Left A
(same as ASLA)

LSLA

Operation



Shifts all bits of A one place to the left. Loads bit 0 with 0. Loads the C bit is from the most significant bit of A.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSLA	INH	48	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

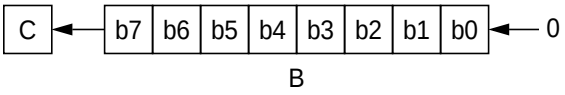
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LSLB

Logical Shift Left B
(same as ASLB)

LSLB

Operation



Shifts all bits of B one place to the left. Loads bit 0 with 0. Loads the C bit from the most significant bit of B.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

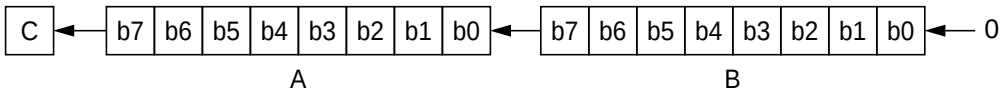
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSLB	INH	58	0

LSLD

Logical Shift Left D
(same as ASLD)

LSLD

Operation



Shifts all bits of D one place to the left. Loads bit 0 with 0. Loads the C bit from the most significant bit of A.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $0000; cleared otherwise

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

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

Code and CPU Cycles

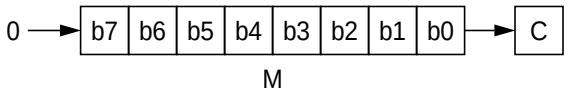
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSLD	INH	59	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LSR — Logical Shift Right M — LSR

Operation



Shifts all bits of M one place to the right. Loads bit 7 with 0. Loads the C bit from the least significant bit of M.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: Cleared

Z: Set if result is $00; cleared otherwise

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

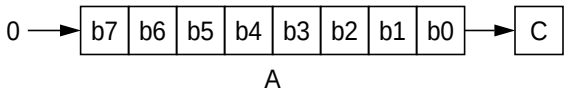
C: M0; set if the LSB of M was set before the shift; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSR *opr16a*	EXT	74 hh ll	rPw0
LSR *oprx0_xysppc*	IDX	64 xb	rPw
LSR *oprx9*,*xysppc*	IDX1	64 xb ff	rPw0
LSR *oprx16*,*xysppc*	IDX2	64 xb ee ff	frPwP
LSR [D,*xysppc*]	[D,IDX]	64 xb	fIfrPw
LSR [*oprx16*,*xysppc*]	[IDX2]	64 xb ee ff	fIPrPw

LSRA — Logical Shift Right A — LSRA

Operation



Shifts all bits of A one place to the right. Loads bit 7 with 0. Loads the C bit from the least significant bit of A.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: Cleared

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

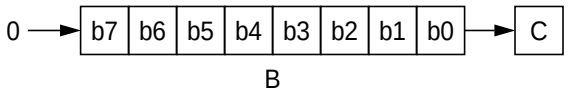
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSRA	INH	44	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

LSRB — Logical Shift Right B — LSRB

Operation



Shifts all bits of B one place to the right. Loads bit 7 with 0. Loads the C bit from the least significant bit of B.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: Cleared

Z: Set if result is $00; cleared otherwise

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

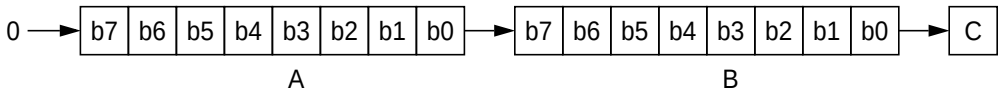
C: B0; set if the LSB of B was set before the shift; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSRB	INH	54	0

LSRD Logical Shift Right D LSRD

Operation



Shifts all bits of D one place to the right. Loads D15 (A7) with 0. Loads the C bit from D0 (B0).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
LSRD	INH	49	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

MAXA — Maximum in A — MAXA

Operation

MAX [(A), (M)] ⇒ A

Subtracts an unsigned 8-bit value in M from an unsigned 8-bit value in A to determine which is larger. Puts the larger value in A. If the values are equal, the Z bit is set. If the value in M is larger, the C bit is set when the value in M replaces the value in A. If the value in A is larger, the C bit is cleared.

MAXA accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate finding the largest value in a list of values.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if (M) is larger than (A); cleared otherwise

Condition code bits reflect internal subtraction: R = (A) – (M).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MAXA *oprx0_xysppc*	IDX	18 18 xb	OrPf
MAXA *oprx9,xysppc*	IDX1	18 18 xb ff	OrPO
MAXA *oprx16,xysppc*	IDX2	18 18 xb ee ff	OfrPP
MAXA [D,*xysppc*]	[D,IDX]	18 18 xb	OfIfrPf
MAXA [*oprx16,xysppc*]	[IDX2]	18 18 xb ee ff	OfIPrPf

MAXM Maximum in M MAXM

Operation

MAX [(A), (M)] ⇒ M

Subtracts an unsigned 8-bit value in M from an unsigned 8-bit value in A to determine which is larger. Puts the larger value in M. If the values are equal, the Z bit is set. If the value in M is larger, the C bit is set. If the value in A is larger, the C bit is cleared when the value in A replaces the value in M.

MAXM accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate controlling the values in a list of values.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if the operation produces a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if (M) is larger than (A); cleared otherwise

Condition code bits reflect internal subtraction: R = (A) – (M).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MAXM *oprx0_xysppc*	IDX	18 1C xb	OrPw
MAXM *oprx9,xysppc*	IDX1	18 1C xb ff	OrPwO
MAXM *oprx16,xysppc*	IDX2	18 1C xb ee ff	OfrPwP
MAXM [D,*xysppc*]	[D,IDX]	18 1C xb	OfIfrPw
MAXM [*oprx16,xysppc*]	[IDX2]	18 1C xb ee ff	OfIPrPw

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

MEM — Determine Grade of Membership (Fuzzy Logic) — MEM

Operation

Grade of membership $\Rightarrow M_Y$
(Y) + $0001 $\Rightarrow$ Y
(X) + $0004 $\Rightarrow$ X

Before executing MEM, initialize A, X and Y. Load A with the current crisp value of a system input variable. Load Y with the fuzzy input RAM location where the grade of membership is to be stored. Load X with the first address of a 4-byte data structure that describes a trapezoidal membership function. The data structure consists of:

- Point_1 — The x-axis starting point for the leading side (at M_X)
- Slope_1 — The slope of the leading side (at M_{X+1})
- Point_2 — The x-axis position of the rightmost point (at M_{X+2})
- Slope_2 — The slope of the trailing side (at M_{X+3})

A slope_1 or slope_2 value of $00 is a special case in which the membership function either starts with a grade of $FF at input = point_1, or ends with a grade of $FF at input = point_2 (infinite slope).

During execution, the value of A remains the same. X is incremented by four and Y is incremented by one.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	?	?

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MEM	Special	01	RRfOw

MINA Minimum in A MINA

Operation

MIN [(A), (M)] ⇒ A

Subtracts an unsigned 8-bit value in M from an unsigned 8-bit value in A to determine which is larger. Puts the smaller value in A. If the values are equal, the Z bit is set. If the value in M is larger, the C bit is set. If the value in A is larger, the C bit is cleared when the value in M replaces the value in A.

MINA accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate finding the smallest value in a list of values.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if the operation produced a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if the value of the value in M is larger than the value in A; cleared otherwise

Condition codes reflect internal subtraction R = (A) – (M).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MINA *oprx0_xysppc*	IDX	18 19 xb	OrPf
MINA *oprx9*,*xysppc*	IDX1	18 19 xb ff	OrPO
MINA *oprx16*,*xysppc*	IDX2	18 19 xb ee ff	OfrPP
MINA [D,*xysppc*]	[D,IDX]	18 19 xb	OfIfrPf
MINA [*oprx16*,*xysppc*]	[IDX2]	18 19 xb ee ff	OfIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

MINM — Minimum in M — MINM

Operation

MIN [(A), (M)] ⇒ M

Subtracts an unsigned 8-bit value in M from an unsigned 8-bit value in A to determine which is larger. Puts the smaller value in M. If the values are equal, the Z bit is set. If the value in M is larger, the C bit is set when the value in A replaces the value in M. If the value in A is larger, the C bit is cleared.

MINM accesses memory with indexed addressing modes for flexibility in specifying operand addresses. Autoincrement and autodecrement functions can facilitate controlling the values in a list of values.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if the operation produced a two's complement overflow; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if the value in M is larger than the value in A; cleared otherwise

Condition codes reflect internal subtraction R = (A) – (M).

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MINM *oprx0_xysppc*	IDX	18 1D xb	OrPw
MINM *oprx9*,*xysppc*	IDX1	18 1D xb ff	OrPwO
MINM *oprx16*,*xysppc*	IDX2	18 1D xb ee ff	OfrPwP
MINM [D,*xysppc*]	[D,IDX]	18 1D xb	OfIfrPw
MINM [*oprx16*,*xysppc*]	[IDX2]	18 1D xb ee ff	OfIPrPw

MOVB Move Byte MOVB

Operation

$(M_1) \Rightarrow M_2$

Moves the value in one 8-bit memory location, M_1, to another 8-bit memory location, M_2. The value in M_1 does not change.

Move instructions can use different addressing modes to access the source and destination of a move. Supported addressing mode combinations are: 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 autoincrement/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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form[1]	Address Mode	Machine Code (Hex)	CPU Cycles
MOVB #*opr8*, *opr16a*	IMM–EXT	`18 0B ii hh ll`	`OPwP`
MOVB #*opr8i*, *oprx0_xysppc*	IMM–IDX	`18 08 xb ii`	`OPwO`
MOVB *opr16a*, *opr16a*	EXT–EXT	`18 0C hh ll hh ll`	`OrPwPO`
MOVB *opr16a*, *oprx0_xysppc*	EXT–IDX	`18 09 xb hh ll`	`OPrPw`
MOVB *oprx0_xysppc*, *opr16a*	IDX–EXT	`18 0D xb hh ll`	`OrPwP`
MOVB *oprx0_xysppc*, *oprx0_xysppc*	IDX–IDX	`18 0A xb xb`	`OrPwO`

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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

MOVW Move Word MOVW

Operation

$(M_1):(M_1 + 1) \Rightarrow M_2:M_2 + 1$

Moves the value in one 16-bit memory location, $M_1:M_1 + 1$, to another 16-bit memory location, $M_2:M_2 + 1$. The value in $M_1:M_1 + 1$ does not change.

Move instructions can use different addressing modes to access the source and destination of a move. These 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 autoincrement/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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form[1]	Address Mode	Machine Code (Hex)	CPU Cycles
MOVW #*opr16i*, *opr16a*	IMM–EXT	18 03 jj kk hh ll	OPWPO
MOVW #*opr16i*, *oprx0_xysppc*	IMM–IDX	18 00 xb jj kk	OPPW
MOVW *opr16a*, *opr16a*	EXT–EXT	18 04 hh ll hh ll	ORPWPO
MOVW *opr16a*, *oprx0_xysppc*	EXT–IDX	18 01 xb hh ll	OPRPW
MOVW *oprx0_xysppc*, *opr16a*	IDX–EXT	18 05 xb hh ll	ORPWP
MOVW *oprx0_xysppc*, *oprx0_xysppc*	IDX–IDX	18 02 xb xb	ORPWO

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

MUL — Multiply, Unsigned — MUL

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

Multiplies the 8-bit unsigned value in A by the 8-bit unsigned value in B and places the 16-bit unsigned result in D. The carry flag allows rounding the high byte of the result through the sequence: MUL, ADCA #0.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	Δ

C: R7; set if bit 7 of the result is set; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
MUL	INH	12	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

NEG — Negate M — NEG

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

Replaces the value in M with its two's complement. A value of $80 does not change.

CCR Effects

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 • $\overline{R6}$ • $\overline{R5}$ • $\overline{R4}$ • $\overline{R3}$ • $\overline{R2}$ • $\overline{R1}$ • $\overline{R0}$; set if there is a two's complement overflow from the implied subtraction from 0; 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 0; cleared otherwise; set in all cases except when (M) = $00

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
NEG *opr16a*	EXT	70 hh ll	rPwO
NEG *oprx0_xysppc*	IDX	60 xb	rPw
NEG *oprx9,xysppc*	IDX1	60 xb ff	rPwO
NEG *oprx16,xysppc*	IDX2	60 xb ee ff	frPwP
NEG [D,*xysppc*]	[D,IDX]	60 xb	fIfrPw
NEG [*oprx16,xysppc*]	[IDX2]	60 xb ee ff	fIPrPw

NEGA Negate A NEGA

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

Replaces the value in A with its two's complement. A value of $80 does not change.

CCR Effects

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 • $\overline{R6}$ • $\overline{R5}$ • $\overline{R4}$ • $\overline{R3}$ • $\overline{R2}$ • $\overline{R1}$ • $\overline{R0}$; set if there is a two's complement overflow from the implied subtraction from 0; 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 0; cleared otherwise; set in all cases except when (A) = $00

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
NEGA	INH	40	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

NEGB Negate B NEGB

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

Replaces the value in B with its two's complement. A value of $80 does not change.

CCR Effects

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 • $\overline{R6}$ • $\overline{R5}$ • $\overline{R4}$ • $\overline{R3}$ • $\overline{R2}$ • $\overline{R1}$ • $\overline{R0}$; set if there is a two's complement overflow from the implied subtraction from 0; 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 0; cleared otherwise; set in all cases except when (B) = $00

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
NEGB	INH	50	0

NOP Null Operation NOP

Operation No operation

This single-byte instruction increments the PC and does nothing else. No other CPU registers are affected. NOP typically is used to produce a time delay, although some software disciplines discourage CPU frequency-based time delays. During debug, NOP instructions are sometimes used to temporarily replace other machine code instructions, thus disabling the replaced instruction(s).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
NOP	INH	A7	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ORAA OR Accumulator A ORAA

Operation

(A) | (M) ⇒ A
or
(A) | imm ⇒ A

Performs logical inclusive OR of the value in A and either the value in M or an immediate value. Puts the result in A.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ORAA #*opr8i*	IMM	8A ii	P
ORAA *opr8a*	DIR	9A dd	rPf
ORAA *opr16a*	EXT	BA hh ll	rPO
ORAA *oprx0_xysppc*	IDX	AA xb	rPf
ORAA *oprx9,xysppc*	IDX1	AA xb ff	rPO
ORAA *oprx16,xysppc*	IDX2	AA xb ee ff	frPP
ORAA [D,*xysppc*]	[D,IDX]	AA xb	fIfrPf
ORAA [*oprx16*,*xysppc*]	[IDX2]	AA xb ee ff	fIPrPf

ORAB OR Accumulator B ORAB

Operation

(B) | (M) ⇒ B
or
(B) | imm ⇒ B

Performs logical inclusive OR of the value in B and either the value in M or an immediate value. Puts the result in B.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Coding (Hex)	CPU Cycles
ORAB #*opr8i*	IMM	CA ii	P
ORAB *opr8a*	DIR	DA dd	rPf
ORAB *opr16a*	EXT	FA hh ll	rPO
ORAB *oprx0_xysppc*	IDX	EA xb	rPf
ORAB *oprx9,xysppc*	IDX1	EA xb ff	rPO
ORAB *oprx16,xysppc*	IDX2	EA xb ee ff	frPP
ORAB [D,*xysppc*]	[D,IDX]	EA xb	fIfrPf
ORAB [*oprx16*,*xysppc*]	[IDX2]	EA xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ORCC OR CCR ORCC

Operation (CCR) | imm ⇒ CCR

Performs a logical inclusive OR of the value in the CCR and an immediate value. Puts the result in the CCR. CCR bits that correspond to 1s in M are set. No other CCR bits change.

NOTE: *The X bit cannot be set by any software instruction.*

CCR Effects

S	X	H	I	N	Z	V	C
⇑	–	⇑	⇑	⇑	⇑	⇑	⇑

A condition code bit is set if the corresponding bit was 1 before the operation or if the corresponding bit in the instruction-provided mask is 1. The X bit cannot be set by any software instruction.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ORCC #*opr8i*	IMM	14 ii	P

PSHA Push A onto Stack PSHA

Operation

(SP) – $0001 ⇒ SP
(A) ⇒ M_{SP}

Decrements SP by one and loads the value in A into the address to which SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHA	INH	36	0s

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PSHB Push B onto Stack PSHB

Operation

(SP) – $0001 ⇒ SP
(B) ⇒ M_{SP}

Decrements SP by one and loads the value in B into the address to which SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHB	INH	37	0s

PSHC Push CCR onto Stack PSHC

Operation

(SP) – $0001 ⇒ SP
(CCR) ⇒ M_{SP}

Decrements SP by one and loads the value in CCR into the address to which the SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHC	INH	39	0S

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PSHD Push D onto Stack PSHD

Operation

$(SP) - \$0002 \Rightarrow SP$
$(A):(B) \Rightarrow M_{SP}:M_{SP+1}$

Decrements SP by two and loads the value in A into the address to which SP points. Loads the value in B into the address to which SP points plus one. After PSHD executes, SP points to the stacked value of A.

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 restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHD	INH	3B	0S

PSHX Push X onto Stack PSHX

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

$(SP) - \$0002 \Rightarrow SP$
$(X_H):(X_L) \Rightarrow M_{SP}:M_{SP+1}$

Decrements SP by two and loads the high byte of X into the address to which SP points. Loads the low byte of X into the address to which SP points plus one. After PSHX executes, SP points to the stacked value of the high byte 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 restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHX	INH	34	0S

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PSHY — Push Y onto Stack — PSHY

Operation

$(SP) - \$0002 \Rightarrow SP$
$(Y_H):(Y_L) \Rightarrow M_{SP}:M_{SP+1}$

Decrements SP by two and loads the high byte of Y into the address to which SP points. Loads the low byte of Y into the address to which SP points plus one. After PSHY executes, SP points to the stacked value of the high byte 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 restore the saved CPU registers just before returning from the subroutine.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PSHY	INH	35	0S

PULA

Pull A from Stack

PULA

Operation

$(M_{SP}) \Rightarrow A$
$(SP) + \$0001 \Rightarrow SP$

Loads A from the address to which SP points. Then increments SP by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULA	INH	32	uf0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PULB — Pull B from Stack — PULB

Operation

$(M_{SP}) \Rightarrow B$
$(SP) + \$0001 \Rightarrow SP$

Loads B from the address to which SP points. Then increments SP by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULB	INH	33	ufO

PULC

Pull CCR from Stack

PULC

Operation

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

Loads CCR from the address to which SP points. Then increments SP by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

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 only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULC	INH	38	uf0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PULD Pull D from Stack PULD

Operation

$(M_{SP}):(M_{SP+1}) \Rightarrow A:B$
$(SP) + \$0002 \Rightarrow SP$

Loads the high byte of D from the address to which SP points. Loads the low byte of D from the address to which SP points plus one. Then increments SP by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULD	INH	3A	Uf0

PULX Pull X from Stack PULX

Operation

$(M_{SP}):(M_{SP + 1}) \Rightarrow X_H:X_L$
$(SP) + \$0002 \Rightarrow SP$

Loads the high byte of X from the address to which SP points. Loads the low byte of X from the address to which SP points plus one. Then increments SP by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULX	INH	30	Uf0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PULY — Pull Y from Stack — PULY

Operation

$(M_{SP}):(M_{SP+1}) \Rightarrow Y_H:Y_L$
$(SP) + \$0002 \Rightarrow SP$

Loads the high byte of Y from the address to which SP points. Loads the low byte of Y from the address to which SP points plus one. Then increments SP by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
PULY	INH	31	Uf0

REV Fuzzy Logic Rule Evaluation REV

Operation MIN – MAX rule evaluation

Performs an unweighted evaluation of a list of rules, using fuzzy inputs to produce fuzzy outputs. REV can be interrupted, so it does not adversely affect interrupt latency.

REV uses an 8-bit unsigned offset from a base address stored in Y to determine the address of each fuzzy input and fuzzy output. 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 begins with the address pointed to by the first rule antecedent and evaluates successive fuzzy input values until it finds 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, REV compares the truth value to successive fuzzy output values until it finds another $FE separator. If the truth value is greater than the current output value, REV writes it to the output. Operation is similar to that of a MAXM instruction. Rule processing continues up to the $FF terminator

Before executing REV, clear fuzzy outputs and initialize A, CCR, X, and Y. Load A with $FF. Clear the V bit. Load X with the address of the first 8-bit rule element in the list. Load Y with the base address for fuzzy inputs and fuzzy outputs.

X points to the element in the rule list that is being evaluated. REV updates X so that execution can resume correctly in case of an interrupt. After execution, X points to the address after the $FF separator at the end of the rule list.

Y points to the base address for the fuzzy inputs and fuzzy outputs. The value in Y does not change during execution.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

REV

Fuzzy Logic Rule Evaluation
(continued)

REV

A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value in A and writes the smaller value to A. After evaluation of all antecedents, A contains the smallest input value. This is the truth value used during consequent processing. For subsequent rules, REV reinitializes A with $FF when it finds an $FE separator. After execution, A contains the truth value for the last rule.

The V 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. The value of V changes as $FE separators are encountered. After execution, V should equal 1, because the last element before the $FF terminator should be a rule consequent. If V is 0 at the end of execution, the rule list is incorrect.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	?

V: Set unless rule structure is incorrect

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
REV	Special	18 3A	`Orfttx0`[1] `ff + Orf`[2]

NOTES:
1. The 3-cycle `ttx` loop is executed once for each element in the rule list.
2. These are additional cycles caused by an interrupt: `ff` is a 2-cycle exit sequence and `Orf` is a 3-cycle re-entry sequence. Execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

REVW Fuzzy Logic Rule Evaluation, Weighted REVW

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation MIN – MAX rule evaluation with optional rule weighting

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.

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.

In weighted evaluation, a table of 8-bit weighting factors, one per rule, must be stored in memory.

REVW begins with the address pointed to by the first rule antecedent, and evaluates successive fuzzy input values until it finds an $FFFE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. If weighted evaluation is enabled, the truth value is modified. Then, beginning with the address pointed to by the first consequent, REVW compares the truth value to successive fuzzy output values until it finds another $FFFE. If the truth value is greater than the current output value, REVW writes it to the output. Operation is similar to that of a MAXM instruction. Rule processing continues up to the $FFFF terminator.

Before executing REVW, clear fuzzy outputs and initialize A, CCR, X, and Y. Load A with $FF. Clear the V bit. Set or clear the C bit for weighted or unweighted evaluation. For weighted evaluation, load Y with the first item in a table of 8-bit weighting factors. Load X with the address of the first 16-bit element in the list.

X points to the element in the list that is being evaluated. REVW updates X so that execution can resume after an interrupt. After execution, X points to the address after the $FFFF separator at the end of the list.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

REVW Fuzzy Logic Rule Evaluation, Weighted (continued) REVW

Y points to the current weighting factor. REVW updates Y so that execution can resume after an interrupt. After execution, Y points to the last weighting factor used. Y does not change in unweighted evalutation.

A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A and writes the smaller value to A. After evaluation of all antecedents, A contains the smallest input value. In unweighted evaluation, this is the truth value for consequent processing. In weighted evaluation, it is multiplied by the quantity rule weight + 1, and the upper eight bits of the result replace the value in A. REVW reinitializes A with $FF when it finds an $FFFE separator. After execution, A holds the truth value for the last rule.

The V 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. The value of V changes as $FFFE separators are found. After 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	!

V: Set unless rule structure is incorrect

C: 1 selects weighted rule evaluation; 0 selects unweighted rule evaluation

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
REVW	Special	18 3B	`ORftTxO`[1] or `ORftTfRfO`[2] `ffff + ORf`[3]

NOTES:

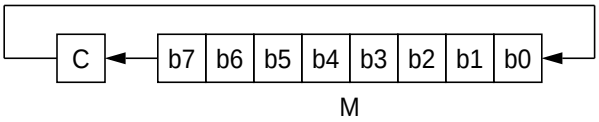
1. Weighting not enabled; the 3-cycle `tTx` loop is executed once for each element in the rule list.
2. Weighting enabled; the 3-cycle `tTx` loop expands to `tTfRf` for separators.
3. These are additional cycles caused by an interrupt: `ffff` is a 4-cycle exit sequence and `ORf` is a 3-cycle re-entry sequence. Execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

ROL

Rotate Left M

ROL

Operation



Shifts all bits of M one place to the left. Bit 0 is loaded from the C 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, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles

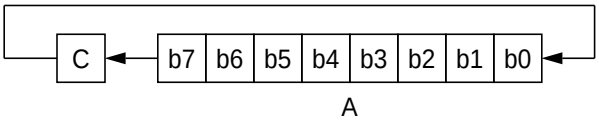
ROL *opr16a*
ROL *oprx0_xysppc*
ROL *oprx9*,*xysppc*
ROL *oprx16*,*xysppc*
ROL [D,*xysppc*]
ROL [*oprx16*,*xysppc*]
	EXT
	IDX
	IDX1
	IDX2
	[D,IDX]
	[IDX2]
		`75 hh ll`
		`65 xb`
		`65 xb ff`
		`65 xb ee ff`
		`65 xb`
		`65 xb ee ff`
			`rPw0`
			`rPw`
			`rPw0`
			`frPwP`
			`fIfrPw`
			`fIPrPw`

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ROLA — Rotate Left A — ROLA

Operation



Shifts all bits of A one place to the left. Bit 0 is loaded from the C 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, ROL HIGH could be used where LOW, MID and HIGH refer to the low, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

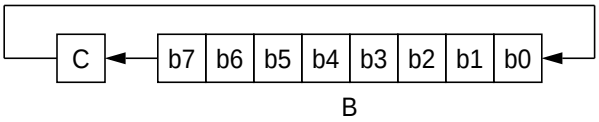
C: A7; set if the MSB of A was set before the shift; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ROLA	INH	45	0

ROLB Rotate Left B ROLB

Operation



Shifts all bits of B one place to the left. Bit 0 is loaded from the C 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, ROL HIGH could be used where LOW, MID and HIGH refer to the low, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

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

Code and CPU Cycles

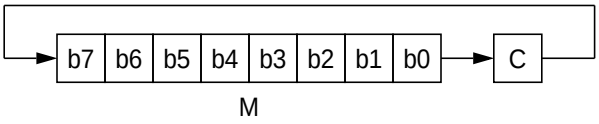
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ROLB	INH	55	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

ROR — Rotate Right M — ROR

Operation



Shifts all bits of M one place to the right. Bit 7 is loaded from the C 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, ROR LOW could be used where LOW, MID and HIGH refer to the low, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] \mid [\overline{N} \bullet C]$ for N and C after the shift; cleared otherwise

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

Code and CPU Cycles

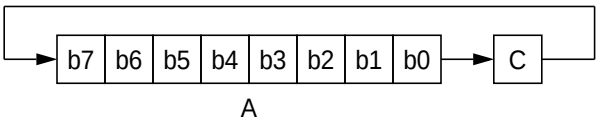
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
ROR *opr16a*	EXT	76 hh ll	rPw0
ROR *oprx0_xysppc*	IDX	66 xb	rPw
ROR *oprx9,xysppc*	IDX1	66 xb ff	rPw0
ROR *oprx16,xysppc*	IDX2	66 xb ee ff	frPwP
ROR [D,*xysppc*]	[D,IDX]	66 xb	fIfrPw
ROR [*oprx16*,*xysppc*]	[IDX2]	66 xb ee ff	fIPrPw

RORA

Rotate Right A

RORA

Operation



Shifts all bits of A one place to the right. Bit 7 is loaded from the C 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, ROR LOW could be used where LOW, MID and HIGH refer to the low, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] \mid [\overline{N} \bullet C]$ for N and C after the shift; cleared otherwise

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

Code and CPU Cycles

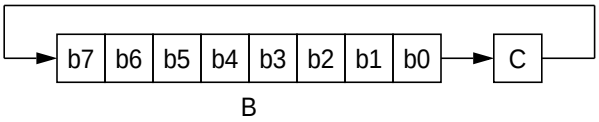
Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RORA	INH	46	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

RORB Rotate Right B RORB

Operation



Shifts all bits of B one place to the right. Bit 7 is loaded from the C 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, ROR LOW could be used where LOW, MID and HIGH refer to the low, middle, and high bytes of the 24-bit value, respectively.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] \mid [\overline{N} \bullet C]$ for N and C after the shift; cleared otherwise

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

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RORB	INH	56	0

RTC — Return from Call — RTC

Operation

$(M_{SP}) \Rightarrow PPAGE$
$(SP) + \$0001 \Rightarrow SP$
$(M_{SP}):(M_{SP+1}) \Rightarrow PC_H:PC_L$
$(SP) + \$0002 \Rightarrow SP$

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 MCUs that do not have expanded memory capability.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RTC	INH	0A	uUnfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

RTI — Return from Interrupt — RTI

Operation

$(M_{SP}) \Rightarrow CCR, (SP) + \$0001 \Rightarrow SP$
$(M_{SP}):(M_{SP+1}) \Rightarrow B:A, (SP) + \$0002 \Rightarrow SP$
$(M_{SP}):(M_{SP+1}) \Rightarrow X_H:X_L, (SP) + \$0004 \Rightarrow SP$
$(M_{SP}):(M_{SP+1}) \Rightarrow PC_H:PC_L, (SP) - \$0002 \Rightarrow SP$
$(M_{SP}):(M_{SP+1}) \Rightarrow Y_H:Y_L, (SP) + \$0004 \Rightarrow SP$

Restores the values of CPU registers CCR, B, A, X, PC, and Y from the stack.

The X 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 finishes restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched.

CCR Effects

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 only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RTI	INH	0B	uUUUUPPP or uUUUUfVfPPP[1]

NOTES:
1. RTI takes 11 cycles if an interrupt is pending.

RTS — Return from Subroutine — RTS

Operation

$(M_{SP}):(M_{SP+1}) \Rightarrow PC_H:PC_L$
$(SP) + \$0002 \Rightarrow SP$

Restores the value of PC from the stack and increments SP by two. Program execution continues at the address restored from the stack.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
RTS	INH	3D	UfPPP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SBA — Subtract B from A — SBA

Operation

(A) – (B) ⇒ A

Subtracts the value in B from the value in A and puts the result in A. The value in B is not affected. The C bit represents a borrow.

CCR Effects

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 • $\overline{B7}$ • $\overline{R7}$ | $\overline{A7}$ • B7 • R7; set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7}$ • B7 | B7 • R7 | R7 • $\overline{A7}$; set if the absolute value of B is larger than the absolute value of A; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SBA	INH	18 16	00

SBCA Subtract with Carry from A SBCA

Operation

(A) – (M) – C ⇒ A
or
(A) – imm – C ⇒ A

Subtracts either the value in M and the C bit or an immediate value and the C bit from the value in A. Puts the result in A. The C bit represents a borrow.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if the absolute value of the content of memory plus previous carry is larger than the absolute value of A; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SBCA #*opr8i*	IMM	82 ii	P
SBCA *opr8a*	DIR	92 dd	rPf
SBCA *opr16a*	EXT	B2 hh ll	rPO
SBCA *oprx0_xysppc*	IDX	A2 xb	rPf
SBCA *oprx9*,*xysppc*	IDX1	A2 xb ff	rPO
SBCA *oprx16*,*xysppc*	IDX2	A2 xb ee ff	frPP
SBCA [D,*xysppc*]	[D,IDX]	A2 xb	fIfrPf
SBCA [*oprx16*,*xysppc*]	[IDX2]	A2 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SBCB Subtract with Carry from B SBCB

Operation

(B) – (M) – C ⇒ B
or
(B) – imm – C ⇒ B

Subtracts either the value in M and the C bit or an immediate value and the C bit from the value in B. Puts the result in B. The C bit represents a borrow.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

C: $\overline{B7}$ • M7 | M7 • R7 | R7 • $\overline{B7}$; set if the absolute value in M plus previous carry is larger than the absolute value in B; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SBCB #*opr8i*	IMM	C2 ii	P
SBCB *opr8a*	DIR	D2 dd	rPf
SBCB *opr16a*	EXT	F2 hh ll	rPO
SBCB *oprx0_xysppc*	IDX	E2 xb	rPf
SBCB *oprx9,xysppc*	IDX1	E2 xb ff	rPO
SBCB *oprx16,xysppc*	IDX2	E2 xb ee ff	frPP
SBCB [D,*xysppc*]	[D,IDX]	E2 xb	fIfrPf
SBCB [*oprx16,xysppc*]	[IDX2]	E2 xb ee ff	fIPrPf

SEC

Set C
(same as ORCC #$01)

SEC

Operation (CCR) | $01 ⇒ CCR

Performs a logical inclusive OR of the value in the CCR and $01. Puts the result in the CCR, setting the C bit. SEC assembles as ORCC #$01.

SEC can be used to initialize the C bit prior to a shift or rotate instruction involving the C bit.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	1

C: Set

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SEC	IMM	14 01	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SEI

Set I
(same as ORCC #$10)

SEI

Operation

(CCR) | $10 ⇒ CCR

Performs a logical inclusive OR of the value in the CCR and $10. Puts the result in the CCR, setting the I bit. SEI assembles as ORCC #$10. When the I bit is set, all I-maskable interrupts are inhibited.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: Set

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SEI	IMM	14 10	P

SEV

Set V
(same as ORCC #$02)

SEV

Operation (CCR) | $02 ⇒ CCR

Performs a logical inclusive OR of the value in the CCR and $02. Puts the result in the CCR, setting the V bit. SEV assembles as ORCC #$02.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	1	–

V: Set

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SEV	IMM	14 02	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SEX — Sign Extend — SEX

Operation

If r1 bit 7 = 0, then $00:(r1) ⇒ r2
If r1 bit 7 = 1, then $FF:(r1) ⇒ r2

Transfers the two's complement value in A, B, or CCR to the low byte of D, X, Y, or SP. Loads the high byte with $00 if bit 7 is 0 or $FF if bit 7 is 1. The result is the 16-bit sign-extended version of the original 8-bit value. SEX is an alternate mnemonic for the TFR r1,r2 instruction, The value in the original register does not change except in the case of SEX A,D (D is A:B).

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SEX *abc,dxysp*	INH	B7 eb	P

Sign-Extend Postbyte (eb) Coding			
Source Form	**Postbyte**	**Object Code**	**Sign Extension**
SEX A,TMP2	0000 X011	B7 03	($00 or $FF):A ⇒ TMP2
SEX A,D	0000 X100	B7 04	($00 or $FF):A ⇒ D
SEX A,X	0000 X101	B7 05	($00 or $FF):A ⇒ X
SEX A,Y	0000 X110	B7 06	($00 or $FF):A ⇒ Y
SEX A,SP	0000 X111	B7 07	($00 or $FF):A ⇒ SP
SEX B,TMP2	0001 X011	B7 13	($00 or $FF):B ⇒ TMP2
SEX B,D	0001 X100	B7 14	($00 or $FF):B ⇒ D
SEX B,X	0001 X101	B7 15	($00 or $FF):B ⇒ X
SEX B,Y	0001 X110	B7 16	($00 or $FF):B ⇒ Y
SEX B,SP	0001 X111	B7 17	($00 or $FF):B ⇒ SP
SEX CCR,TMP2	0010 X011	B7 23	($00 or $FF):CCR ⇒ TMP2
SEX CCR,D	0010 X100	B7 24	($00 or $FF):CCR ⇒ D
SEX CCR,X	0010 X101	B7 25	($00 or $FF):CCR ⇒ X
SEX CCR,Y	0010 X110	B7 26	($00 or $FF):CCR ⇒ Y
SEX CCR,SP	0010 X111	B7 27	($00 or $FF):CCR ⇒ SP

STAA Store Accumulator A STAA

Operation

(A) ⇒ M

Stores the value in A in M. The value in A does not change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STAA *opr8a*	DIR	5A dd	Pw
STAA *opr16a*	EXT	7A hh ll	PwO
STAA *oprx0_xysppc*	IDX	6A xb	Pw
STAA *oprx9*,*xysppc*	IDX1	6A xb ff	PwO
STAA *oprx16*,*xysppc*	IDX2	6A xb ee ff	PwP
STAA [D,*xysppc*]	[D,IDX]	6A xb	PIfw
STAA [*oprx16*,*xysppc*]	[IDX2]	6A xb ee ff	PIPw

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

STAB Store Accumulator B STAB

Operation

(B) ⇒ M

Stores the value in B in M. The value in B does not change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STAB *opr8a*	DIR	5B dd	Pw
STAB *opr16a*	EXT	7B hh ll	PwO
STAB *oprx0_xysppc*	IDX	6B xb	Pw
STAB *oprx9*,*xysppc*	IDX1	6B xb ff	PwO
STAB *oprx16*,*xysppc*	IDX2	6B xb ee ff	PwP
STAB [D,*xysppc*]	[D,IDX]	6B xb	PIfw
STAB [*oprx16*,*xysppc*]	[IDX2]	6B xb ee ff	PIPw

STD Store D STD

Operation (A):(B) ⇒ M:M + 1

Stores the value in A in M and the the value in B in M:M + 1. The values in A and B do not change.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STD *opr8a*	DIR	5C dd	PW
STD *opr16a*	EXT	7C hh ll	PWO
STD *oprx0_xysppc*	IDX	6C xb	PW
STD *oprx9*,*xysppc*	IDX1	6C xb ff	PWO
STD *oprx16*,*xysppc*	IDX2	6C xb ee ff	PWP
STD [D,*xysppc*]	[D,IDX]	6C xb	PIfW
STD [*oprx16*,*xysppc*]	[IDX2]	6C xb ee ff	PIPW

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

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

When the S bit is set, STOP is disabled and operates like a 2-cycle NOP instruction. When S is cleared, STOP stacks CPU registers, stops all system clocks, and puts the device in standby mode. Standby mode minimizes power consumption. The contents of registers and the states of I/O pins do not change.

Asserting $\overline{\text{RESET}}$, $\overline{\text{XIRQ}}$, or $\overline{\text{IRQ}}$ ends standby mode. If the clock reference crystal also stops during low-power mode, crystal startup delay lengthens recovery time.

If $\overline{\text{XIRQ}}$ is asserted while the X mask bit = 0 ($\overline{\text{XIRQ}}$ interrupts enabled), execution resumes with a vector fetch for the $\overline{\text{XIRQ}}$ interrupt. If the X mask bit = 1 ($\overline{\text{XIRQ}}$ interrupts disabled), a 2-cycle recovery sequence, including an O cycle, adjusts the instruction queue, and execution continues with the next instruction after STOP.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STOP	INH	18 3E	`OOSSSSsf` (enter stop mode) `fVfPPP` (exit stop mode) `ff` (continue stop mode) `OO` (if stop mode disabled by S = 1)

STS Store SP STS

Operation

$(SP_H):(SP_L) \Rightarrow M:M + 1$

Stores the high byte of SP in M and the low byte in M + 1.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STS *opr8a*	DIR	5F dd	PW
STS *opr16a*	EXT	7F hh ll	PWO
STS *oprx0_xysppc*	IDX	6F xb	PW
STS *oprx9,xysppc*	IDX1	6F xb ff	PWO
STS *oprx16,xysppc*	IDX2	6F xb ee ff	PWP
STS [D,*xysppc*]	[D,IDX]	6F xb	PIfW
STS [*oprx16,xysppc*]	[IDX2]	6F xb ee ff	PIPW

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

STX — Store X — STX

Operation

$(X_H):(X_L) \Rightarrow M:M + 1$

Stores the high byte of X in M and the low byte in M + 1.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STX *opr8a*	DIR	5E dd	PW
STX *opr16a*	EXT	7E hh ll	PWO
STX *oprx0_xysppc*	IDX	6E xb	PW
STX *oprx9,xysppc*	IDX1	6E xb ff	PWO
STX *oprx16,xysppc*	IDX2	6E xb ee ff	PWP
STX [D,*xysppc*]	[D,IDX]	6E xb	PIfW
STX [*oprx16,xysppc*]	[IDX2]	6E xb ee ff	PIPW

STY — Store Y — STY

Operation $(Y_H):(Y_L) \Rightarrow M:M + 1$

Stores the high by of Y in M and the low byte in M + 1.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
STY *opr8a*	DIR	5D dd	PW
STY *opr16a*	EXT	7D hh ll	PWO
STY *oprx0_xysppc*	IDX	6D xb	PW
STY *oprx9*,*xysppc*	IDX1	6D xb ff	PWO
STY *oprx16*,*xysppc*	IDX2	6D xb ee ff	PWP
STY [D,*xysppc*]	[D,IDX]	6D xb	PIfW
STY [*oprx16*,*xysppc*]	[IDX2]	6D xb ee ff	PIPW

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SUBA — Subtract from A — SUBA

Operation

(A) – (M) ⇒ A
or
(A) – imm ⇒ A

Subtracts either the value in M or an immediate value from the value in A. Puts the result in A. The C bit represents a borrow.

CCR Effects

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 • $\overline{M7}$ • $\overline{R7}$ | $\overline{A7}$ • M7 • R7; set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7}$ • M7 | M7 • R7 | R7 • $\overline{A7}$; set if the value in M is larger than the value in A; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SUBA #*opr8i*	IMM	80 ii	P
SUBA *opr8a*	DIR	90 dd	rPf
SUBA *opr16a*	EXT	B0 hh ll	rPO
SUBA *oprx0_xysppc*	IDX	A0 xb	rPf
SUBA *oprx9,xysppc*	IDX1	A0 xb ff	rPO
SUBA *oprx16,xysppc*	IDX2	A0 xb ee ff	frPP
SUBA [D,*xysppc*]	[D,IDX]	A0 xb	fIfrPf
SUBA [*oprx16*,*xysppc*]	[IDX2]	A0 xb ee ff	fIPrPf

SUBB Subtract from B SUBB

Operation

(B) – (M) ⇒ B
or
(B) – imm ⇒ B

Subtracts either the value in M or an immediate value from the value in B. Puts the result in B. The C bit represents a borrow.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is $00; cleared otherwise

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

C: $\overline{B7}$ • M7 | M7 • R7 | R7 • $\overline{B7}$; set if the value in M is larger than the value in B; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SUBB #*opr8i*	IMM	C0 ii	P
SUBB *opr8a*	DIR	D0 dd	rPf
SUBB *opr16a*	EXT	F0 hh ll	rPO
SUBB *oprx0_xysppc*	IDX	E0 xb	rPf
SUBB *oprx9,xysppc*	IDX1	E0 xb ff	rPO
SUBB *oprx16,xysppc*	IDX2	E0 xb ee ff	frPP
SUBB [D,*xysppc*]	[D,IDX]	E0 xb	fIfrPf
SUBB [*oprx16,xysppc*]	[IDX2]	E0 xb ee ff	fIPrPf

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

SUBD — Subtract from D — SUBD

Operation

(A):(B) – (M):(M + 1) ⇒ A:B
or
(A):(B) – imm ⇒ A:B

Subtracts either the value in M:M + 1 or an immediate value from the value in D. Puts the result in D. The C bit represents a borrow.

CCR Effects

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

C: $\overline{D15}$ • M15 | M15 • R15 | R15 • $\overline{D15}$; set if the value in M is larger than the value in D; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SUBD #*opr16i*	IMM	83 jj kk	PO
SUBD *opr8a*	DIR	93 dd	RPf
SUBD *opr16a*	EXT	B3 hh ll	RPO
SUBD *oprx0_xysppc*	IDX	A3 xb	RPf
SUBD *oprx9,xyssp*	IDX1	A3 xb ff	RPO
SUBD *oprx16,xysppc*	IDX2	A3 xb ee ff	fRPP
SUBD [D,*xysppc*]	[D,IDX]	A3 xb	fIfRPf
SUBD [*oprx16*,*xysppc*]	[IDX2]	A3 xb ee ff	fIPRPf

SWI — Software Interrupt — SWI

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}
1 ⇒ I
SWI vector ⇒ PC

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 and CPU registers Y, X, B, A, and CCR, decrementing SP before each item is stacked. Sets the I bit and loads PC with the SWI vector. Instruction execution resumes at the address to which the vector points. SWI is not affected by the I bit.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: Set

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
SWI	INH	3F	VSPSSPSsP[1]

NOTES:
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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TAB — Transfer A to B — TAB

Operation

(A) ⇒ B

Loads the value in A into B. The former value in B is lost; the value in A does not change. Unlike the general transfer instruction TFR A,B which does not affect condition code bits, the TAB instruction affects the N, Z, and V bits.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TAB	INH	18 0E	00

TAP

Transfer A to CCR
(same as TFR A,CCR)

TAP

Operation (A) ⇒ CCR

Loads the value in A into the CCR. The value in A does not change. The X 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 one-cycle delay before the system allows interrupt requests. This delay prevents interrupts from occurring between instructions in the sequences CLI, WAI and CLI, SEI. TAP assembles as TFR A,CCR.

CCR Effects

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

Condition codes take on the value of the corresponding bit of accumulator A, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TAP	INH	B7 02	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TBA — Transfer B to A — TBA

Operation

(B) ⇒ A

Loads the value in B into A. The former value in A is lost; the value in B does not change. Unlike the general transfer instruction TFR B,A, which does not affect condition code bits, the TBA instruction affects the N, Z, and V bits.

CCR Effects

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: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TBA	INH	18 0F	00

TBEQ Test and Branch if Equal to Zero TBEQ

Operation If (counter) = 0, then (PC) + $0003 + Rel ⇒ PC

Tests the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter register reaches zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TBEQ *abdxysp,rel9*	REL (9-bit)	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding				
Source Form	**Postbyte[1]**	**Object Code**	**Counter Register**	**Offset**
TBEQ A, *rel9*	0100 X000	`04 40 rr`	A	Positive
TBEQ B, *rel9*	0100 X001	`04 41 rr`	B	
TBEQ D, *rel9*	0100 X100	`04 44 rr`	D	
TBEQ X, *rel9*	0100 X101	`04 45 rr`	X	
TBEQ Y, *rel9*	0100 X110	`04 46 rr`	Y	
TBEQ SP, *rel9*	0100 X111	`04 47 rr`	SP	
TBEQ A, *rel9*	0101 X000	`04 50 rr`	A	Negative
TBEQ B, *rel9*	0101 X001	`04 51 rr`	B	
TBEQ D, *rel9*	0101 X100	`04 54 rr`	D	
TBEQ X, *rel9*	0101 X101	`04 55 rr`	X	
TBEQ Y, *rel9*	0101 X110	`04 56 rr`	Y	
TBEQ SP, *rel9*	0101 X111	`04 57 rr`	SP	

NOTES:

1. Bits 7:6:5 select TBEQ or TBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

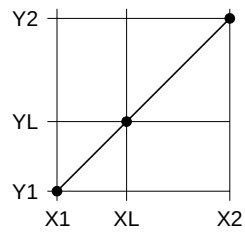
PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TBL Table Lookup and Interpolate TBL

Operation

$(M) + [(B) \times ((M + 1) - (M))] \Rightarrow A$

Linearly interpolates and stores in A one of 256 values between a pair of data entries, Y1 and Y2, in a lookup table. Data entries represent y coordinates of line segment endpoints. Table entries and interpolated results are 8-bit values.



Before executing TBL, point an indexing register at the Y1 value closest to but less than or equal to the Y value to interpolate. Point to Y1 using any indexed addressing mode except indirect, 9-bit offset, and 16-bit offset. The next table entry after Y1 is Y2. Load B with a binary fraction (radix point to the left of the MSB) representing the ratio:

$(XL - X1) \div (X2 - X1)$
where
X1 = Y1 and X2 = Y2
XL is the x coordinate of the value to interpolate

The 8-bit unrounded result, YL, is calculated using the expression:
$YL = Y1 + [(B) \times (Y2 - Y1)]$
where
Y1 = 8-bit data entry pointed to by the effective address
Y2 = 8-bit data entry pointed to by the effective address plus one

The 16-bit intermediate value $(B) \times (Y2 - Y1)$ has a radix point between bits 7 and 8. The result in A is the sum of the upper 8 bits (the integer part) of the intermediate 16-bit value and the 8-bit value Y1.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

N: Set if MSB of result is set; cleared otherwise
Z: Set if result is \$00; cleared otherwise
C: Set if result can be rounded up; cleared otherwise

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TBL *oprx0_xysppc*	IDX	18 3D xb	ORfffP

TBNE Test and Branch if Not Equal to Zero TBNE

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Operation

If (counter) ≠ 0, then (PC) + $0003 + Rel ⇒ PC

Tests the counter register A, B, D, X, Y, or SP. Branches to a relative destination if the counter does not reach zero. Rel is a 9-bit two's complement offset for branching forward or backward in memory. Branching range is $100 to $0FF (–256 to +255) from the address following the last byte of object code in the instruction.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TBNE *abdxysp,rel9*	REL (9-bit)	`04 lb rr`	`PPP` (branch) `PPO` (no branch)

Loop Primitive Postbyte (`lb`) Coding				
Source Form	**Postbyte[1]**	**Object Code**	**Counter Register**	**Offset**
TBNE A, *rel9*	0110 X000	`04 60 rr`	A	Positive
TBNE B, *rel9*	0110 X001	`04 61 rr`	B	
TBNE D, *rel9*	0110 X100	`04 64 rr`	D	
TBNE X, *rel9*	0110 X101	`04 65 rr`	X	
TBNE Y, *rel9*	0110 X110	`04 66 rr`	Y	
TBNE SP, *rel9*	0110 X111	`04 67 rr`	SP	
TBNE A, *rel9*	0111 X000	`04 70 rr`	A	Negative
TBNE B, *rel9*	0111 X001	`04 71 rr`	B	
TBNE D, *rel9*	0111 X100	`04 74 rr`	D	
TBNE X, *rel9*	0111 X101	`04 75 rr`	X	
TBNE Y, *rel9*	0111 X110	`04 76 rr`	Y	
TBNE SP, *rel9*	0111 X111	`04 77 rr`	SP	

NOTES:

1. Bits 7:6:5 select TBEQ or TBNE; bit 4 is the offset sign bit: bit 3 is not used; bits 2:1:0 select the counter register.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TFR Transfer Register TFR

Operation

See the table on the next page.

Transfers the value in a source register A, B, CCR, D, X, Y, or SP to a destination register A, B, CCR, D, X, Y, or SP. Transfers involving TMP2 and TMP3 are reserved for Motorola use.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

or

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

CCR bits affected only when the CCR is the destination register. The X 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 X can only be set by a reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TFR *abcdxysp,abcdxysp*	INH	B7 eb	P

TFR

Transfer Register
(continued)

TFR

Transfer Postbyte (eb) Coding							
Source Form	**Postbyte**	**Object Code**	**Transfer**	**Source Form**	**Postbyte**	**Object Code**	**Transfer**
TFR A,A	0000 X000	B7 00	A ⇒ A	TFR B,A	0100 X000	B7 40	B ⇒ A
TFR A,B	0000 X001	B7 01	A ⇒ B	TFR B,B	0100 X001	B7 41	B ⇒ B
TFR A,CCR	0000 X010	B7 02	A ⇒ CCR	TFR B,CCR	0100 X010	B7 42	B ⇒ CCR
TFR A,TMP2	0000 X011	B7 03	sex:A ⇒ TMP2	TFR D,TMP2	0100 X011	B7 43	D ⇒ TMP2
TFR A,D	0000 X100	B7 04	sex:A ⇒ D	TFR D,D	0100 X100	B7 44	D ⇒ D
TFR A,X	0000 X101	B7 05	sex:A ⇒ X	TFR D,X	0100 X101	B7 45	D ⇒ X
TFR A,Y	0000 X110	B7 06	sex:A ⇒ Y	TFR D,Y	0100 X110	B7 46	D ⇒ Y
TFR A,SP	0000 X111	B7 07	sex:A ⇒ SP	TFR D,SP	0100 X111	B7 47	D ⇒ SP
TFR B,A	0001 X000	B7 10	B ⇒ A	TFR X,A	0101 X000	B7 50	X_L ⇒ A
TFR B,B	0001 X001	B7 11	B ⇒ B	TFR X,B	0101 X001	B7 51	X_L⇒ B
TFR B,CCR	0001 X010	B7 12	B ⇒ CCR	TFR X,CCR	0101 X010	B7 52	X_L⇒ CCR
TFR B,TMP2	0001 X011	B7 13	sex:B ⇒ TMP2	TFR X,TMP2	0101 X011	B7 53	X ⇒ TMP2
TFR B,D	0001 X100	B7 14	sex:B ⇒ D	TFR X,D	0101 X100	B7 54	X ⇒ D
TFR B,X	0001 X101	B7 15	sex:B ⇒ X	TFR X,X	0101 X101	B7 55	X ⇒ X
TFR B,Y	0001 X110	B7 16	sex:B ⇒ Y	TFR X,Y	0101 X110	B7 56	X ⇒ Y
TFR B,SP	0001 X111	B7 17	sex:B ⇒ SP	TFR X,SP	0101 X111	B7 57	X ⇒ SP
TFR CCR,A	0010 X000	B7 20	CCR ⇒ A	TFR Y,A	0110 X000	B7 60	Y_L ⇒ A
TFR CCR,B	0010 X001	B7 21	CCR ⇒ B	TFR Y,B	0110 X001	B7 61	Y_L ⇒ B
TFR CCR,CCR	0010 X010	B7 22	CCR ⇒ CCR	TFR Y,CCR	0110 X010	B7 62	Y_L ⇒ CCR
TFR CCR,TMP2	0010 X011	B7 23	sex:CCR ⇒ TMP2	TFR Y,TMP2	0110 X011	B7 63	Y ⇒ TMP2
TFR CCR,D	0010 X100	B7 24	sex:CCR ⇒ D	TFR Y,D	0110 X100	B7 64	Y ⇒ D
TFR CCR,X	0010 X101	B7 25	sex:CCR ⇒ X	TFR Y,X	0110 X101	B7 65	Y ⇒ X
TFR CCR,Y	0010 X110	B7 26	sex:CCR ⇒ Y	TFR Y,Y	0110 X110	B7 66	Y ⇒ Y
TFR CCR,SP	0010 X111	B7 27	sex:CCR ⇒ SP	TFR Y,SP	0110 X111	B7 67	Y ⇒ SP
TFR TMP3,A	0011 X000	B7 30	$TMP3_L$ ⇒ A	TFR SP,A	0111 X000	B7 70	SP_L ⇒ A
TFR TMP3,B	0011 X001	B7 31	$TMP3_L$ ⇒ B	TFR SP,B	0111 X001	B7 71	SP_L ⇒ B
TFR TMP3,CCR	0011 X010	B7 32	$TMP3_L$ ⇒ CCR	TFR SP,CCR	0111 X010	B7 72	SP_L ⇒ CCR
TFR TMP3,TMP2	0011 X011	B7 33	TMP3 ⇒ TMP2	TFR SP,TMP2	0111 X011	B7 73	SP ⇒ TMP2
TFR TMP3,D	0011 X100	B7 34	TMP3 ⇒ D	TFR SP,D	0111 X100	B7 74	SP ⇒ D
TFR TMP3,X	0011 X101	BY 35	TMP3 ⇒ X	TFR SP,X	0111 X101	B7 75	SP ⇒ X
TFR TMP3,Y	0011 X110	B7 36	TMP3 ⇒ Y	TFR SP,Y	0111 X110	B7 76	SP ⇒ Y
TFR TMP3,SP	0011 X111	B7 37	TMP3 ⇒ SP	TFR SP,SP	0111 X111	B7 77	SP ⇒ SP

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TPA

Transfer CCR to A
(same as TFR CCR,A)

TPA

Operation (CCR) ⇒ A

Transfers the value in CCR to A. The CCR value does not change. TPA assembles as TFR CCR,A.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TPA	INH	B7 20	P

TRAP Unimplemented Opcode Trap TRAP

Operation

(SP) – \$0002 ⇒ SP; RTN_H:RTN_L ⇒ M_{SP}:$M_{SP + 1}$
(SP) – \$0002 ⇒ SP; (Y_H):(Y_L) ⇒ M_{SP}:$M_{SP + 1}$
(SP) – \$0002 ⇒ SP; (X_H):(X_L) ⇒ M_{SP}:$M_{SP + 1}$
(SP) – \$0002 ⇒ SP; (B):(A) ⇒ M_{SP}:$M_{SP + 1}$
(SP) – \$0001 ⇒ SP; (CCR) ⇒ M_{SP}
1 ⇒ I
(trap vector) ⇒ PC

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 two of the opcode map are used. If the CPU attempts to execute one of the unimplemented opcodes on page two, 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, CPU registers Y, X, B, A, and CCR, decrementing the SP before each item is stacked. The I 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.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	1	–	–	–	–

I: Set

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TRAP *trapnum*	INH	18 tn[1]	OVSPSSPSsP

NOTES:
1. The value tn is an unimplemented page two opcode from \$30 to \$39 or \$40 to \$FF.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TST — Test M — TST

Operation

(M) – $00

Subtracts $00 from the value in M. The condition code bits reflect the result. The value in M does not change.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than 0, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Effects

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: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TST *opr16a*	EXT	F7 hh ll	rPO
TST *oprx0_xysppc*	IDX	E7 xb	rPf
TST *oprx9*,*xysppc*	IDX1	E7 xb ff	rPO
TST *oprx16*,*xysppc*	IDX2	E7 xb ee ff	frPP
TST [D,*xysppc*]	[D,IDX]	E7 xb	fIfrPf
TST [*oprx16*,*xysppc*]	[IDX2]	E7 xb ee ff	fIPrPf

TSTA Test A TSTA

Operation

(A) – $00

Subtracts $00 from the value in A. The condition code bits reflect the result. The value in A does not change.

The TSTA instruction provides limited information when testing unsigned values. Since no unsigned value is less than 0, BLO and BLS have no utility following TSTA. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Effects

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: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TSTA	INH	97	0

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TSTB Test B TSTB

Operation

(B) – $00

Subtracts $00 from the value in B. The condition code bits reflect the result. The value in B does not change.

The TSTB instruction provides limited information when testing unsigned values. Since no unsigned value is less than 0, BLO and BLS have no utility following TSTB. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Effects

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: Cleared
C: Cleared

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TSTB	INH	D7	0

TSX

Transfer SP to X
(same as TFR SP,X)

TSX

Operation

(SP) ⇒ X

Transfers the value in SP to X. The value in SP does not change. After a TSX instruction, X points at the last value that was stored on the stack. TSX assembles as TFR SP,X.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TSX	INH	B7 75	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TSY

Transfer SP to Y
(same as TFR SP,Y)

TSY

Operation (SP) ⇒ Y

Transfers the value in SP to Y. The value in SP does not change. After a TSY instruction, Y points at the last value that was stored on the stack. TPY assembles as TFR SP,Y.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TSY	INH	B7 76	P

TXS

Transfer X to SP
(same as TFR X,SP)

TXS

Operation (X) ⇒ SP

Transfers the value in X to SP. The value in X does not change. TXS assembles as TFR X,SP.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TXS	INH	B7 57	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

TYS

Transfer Y to SP
(same as TFR Y,SP)

TYS

Operation (Y) ⇒ SP

Transfers the value in Y to SP. The value in Y does not change. TYS assembles as TFR Y,SP.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
TYS	INH	B7 67	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

Puts the CPU into a wait state. Uses the address of the instruction following WAI as a return address. Stacks the return address and CPU registers Y, X, B, A, and CCR, decrementing 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) and fetches the vector corresponding to the interrupt sensed. Program execution continues at the location to which the vector points.

CCR Effects

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 causes the I bit (and the X bit, if the interrupt was $\overline{\text{XIRQ}}$) to be set as the interrupt vector is fetched.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
WAI	INH	3E	`0SSSSsf` (before interrupt) `fVfPPP` (after interrupt)

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

WAV — Calculate Weighted Average — WAV

Operation

Partial product = (M pointed to by X) × (M pointed to by Y)
Sum of products (24-bit SOP) = previous SOP + partial product
Sum of weights (16-bit SOW) = previous SOW + (M pointed to by Y)
(X) + $0001 ⇒ X; (Y) + $0001 ⇒ Y, (B) – $01 ⇒ B
Repeat until B = $00; leave SOP in Y:D, SOW in X

Calculates weighted averages of values in memory. Uses indexed (X) addressing to access one source operand list, and indexed (Y) addressing mode to access another source operand list. Accumulator B is the counter that controls the number of elements to be included in the weighted average.

For each data point pair, a 24-bit SOP and a 16-bit SOW accumulates in temporary registers. When B reaches zero (no more data pairs), the SOP goes in Y:D. The SOW goes in X. To get the final weighted average, divide (Y):(D) by (X) with an EDIV after the WAV.

WAV can be interrupted. If an interrupt occurs, the intermediate results (six bytes) are stacked in the order $SOW_{[15:0]}$, $SOP_{[15:0]}$, $00:SOP_{[23:16]}$. The wavr pseudoinstruction resumes WAV execution. The interrupt mechanism is reentrant; new WAV instructions can be started and interrupted while a previous WAV instruction is interrupted.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	?	–	?	1	?	?

Z: Set
H, N, V, and C may be altered by this instruction.

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
WAV	Special	18 3C	`Offrrrrrf0`[1] `SSS+UUUrr`[2]

NOTES:
1. The 7-cycle loop `frrrrrf` is the loop for one iteration of SOP and SOW accumulation.
2. These are additional cycles caused by interrupt: `SSS` is a three-cycle exit sequence and `UUUrr` is a five-cycle re-entry sequence. Six extra bytes of stack are used for intermediate values.

XGDX

Exchange D with X
(same as EXG D,X)

XGDX

Operation (D) ⇔ (X)

Exchanges the value in D with the value in X. XGDX assembles as EXG D,X.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
XGDX	INH	B7 C5	P

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

XGDY

Exchange D with Y
(same as EXG D,Y)

XGDY

Operation (D) ⇔ (Y)

Exchanges the value in D with the value in Y. XGDY assembles as EXG D,Y.

CCR Effects

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Code and CPU Cycles

Source Form	Address Mode	Machine Code (Hex)	CPU Cycles
XGDY	INH	B7 C6	P

Appendix C Fuzzy Logic Support

C.1 General

This section describes the use of fuzzy logic in control systems, discusses the fuzzy logic instructions, and provides examples of fuzzy logic programs.

C.2 Introduction

There are four instructions that perform fuzzy logic tasks. Several other instructions are also useful in fuzzy logic programs.

This section explains the basic fuzzy logic algorithm for which the four fuzzy logic instructions are intended. Each fuzzy logic instruction is then explained in detail. Finally, other custom fuzzy logic algorithms are discussed, with emphasis on other useful instructions.

The four fuzzy logic instructions are:

- MEM — evaluates trapezoidal membership functions
- REV and REVW — perform unweighted or weighted MIN-MAX rule evaluation
- WAV — performs weighted average defuzzification on singleton output membership functions

Other instructions that are useful for custom fuzzy logic programs include MINA, EMIND, MAXM, EMAXM, TBL, ETBL, and EMACS. For higher resolution fuzzy programs, the extended math instructions are also useful. Indexed addressing modes help simplify access to fuzzy logic data structures stored as lists or tabular data structures in memory.

C.3 Fuzzy Logic Basics

This overview of basic fuzzy logic concepts is the background for a detailed explanation of the fuzzy logic instructions.

In general, fuzzy logic provides for definitions of sets that have fuzzy boundaries rather than the crisp boundaries of Aristotelian logic. The sets can overlap so that, for a particular input value, one or more sets may be true at the same time. As the input varies out of the range of one set and 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 that emulate human perceptions such as “temperature is warm,” in which humans recognize gradual boundaries. This perception seems to be important to the human ability to solve certain types of complex problems that elude traditional control methods.

Fuzzy sets are a means of using linguistic expressions such as “temperature is warm” as labels in rules that can be evaluated with a high degree of numerical precision and repeatability. A specific set of input conditions always produces the same result, just as a conventional control system does.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

A microcontroller-based fuzzy logic control system has two parts:

- A fuzzy inference kernel which is executed periodically to determine system outputs based on current system inputs
- A knowledge base which contains membership functions and rules

Figure C-1 is a block diagram of this kind of 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 nonvolatile memory. A fuzzy inference kernel can be written by a programmer who does not know how the application system works. All that the programmer needs to do with knowledge base information is store it in the memory locations used by the kernel.

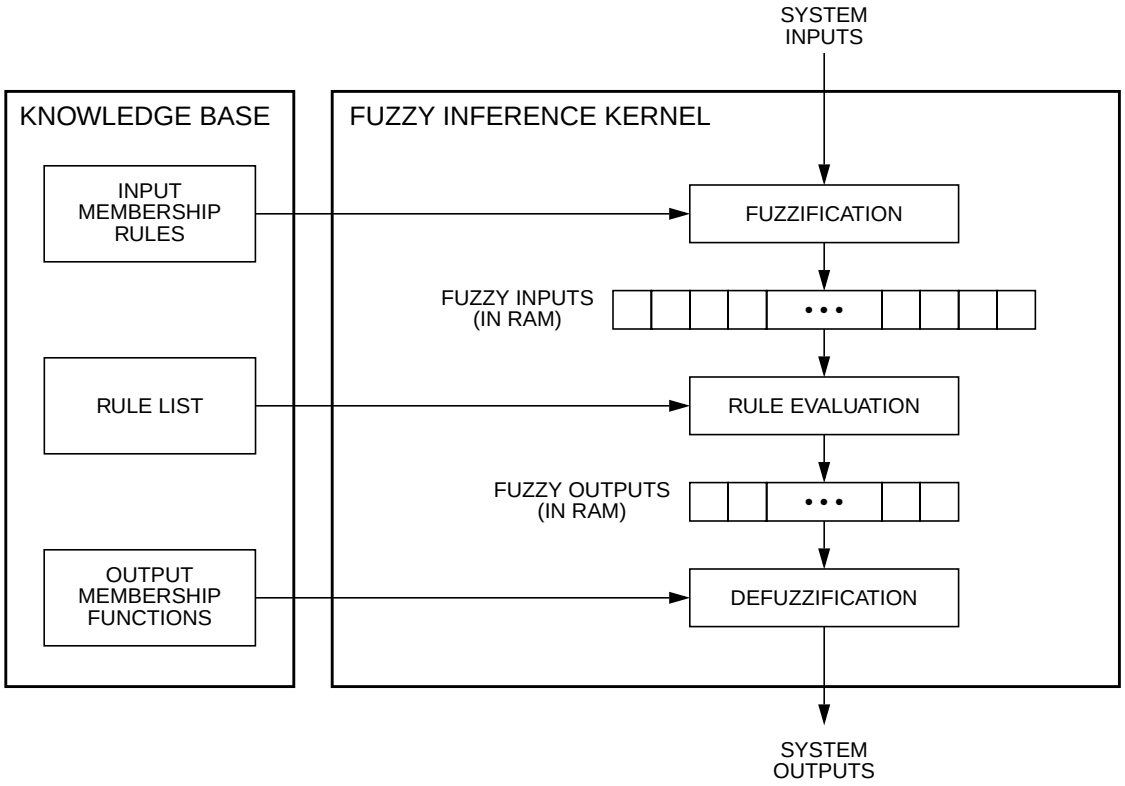


Figure C-1 Block Diagram of a Fuzzy Logic System

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. On the other hand, 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.

C.3.1 Fuzzification (MEM)

During the fuzzification step, the current system input values are compared to 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 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 C-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 ($00 or 0%) to completely true ($FF or 100%). 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% true ($40). The value $40 is stored to a RAM location, and is called a fuzzy input (in this case, the fuzzy input for “the temperature is warm”). There is a RAM location for each fuzzy input (for each label of each system input).

When the fuzzification step begins, the current value of the system input is in an accumulator, 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

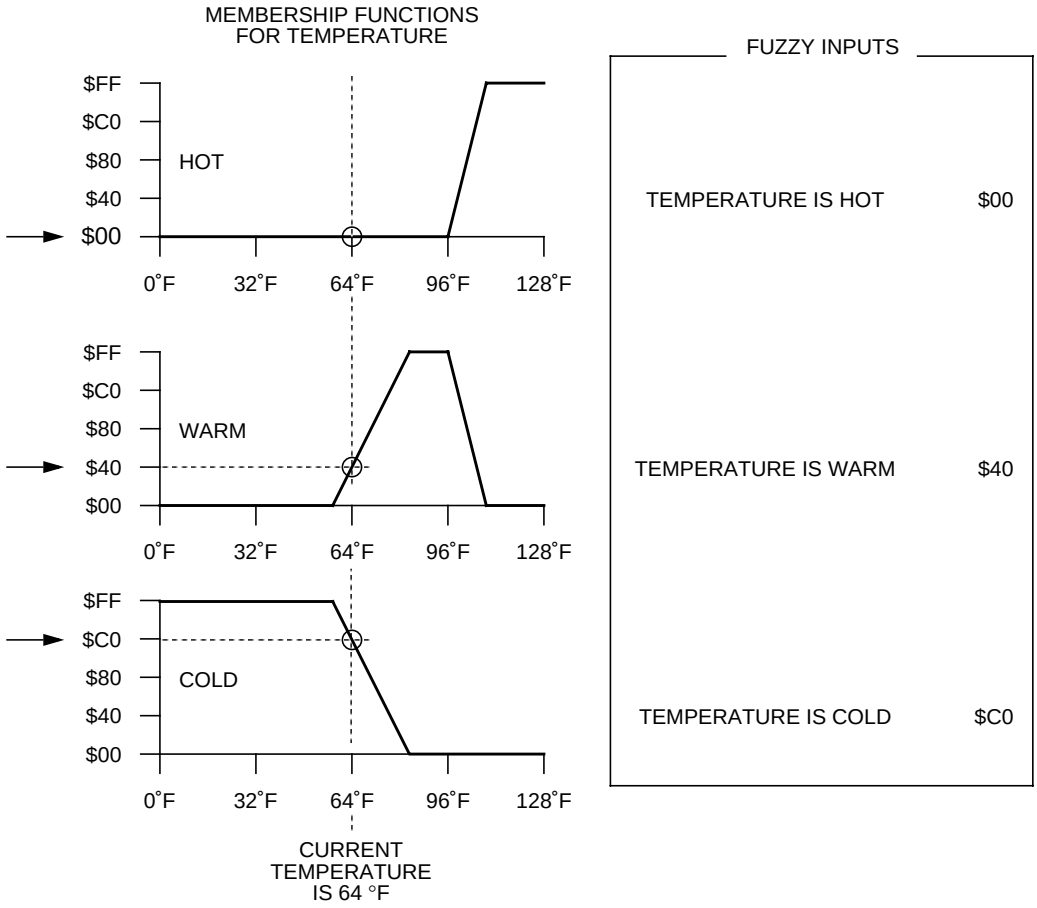


Figure C-2 Fuzzification Using Membership Functions

C.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.

There are two variations of the rule evaluation instruction. The REV instruction provides for unweighted rules that are considered to be equally important. The REVW instruction is similar but allows each rule to have a weighting factor which is stored in a separate parallel data structure in the knowledge base. REV and REVW 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. The following is an example of a typical rule.

If temperature is warm and pressure is high then heat is (should be) off.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

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. In order 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 STAR12 CPU, 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 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 cleared, 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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

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% while, at the same time, another indicates that the system output should be low with a degree of truth of 25%. The defuzzification step resolves these ambiguities.

C.3.3 Defuzzification (WAV)

The final step in the fuzzy logic program combines the raw fuzzy outputs into a composite system output. Instead of the trapezoidal shapes used for inputs, singletons are typically used 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
F_i are the fuzzy outputs from RAM

For a common fuzzy logic program, 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.

C.4 Example Inference Kernel

Figure C-3 is a complete fuzzy inference kernel written in 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.

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.

```
*
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_LOOP1MEM                 ;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 C-3 Fuzzy Inference Engine

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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

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 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 STAR12 CPU. 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.

C.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.

C.5.1 Membership Function Definitions

Figure C-4 shows how a normal membership function is specified. 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.

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. **C.5.2 Abnormal Membership Function Definitions** discusses these cases. The following rules apply to normal membership functions.

- $00 ≤ point1 < $FF
- $00 < point2 ≤ $FF

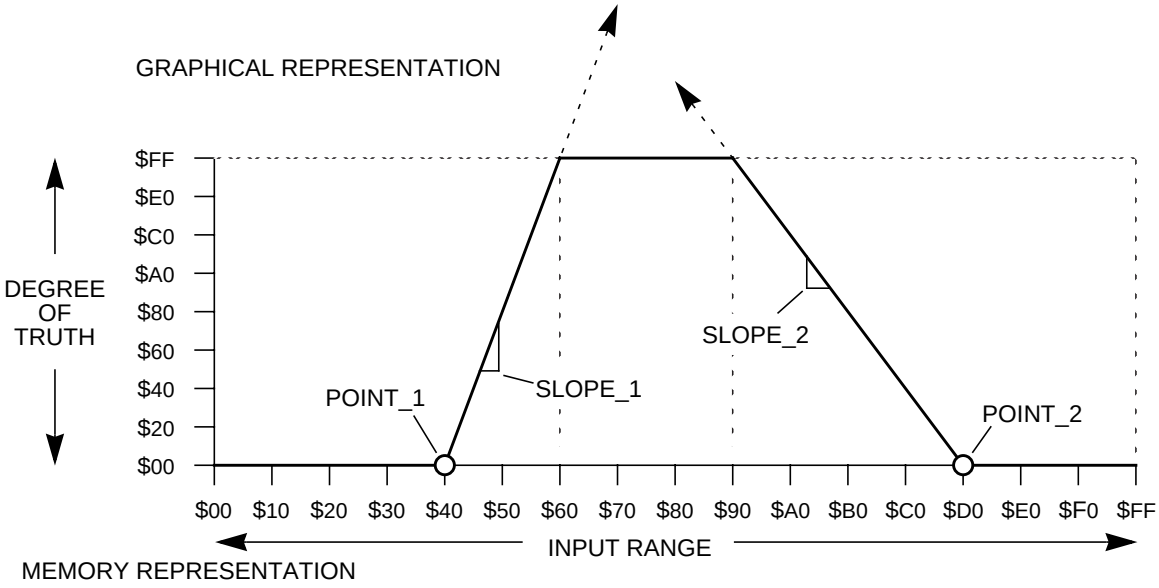
- point1 < point2
- The sloping sides of the trapezoid meet at or above $FF

Each system input such as temperature has several labels such as cold, cool, normal, warm, and hot. Each label of each system input must have a membership function to describe its meaning in an unambiguous numerical way. Typically, there are three to seven labels per system input, but there is no practical restriction on this number as far as the fuzzification step is concerned.

C.5.2 Abnormal Membership Function Definitions

In the STAR12 CPU, 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.

Although a good fuzzy development tool does 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 STAR12 CPU treats these cases can be helpful for debugging.



MEMORY REPRESENTATION

ADDR	$40	X-POSITION OF POINT_1
ADDR+1	$D0	X-POSITION OF POINT_2
ADDR+2	$08	SLOPE_1 ($FF/(X-POS OF SATURATION – POINT_1))
ADDR+3	$04	SLOPE_2 ($FF/(POINT_2 – X-POS OF SATURATION))

Figure C-4 Defining a Normal Membership Function

A close examination of the MEM instruction algorithm shows how such membership functions are evaluated. **Figure C-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 **Appendix B Instruction Set and**

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Commands for details). The upper portion of the box includes information about bus activity, if any, during this cycle. 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. In cycle 3, for example, two 8-bit subtractions take place and a flag is set based on the results.

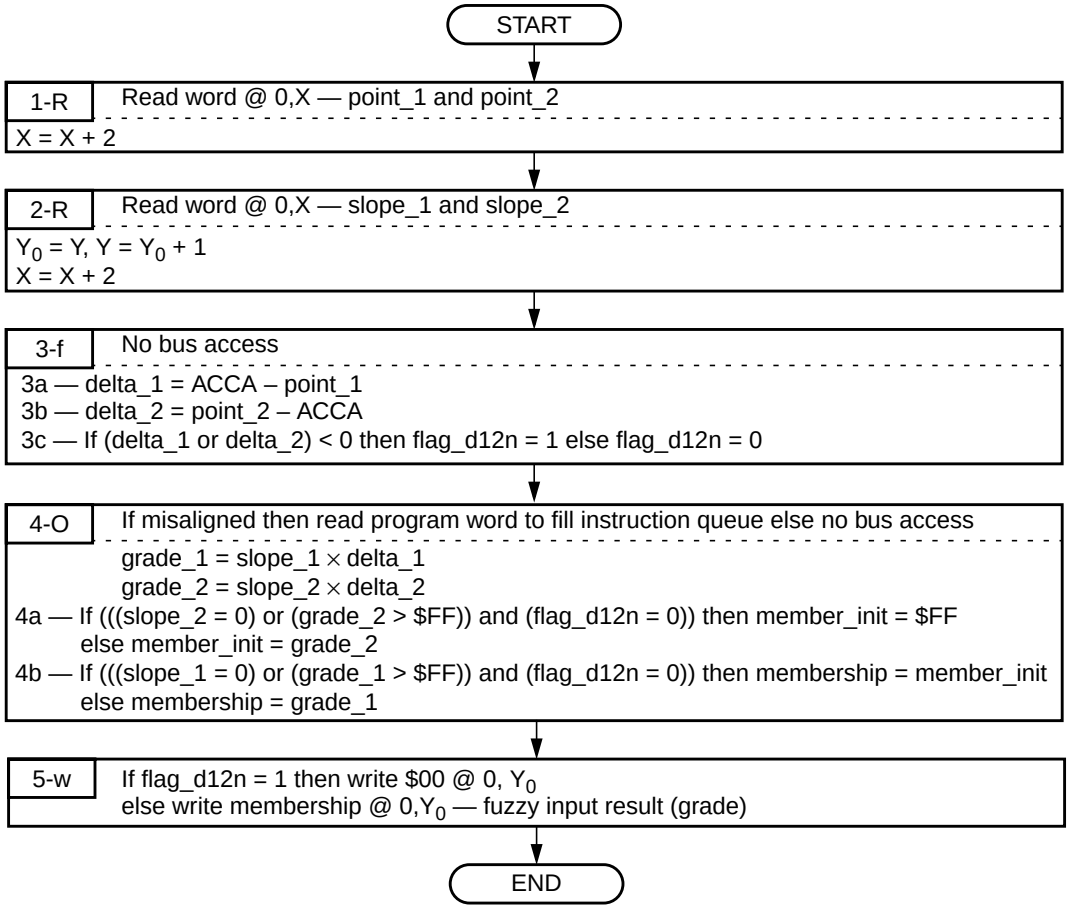


Figure C-5 MEM Instruction Flow Diagram

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 is negative, and the flag equals 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.

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 equals grade_1. If the input is outside the trapezoid, flag_d12n is one and grade_1 and grade_2 would have been forced to $00 in cycle 3. The else condition of 4b sets the result to $00.

The following special cases represent abnormal membership function definitions. The explanations describe how the specific algorithm in the STAR12 CPU 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.

C.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.

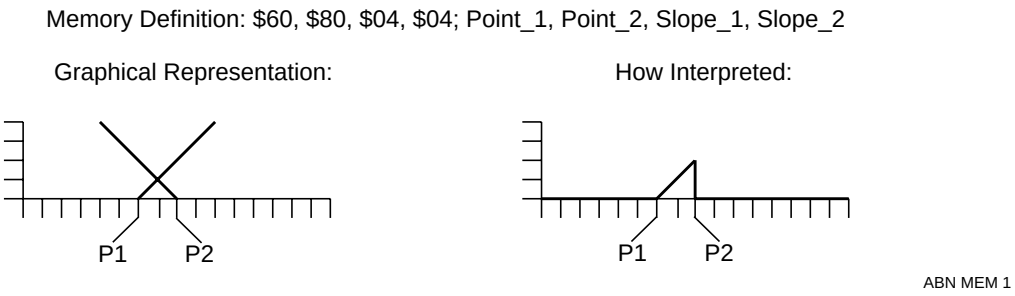


Figure C-6 Abnormal Membership Function Case 1

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.

C.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 (grade_1 > $FF) term of 4b kicks in, making the expression true so grade equals grade (no overwrite). The result from here to point_2 becomes controlled by the else part of 4a (grade = grade_2), and the result follows the right sloping side.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

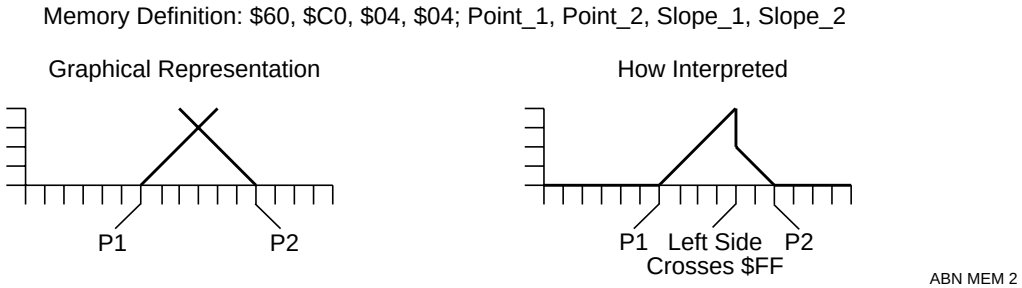


Figure C-7 Abnormal Membership Function Case 2

C.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.

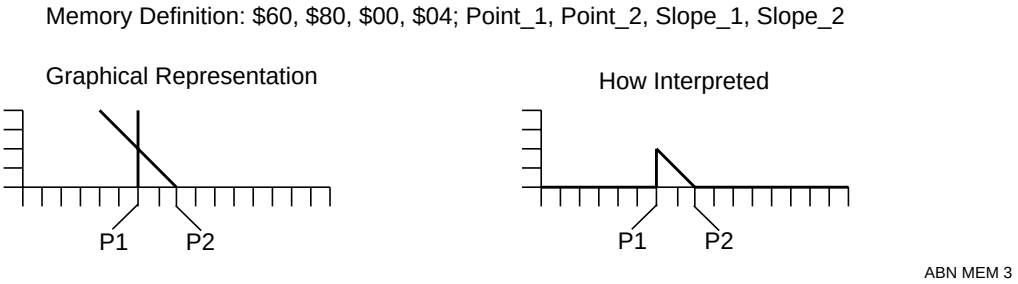


Figure C-8 Abnormal Membership Function Case 3

C.6 REV, REVW Instruction Details

This section provides a more detailed explanation of the rule evaluation instructions, REV and REVW. The data structures that 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.

C.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.

C.6.1.1 Initialization Prior to Executing REV

Some CPU registers and memory locations need to be initialized before executing the REV instruction. X and Y index registers are index pointers to the rule list and the fuzzy inputs and outputs. The A accumulator holds intermediate calculation results and needs to be initially set to $FF. The V bit is an instruction status indicator showing whether antecedents or consequents are being processed. Initially, the V bit is cleared to indicate antecedents are being processed. The fuzzy outputs in working RAM locations need to be cleared to $00. Improper initialization produces erroneous results.

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 points 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 execution of REV begins, 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 holds the truth value for the last rule in the rule list. The V 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.

C.6.1.2 Interrupt Details

The REV instruction includes a three-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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is serviced. 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.

When a REV instruction is interrupted, the stacked value of the program counter, PC, 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 code register, indicate the current status of the interrupted REV instruction, the rule evaluation operation resumes where it was interrupted.

C.6.1.3 Cycle-by-Cycle REV Details

The central element of the REV instruction is a three-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 servicing the requested interrupt.

Figure C-9 is a REV instruction flow diagram. Each box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the upper left corner of each box are execution cycle codes (refer to **Appendix B Instruction Set and Commands** for details).

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.

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. The 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, the digit after the decimal point, changes to a new value.

In cycle 1.0, the CPU does an optional program word access to replace the $18 prebyte of the REV instruction. Notice that cycle 7.0 is also an O cycle. One of these cycles is a program word fetch, while the other is a free cycle in which the CPU does not access the bus. Although the $18 page prebyte is part of the REV instruction, the CPU treats it as a separate single-cycle instruction.

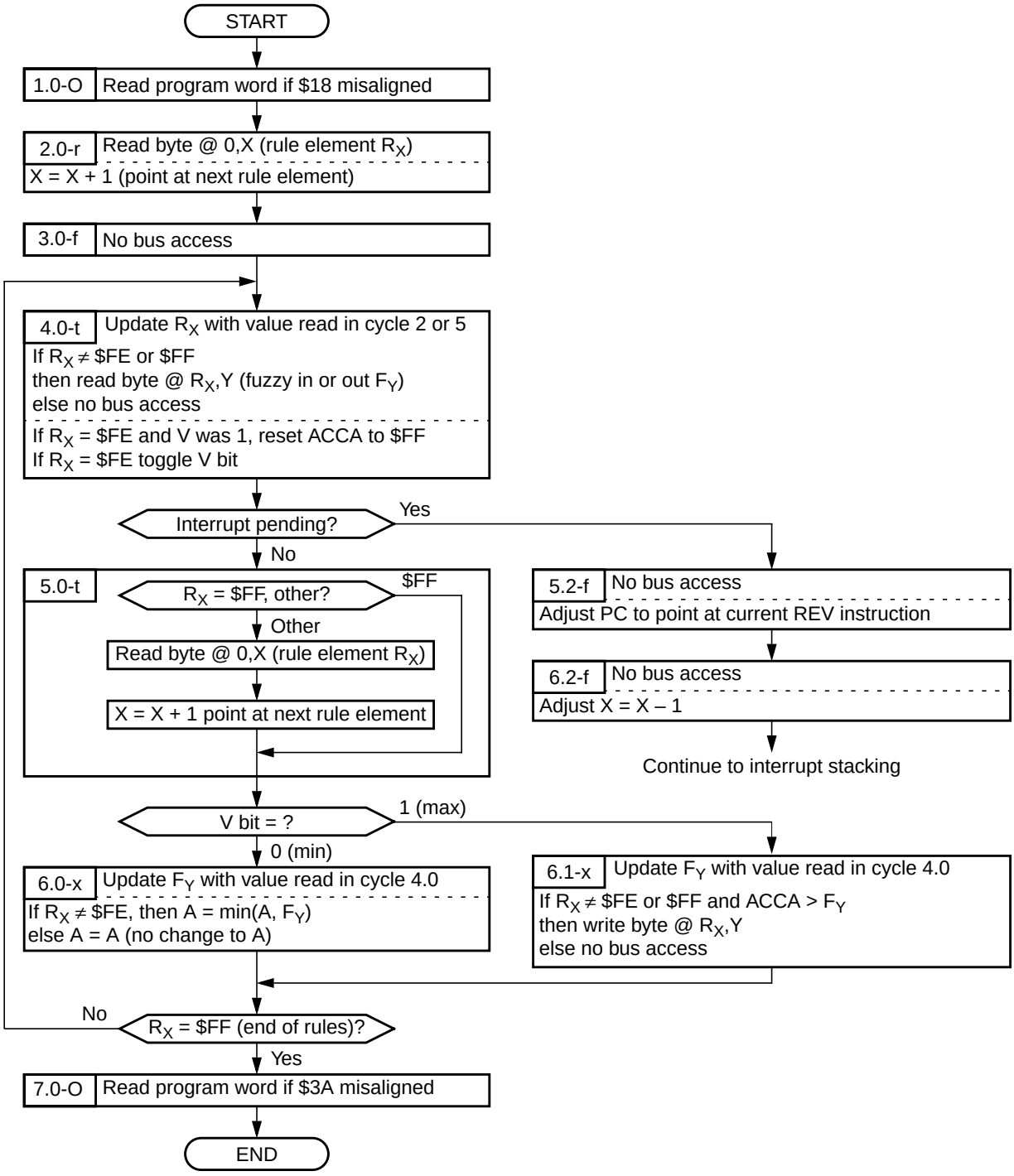


Figure C-9 REV Instruction Flow Diagram

Rule evaluation begins at cycle 2.0 with a byte read of the first element in the rule list. Usually this is 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 satisfies the required delay between a read and when data is valid to the CPU. Some internal

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

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 on whether rule antecedents or consequents are being processed, the loop consists of cycles 4.0, 5.0, and 6.0 or 4.0, 5.0, and 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, in which 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 is used during the next cycle 6.0 or 6.1. Besides being the offset from Y for this read, the previously read R_X can be a separator character, $FE. If R_X is $FE and the V bit was one, this indicates a switch from processing consequents of one rule to processing 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 R_X is $FE, the V bit toggles 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 is not used until cycle 4.0 of the next pass through the loop.

Between cycle 5.0 and 6.x, the V bit determines 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 min operation compares the current value in the A accumulator to the fuzzy input that was read in the previous cycle 4.0 and puts the lower value in the A accumulator. If R_X is $FE, this is the transition between rule antecedents and rule consequents, and the min operation is skipped although the cycle is still used. Cycle 6.0/6.1 is an x cycle because it could be a byte write or a free cycle.

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 service. Cycles 5.2 and 6.2 adjust the PC and X index register so the REV operation can recover after the interrupt. In cycle 5.2, PC is decremented so that it points to the currently-running REV instruction. After the interrupt, rule evaluation resumes, but the stacked values for the index registers, accumulator A, and CCR cause the operation to pick up where it left off. In cycle 6.2, the X index register is decremented by one because the last rule byte needs to be refetched when the REV instruction resumes.

After cycle 6.2, the REV instruction is finished, and execution continues with the normal interrupt processing flow.

C.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 depends 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 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 marker separating antecedents from consequents is 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 are the addresses of the reset vector, there is never a fuzzy input or output at either of these locations.

C.6.2.1 Initialization Prior to Executing REVW

Some CPU registers and memory locations need to be initialized before executing the REVW instruction. X and Y index registers are index pointers to the rule list and the list of rule weights. The A accumulator holds intermediate calculation results and needs to be initialized to $FF. The V bit is an instruction status indicator that shows whether antecedents or consequents are being processed. Initially the V bit is cleared to indicate antecedent processing. The C bit enables (1) or disables (0) rule weighting. The fuzzy outputs in working RAM locations need to be cleared to $00. Improper initialization produces erroneous results.

Initialize the X index register with 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 points at the next address past the $FFFF separator word that marks the end of the rule list.

Initialize the Y index register with 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 holds 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 the C bit is one, rule weights are enabled, and the rule truth value is multiplied by the rule weight just before consequent processing starts. During consequent processing, A holds the weighted or unweighted 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 executing REVW, initialize A with $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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Both the C and V bits must be initialized before 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, the fuzzy output is compared to the weighted 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 are evaluated, the fuzzy output contains the truth value for the most-true rule referencing that fuzzy output.

After REVW finishes, accumulator A holds the weighted truth value for the last rule in the rule list. The V bit should be one because the last element before the $FFFF end marker should be a rule consequent. If V is zero after executing REVW, it indicates the rule list is structured incorrectly.

C.6.2.2 Interrupt Details

The REVW instruction includes a three-cycle processing loop for each word in the rule list. This loop expands to five cycles between antecedents and consequents to allow time for multiplication by the rule weight. Within this loop is a check to see if any qualified interrupt request is pending. If an interrupt is detected, the CPU registers are stacked and the interrupt request is serviced. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REVW instruction resumes as if it had not been interrupted.

When a REVW instruction is interrupted, the stacked value of the program counter, PC, 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 and V bits in the condition code register, indicate the status of the interrupted REVW instruction, the rule evaluation operation resumes where it was interrupted.

C.6.2.3 Cycle-by-Cycle REVW Details

The central element of the REVW instruction is a three-cycle loop that is executed once for each word in the rule list. This loop takes five cycles in the special-case pass in which weights are enabled (C = 1) and the $FFFE separator word is read between the rule antecedents and the rule consequents. 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 request comes, there is a special small sequence to save CPU status on the stack before the interrupt is serviced.

Figure C-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 box correspond to the execution cycle codes (refer to **Appendix B Instruction Set and Commands** for details).

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 in cycle 4.0. The X index register is incremented by two to point to 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 toggles at every $FFFE encountered and indicates which transition is taking place. If V is zero, a change from antecedents to consequents is taking place, and it is time to apply weighting if weighting is enabled. 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 is used in cycle 4.0 of the next loop pass. The 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 are the weighted result. By using weight + 1, the result can range from $0.0 \times A$ to $1.0 \times A$. After 8.2, flow continues to the next loop pass at cycle 4.0.

At cycle 4.0, if R_X is $FFFE and V was one, a change from consequents to antecedents of a new rule is taking place, so accumulator A must be reinitialized to $FF. During processing of rule antecedents, A is updated with the smaller of A and the current fuzzy input (cycle 6.0). Cycle 5.0 usually reads the next rule word and updates the pointer in X. This read is skipped if the current R_X is the end of rules mark, $FFFF. If this is a weight multiply pass, the read is delayed until cycle 7.2. During processing of consequents, cycle 6.1 optionally updates a fuzzy output if the value in accumulator A is larger.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

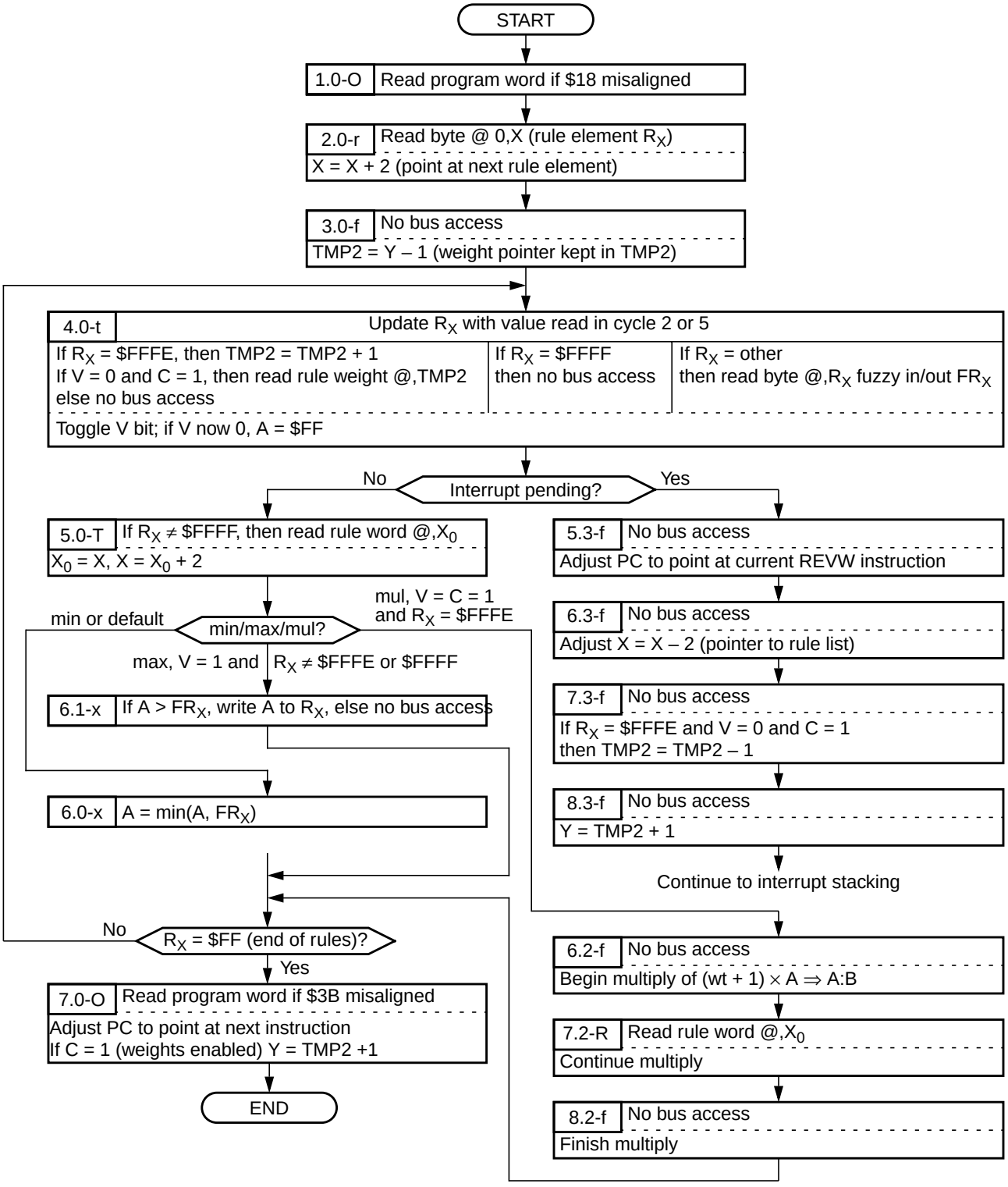


Figure C-10 REVW Instruction Flow Diagram

After all rules are processed, cycle 7.0 updates the PC to point at the next instruction. If weights are enabled, Y is updated to point at the location that immediately follows the last rule weight.

C.7 WAV Instruction Details

The WAV instruction performs weighted average calculations used in defuzzification. The pseudoinstruction wavr resumes 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 (8-bit values)
F_i are the fuzzy outputs from RAM (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 which equals the number of 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 CPU handles this difficulty with a pseudo-instruction, wavr, which is specifically intended to resume an interrupted weighted average calculation. Refer to **C.7.3 Cycle-by-Cycle Details for WAV and wavr** for details.

C.7.1 Initialization Prior to Executing WAV

Before executing the WAV instruction, index registers X and Y and accumulator B must be initialized. 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. Accumulator B contains the iteration count n and must be initialized with the number of labels for this system output.

C.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 to see whether a qualified interrupt request is pending. If an interrupt request is detected, the CPU registers and the current values of the internal temporary registers for the 24-bit and 16-bit sums are stacked, and the interrupt is serviced.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

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 pseudoinstruction. The wavr pseudoinstruction causes the CPU to execute a special WAV resumption sequence. The wavr recovery sequence adjusts the PC so that it looks as it did during execution of the original WAV instruction, then jumps back into the WAV processing loop. If another interrupt request 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.

C.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 resumes the weighted average operation after a WAV instruction has been interrupted. This recovery operation is called the wavr pseudoinstruction.

Figure C-11 is a flow diagram of the WAV instruction including the wavr pseudoinstruction. Each 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 are execution cycle codes (refer to **Appendix B Instruction Set and Commands** for details).

In terms of cycle-by-cycle bus activity, the $18 page select prebyte is treated as a one-byte instruction. In cycle 1.0 of the WAV instruction, one word of program information is 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.

Cycle 2.0 clears three internal 16-bit temporary registers 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. Keeping these sums inside the CPU reduces bus accesses and optimizes the WAV operation for high speed.

Cycles 3.0 through 9.0 form the seven-cycle main loop for WAV. The value in the 8-bit B accumulator counts the number of loop iterations. B is decremented at the top of the loop in cycle 3.0, and the test for zero is located at the bottom of the loop after cycle 9.0. Cycles 4.0 and 5.0 fetch the 8-bit operands for one iteration of the loop. The X and Y index registers are used to access these operands. The index registers are incremented as the operands are fetched. Cycle 6.0 accumulates the current fuzzy output into TMP3. Cycles 7.0 and 8.0 perform the eight-by-eight multiply of F_i times S_i. TMP1:TMP2 accumulates the product during cycles 8.0 and 9.0. Even though the sum-of-products does not exceed 24 bits, the sum is maintained in the 32-bit combined TMP1:TMP2 register because it is easier to use existing 16-bit operations than to create a new smaller operation to handle the high bits of this sum.

Since the weighted average operation could be quite long, it is made to be interruptible. The usual longest latency path is from very early in cycle 6.0 through cycle 9.0 to the top of the loop in cycle 3.0, and through cycle 5.0 to the interrupt check. The three-cycle (6.1 through 8.1) exit sequence gives this latency path a total of 10 cycles.

If the WAV instruction is interrupted, the internal temporary registers TMP3, TMP2, and TMP1 need to be stored on the stack so that the operation can be resumed. Since the WAV instruction includes initialization in cycle 2.0, the recovery path after an interrupt needs to be different. The wavr pseudoinstruction has the same opcode as WAV, but it is on the first page of the opcode map so it doesn't have the $18 page 2 prebyte that WAV has. When WAV is interrupted, the PC is adjusted to point at the second byte of the WAV object code, so that it is interpreted as the wavr pseudoinstruction 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

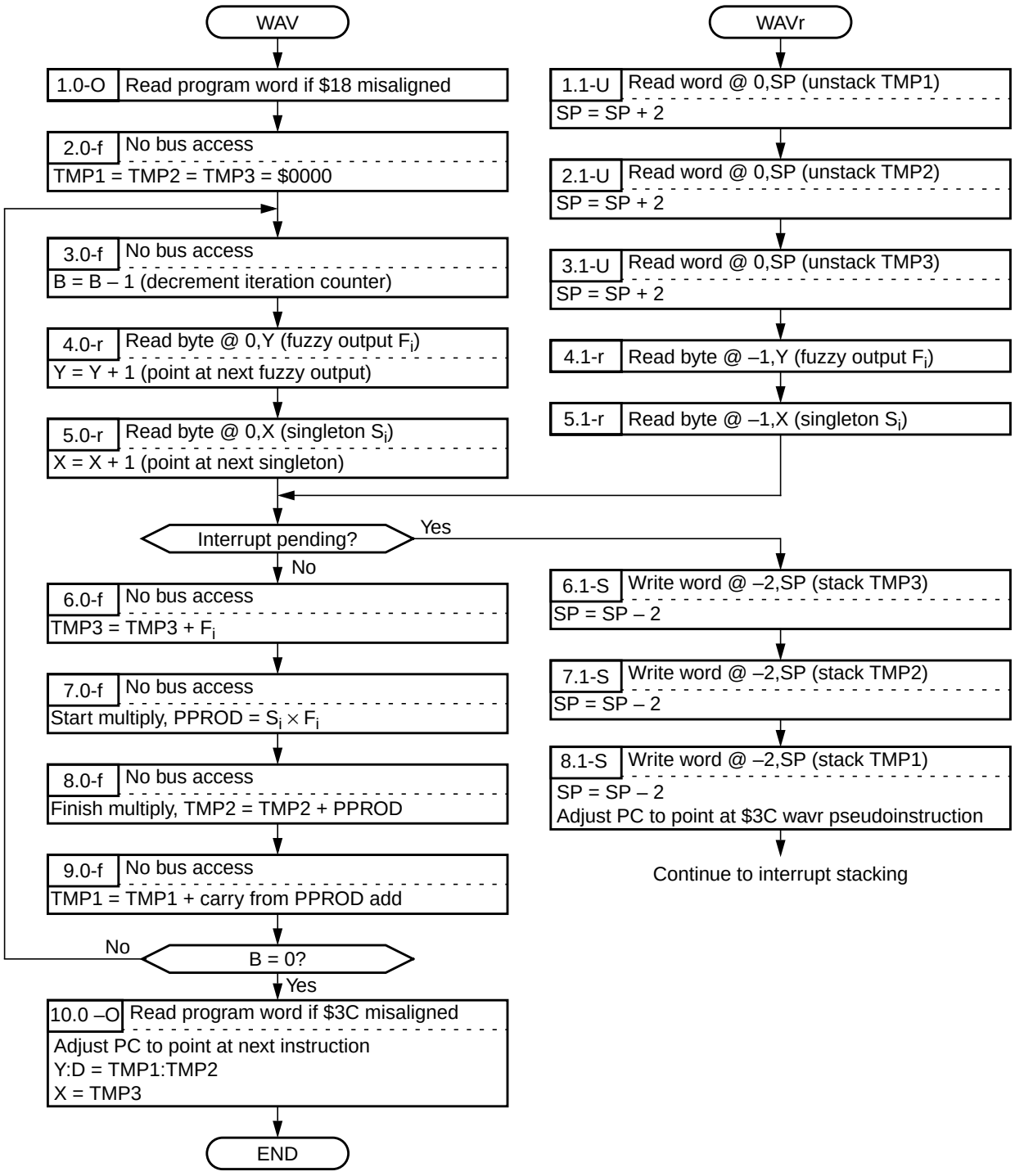


Figure C-11 WAV and wavr Instruction Flow Diagram

The resume sequence includes recovery of the temporary registers from the stack (1.1 through 5.0), and reads to get the operands for the current iteration. The normal WAV flow is then rejoined at cycle 6.0.

Upon normal completion of the instruction (cycle 10.0), the PC is adjusted so it points to the next instruction. The results transfer from the TMP registers into CPU registers in such a way that the EDIV

instruction can divide the sum-of-products by the sum-of-weights. TMP1:TMP2 transfers into Y:D and TMP3 transfers into X.

C.8 Custom Fuzzy Logic Programming

The basic fuzzy logic inference techniques described above 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 support only variations of MIN-MAX rule evaluation. Other methods have been discussed in fuzzy logic literature. The weighted average of singletons is not the only defuzzification technique. The STAR12 CPU has several instructions and addressing modes that can be helpful in developing custom fuzzy logic systems.

C.8.1 Fuzzification Variations

The MEM instruction supports trapezoidal and several other membership functions, including functions with vertical (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 trapezoids butted against the right, left, or neither end of the x-axis. Many other membership function shapes are possible with sufficient memory space and processing bandwidth.

Tabular membership functions offer total flexibility in shape and very fast evaluation time. However, tables take as many as 256 bytes of memory space per system input label. This makes them impractical for most microcontroller-based fuzzy systems. The STAR12 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. Indexed addressing identifies the effective address of the data point at the beginning of the line segment. The data value for the end point of the line segment is the next consecutive memory location. The data values are bytes for TBL and words for ETBL. In both cases, the B accumulator contains the ratio

$$\frac{\text{x-distance from beginning of line segment to lookup point}}{\text{x-distance from beginning to end of line segment}}$$

The value in 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 identifies 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 the x input to the table. The upper eight bits in A are used as a coarse lookup to find the line segment of interest, and the lower eight bits in B are used to interpolate within this line segment.

In the program sequence:

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

```
        LDX             #TBL_START
        LDD             DATA_IN
        TBL             A,X
```

The notation A,X causes the TBL instruction to use the A^{th} line segment in the table. The low 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 that 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 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, in which all values have 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 C-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)
                    BLS     FIND_LOOP           ;loop if XL .le. Xn
* on fall thru, XB@-2,Y YB@-1,Y XE@0,Y and YE@1,Y
                    TFR     D,X                 ;save XL in high half of X
                    CLRA                        ;zero upper half of D
                    LDAB    0,Y                 ;D = 0:XE
                    SUBB    -2,Y                ;D = 0:(XE-XB)
                    EXG     D,X                 ;X = (XE-XB).. D = XL:junk
                    SUBA    -2,Y                ;A = (XL-XB)
                    EXG     A,D                 ;D = 0:(XL-XB), uses trick of EXG
                    FDIV                        ;X reg = (XL-XB)/(XE-XB)
                    EXG     D,X                 ;move fractional result to A:B
                    EXG     A,B                 ;byte swap - need result in B
                    TSTA                        ;check for rounding
                    BPL     NO_ROUND
                    INCB                        ;round B up by 1
NO_ROUND            LDAA    1,Y                 ;YE
                    PSHA                        ;put on stack for TBL later
                    LDAA    -1,Y                ;YB
                    PSHA                        ;now YB@0,SP and YE@1,SP
                    TBL     2,SP+               ;interpolate and deallocate
                                                ;stack temps
```

Figure C-12 Endpoint Table Handling

The basic idea is to find the segment of interest, temporarily build a one-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 that 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 nonmembership to membership as the system input value approaches the central range of the membership function. Let us examine the human problem of stopping a car at an intersection. We might use rules like “If intersection is close and speed is fast, apply brakes.” The meaning of the labels “close” and “fast”, reflected in membership function shape and position, is different for a teenager than it is 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.

C.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 STAR12 CPU 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 STAR12 CPU has min and max instructions for 8- or 16-bit operands, with one operand in an accumulator and the other in a referenced memory location. There are separate variations that replace the accumulator or the memory location with the result. While processing rule antecedents in a fuzzy inference program, a reference value must be compared to each of the referenced fuzzy inputs, and the smallest input must end up in an accumulator. The instruction:

```
EMIND     2,X+      ;process one rule antecedent
```

automates the central operations needed to process rule antecedents. The E stands for extended, so this instruction compares 16-bit operands. The D at the end of the mnemonic stands for the D accumulator, which is both the first operand for the comparison and the destination of the result. The 2,X+ is an indexed addressing specification that says X points to the second operand for the comparison.

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 replaces the referenced memory operand. Again, indexed addressing is used. These two instructions can form the working part of a 16-bit resolution fuzzy inference routine.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

There are many other methods of performing inference, but the min-max method is most widely used. Since the STAR12 is a general-purpose microcontroller, the programmer has complete freedom to program any algorithm desired. A custom algorithm would typically take more code space and execution time than a routine that used the built-in REV or REVW instructions.

C.8.3 Defuzzification Variations

There are two main areas where other STAR12 instructions can help with custom defuzzification routines. The first case is working with operands with 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 automate the multiply and accumulate function. The STAR12 CPU has extended math capabilities, including 32-bit by 16-bit divide instructions and the EMACS instruction which uses 16-bit input operands and accumulates the sum to a 32-bit memory location.

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 over 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 STAR 12 CPU is well suited for more computationally challenging algorithms than weighted average. A 32-bit by 16-bit divide instruction takes eleven or twelve 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 twelve 8-MHz cycles per iteration, including accessing all operands from memory and storing the result to memory.

Appendix D M68HC11 to STAR12 Upgrade

D.1 General

This appendix discusses aspects of upgrading system software from one based upon the Motorola 68HC11 to one using the STAR12 CPU. In general, the STAR12 is a proper superset of the M68HC11 instruction set (as was the HC12 CPU prior to the STAR12).

D.2 Source Code Compatibility

Every M68HC11 instruction mnemonic and source code statement can be assembled directly with a STAR12 assembler with no modifications.

The STAR12 supports all M68HC11 addressing modes and includes several new variations of indexed addressing. STAR12 instructions affect condition code bits in the same way as M68HC11 instructions.

STAR12 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 STAR12 assembler automatically translates a few M68HC11 instruction mnemonics into functionally equivalent STAR12 instructions. For example, the STAR12 does not have an increment stack pointer (INS) instruction, so the INS mnemonic is translated to LEAS 1,S. The STAR12 does provide single-byte DEX, DEY, INX, and INY instructions because the LEAX and LEAY instructions do not affect the condition codes, while the M68HC11 instructions update the Z bit according to the result of the decrement or increment.

Table D-1 shows M68HC11 instruction mnemonics that are automatically translated into equivalent STAR12 instructions. This translation is performed by the assembler so there is no need to modify an old M68HC11 program in order to assemble it for the STAR12. In fact, the M68HC11 mnemonics can be used in new STAR12 programs.

Table D-1 Translated M68HC11 Mnemonics

M68HC11 Mnemonic	Equivalent STAR12 Instruction	Comments
ABX ABY	LEAX B,X LEAY B,Y	Since STAR12 has accumulator offset indexing, ABX and ABY are rarely used in new STAR12 programs. ABX was one byte on M68HC11 but ABY was two bytes. The LEA substitutes are two bytes.
CLC CLI CLV SEC SEI SEV	ANDCC #$FE ANDCC #$EF ANDCC #$FD ORCC #$01 ORCC #$10 ORCC #$02	ANDCC and ORCC now allow more control over the CCR, including the ability to set or clear multiple bits in a single instruction. These instructions took one byte each on M68HC11 while the ANDCC and ORCC equivalents take two bytes each.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

Table D-1 Translated M68HC11 Mnemonics

M68HC11 Mnemonic	Equivalent STAR12 Instruction	Comments
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 STAR12 duplicate the function of DES and INS. These instructions were one byte on M68HC11 and two bytes on STAR12.
TAP TPA TSX TSY TXS TYS XGDX XGDY	TFR A,CCR TFR CCR,A TFR S,X TFR S,Y TFR X,S TFR Y,S EXG D,X EXG D,Y	The M68HC11 had a small collection of specific transfer and exchange instructions. STAR12 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 STAR12 transfer/exchange costs one extra byte compared to the M68HC11. The substitute instructions execute in one cycle rather than two.

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

The STAR12 has a two-page opcode map, rather than the four-page M68HC11 map. This is largely due to redesign of the indexed addressing modes. Most of pages 2, 3, and 4 of the M68HC11 opcode map are required because Y-indexed instructions use different opcodes than X-indexed instructions. Approximately two-thirds of the M68HC11 page 1 opcodes are unchanged in STAR12, and some M68HC11 opcodes have been moved to page 1 of the STAR12 opcode map. Object code for each of the moved instructions is one byte smaller than object code for the equivalent M68HC11 instruction. **Table D-2** shows instructions that assemble to one byte less object code on the STAR12.

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 8K byte M68HC11 assembly code program, was reassembled for the STAR12. 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.

Table D-2 Instructions with Smaller Object Code

Instruction	Comments
DEY INY	Page 2 opcodes in M68HC11 but page 1 in STAR12.
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 STAR12 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.

Table D-2 Instructions with Smaller Object Code

Instruction	Comments
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, STAR12 and M68HC11 object code are the same size.

The relative size of code for M68HC11 vs. code for STAR12 has also been tested by rewriting several smaller programs from scratch. In these cases, the STAR12 code is typically about 30% smaller. These savings are mostly due to improved indexed addressing. A C program compiled for the STAR12 is about 30% smaller than the same program compiled for the M68HC11. The savings are largely due to better indexing.

D.3 Programmer's Model and Stacking

The STAR12 programming model and stacking order are identical to those of the M68HC11.

D.4 True 16-Bit Architecture

The M68HC11 is a direct descendant of the M6800, one of the first microprocessors, which was introduced in 1974. The M6800 was strictly an 8-bit machine, with 8-bit data buses and 8-bit instructions. As 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 STAR12 is a true 16-bit implementation, but it retains the ability to work with the mostly 8-bit M68HC11 instruction set. The larger ALU of the STAR12 is used to calculate 16-bit pointers and to speed up math operations.

D.4.1 Bus Structures

The STAR12 is a 16-bit processor with 16-bit data paths. Typical STAR12 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. STAR12 based systems include an on-chip block in the Core 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 Core 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.

D.4.2 Instruction Queue

The CPU has a three-word instruction queue for storing 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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

instruction queue, beginning with an aligned word read at the natural boundary of the misaligned reset vector. The instruction queue logic starts execution with the opcode in the low half of this word.

The instruction queue makes three bytes of program information (starting with the instruction opcode) directly available to the CPU at the beginning of every instruction. As each instruction executes, it performs enough additional program fetches to refill the space it took up in the queue. Alignment information is maintained by 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 sorts out the information in the queue to present the opcode and additional bytes of information as CPU inputs. A control algorithm determines whether the opcode is in the even or odd half of the word at the head of the queue. The execution sequence for all instructions is independent of the alignment of the instruction.

The only situation in which 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. Optional 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 optional cycle to fetch an extra word of object code. If the queue is aligned at the start of an instruction with an odd byte count, the last byte of object code shares a queue word with the opcode of the next instruction. Since this word holds part of the next instruction, the queue cannot advance after the odd byte executes, or the first byte of the next instruction would be lost. In this case, the optional 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 optional 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 optional cycle takes one bus cycle when it appears as a free cycle, but it takes two bus cycles when used to perform a program fetch. In this case, the on-chip integration module freezes the CPU clocks long enough to perform the cycle as two smaller accesses. The CPU handles only 16-bit data, and is not aware that the 16-bit program access is split into two 8-bit accesses.

In order to allow development systems to track events in the STAR12 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.

D.4.3 Stack Function

Both the M68HC11 and the STAR12 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 STAR12 systems 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

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 STAR12 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 STAR12 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. STAR12 TSX and TXS instructions are ordinary transfers — the STAR12 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 STAR12 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 STAR12, 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 STAR12 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 STAR12, change either the initial LDS #$xxxx, or the absolute extended address used to read the stack.

D.5 Improved Indexing

The STAR12 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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

at the address of the operand for the instruction. A load accumulator instruction assembles into two bytes of object code, the opcode and a one-byte offset. Using Y as the reference, the same instruction assembles into three bytes (a page prebyte, the opcode, and a one-byte offset.) Analysis of M68HC11 source code indicates that the offset is most frequently zero and very seldom greater than four.

The STAR12 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 autoinc/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.

D.5.1 Constant Offset Indexing

The STAR12 offers three variations of constant offset indexing in order to optimize the efficiency of object code generation.

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

The 5-bit constant offset variation covers the most frequent indexing requirements by including the offset in the postbyte. This reduces a load accumulator indexed instruction to two bytes of object code, and matches the object code size of the smallest M68HC11 indexed instructions, which can only use X as the index register. The STAR12 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 64K byte address space. Since the address consists of 16 bits, the 16-bit offset can be regarded as a signed (–32,768 to +32767) or unsigned (0 to 65,535) value. In 16-bit constant offset mode, the offset is supplied in two extension bytes after the opcode and postbyte.

D.5.2 Autoincrement/Autodecrement Indexing

The STAR12 provides greatly enhanced autoincrement and autodecrement modes of indexed addressing. In the STAR12, the index modification may be specified 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 accessed. X, Y, and SP can be used as the index reference, but this mode does not allow PC to be the index reference. Modifying PC 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 1808 1808	LDAA0,Y INY INY	A671	LDAA2,Y+

The M68HC11 object code takes seven bytes, while the STAR12 takes only two bytes to accomplish the same functions. Three bytes of M68HC11 code are due to the page prebyte for each Y-related instruction ($18). STAR12 postincrement indexing capability allows the two INY instructions to be absorbed into the LDAA indexed instruction. The replacement code is not identical to the original three-instruction sequence because the Z bit is affected by the M68HC11 INY instructions, while the Z bit in the STAR12 is determined by the value loaded into A.

D.5.3 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 STAR12 operations.

```
C6  05          LDAB  #$05     [2]
CE  10  00 LOOP LDX   #$1000   [3]        C6  05          LDAB  #$05     [1]
3A              ABX            [3]        CE  10  00      LDX   #$1000   [2]
A6  00          LDAA  0,X      [4]        A6  E5      LOOP LDAA  B,X     [3]

5A              DECB           [2]        04  31  FB      DBNE  B,LOOP   [3]
26  F7          BNE   LOOP     [3]
```

The STAR12 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.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

D.5.4 Indirect Indexing

The STAR12 allows some forms of indexed indirect addressing in which 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 accumulator D indexed-indirect. The indexing register can be X, Y, SP, or PC as in other STAR12 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 to implement pointers. Accumulator D indexed-indirect addressing can implement a runtime-computed GOTO function. Indirect addressing is also useful in high level language compilers. For instance, PC-relative indirect indexing can efficiently implement some C case statements.

D.6 Improved Performance

STAR12 based systems provide a number of performance improvements over the M68HC11. These improvements include cycle count reduction, faster math instruction execution and reduction of code size. Each of these aspects is discussed in the subsections below.

D.6.1 Reduced Cycle Counts

No M68HC11 instruction takes less than two cycles, but the STAR12 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 STAR12 can fetch 16 bits of information at a time, rather than eight bits at a time.

D.6.2 Fast Math

The STAR12 has some of the fastest math ever designed into a Motorola general-purpose Core. Much of the speed is due to an execution unit that can perform several operations simultaneously. **Table D-3** compares the speed of STAR12 and M68HC11 math instructions. The STAR12 requires fewer cycles to perform an operation, and the cycle time is half that of the M68HC11.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Table D-3 Comparison of Math Instruction Speeds

Instruction Mnemonic	Math Operation	M68HC11 1 Cycle = 250 ns	M68HC11 w/Coprocessor 1 Cycle = 250 ns	STAR12 1 Cycle = 125 ns
MUL	$8 \times 8 = 16$ (signed)	10 cycles	—	1 cycle
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	13 cycles

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

D.6.3 Code Size Reduction

STAR12 assembly language programs written from scratch tend to be 30% smaller than equivalent programs written for the M68HC11. This figure has been independently qualified by 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 STAR12 requires about 50 bytes. The STAR12 fuzzy logic instructions replace whole subroutines in the M68HC11 version. Table lookup instructions also greatly reduce code space.

Other STAR12 code space reductions are more subtle. Memory to memory moves are one example. The STAR12 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 STAR12, the LEAS, LEAX, and LEAY instructions

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

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 STAR12 instructions such as MIN and MAX combine the actions of several M68HC11 instructions into a single operation.

D.7 Additional Functions

The STAR12 offers many new functions over that of the M68HC11. These new features are discussed in the subsections below,

D.7.1 New Instructions

The STAR12 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 64K byte address map for STAR12 systems incorporating this feature. **Table D-4** is a summary of these new instructions. Subsequent paragraphs discuss significant enhancements.

Table D-4 New STAR12 Instructions

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 >64K byte 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	Extended table lookup and interpolate; 16-bit entries
EXG	Inherent	Exchange register contents
IBEQ	Relative	Increment and branch if equal to zero; looping primitive

Table D-4 New STAR12 Instructions

Mnemonic	Addressing Modes	Brief Functional Description
IBNE	Relative	Increment and branch if not equal to zero; looping primitive
IDIVS	Inherent	Signed integer divide D/X ⇒ 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 (if 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 MOVW	Combinations of immediate, extended,and indexed	Move byte from one memory location to another Move word from one memory location to another
ORCC	Immediate	OR CCR with mask; replaces SEC, SEI, and SEV
PSHC	Inherent	Push CCR onto stack
PSHD	Inherent	Push double accumulator onto stack
PULC	Inherent	Pull CCR from stack
PULD	Inherent	Pull double accumulator from stack
REV	Special	Fuzzy logic rule evaluation
REVW	Special	Fuzzy logic rule evaluation with weights
RTC	Inherent	Restore program page and return address from stack; used with CALL instruction, allows easy access to extended 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

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

D.7.2 Memory-to-Memory Moves

The STAR12 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. Indexed addressing for move instructions is limited to direct indexing modes that require no extension bytes (9- and 16-bit constant offsets are not allowed). This leaves the 5-bit signed constant offset, accumulator offset, and the autoincrement/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.

D.7.3 Universal Transfer and Exchange

The M68HC11 has only eight transfer instructions and two exchange instructions. The STAR12 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, and 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 STAR12 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.

D.7.4 Loop Construct

The STAR12 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 nonzero 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.

D.7.5 Long Branches

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

D.7.6 Minimum and Maximum Instructions

Control programs often need to restrict data values within upper and lower limits. The STAR12 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 determines the smallest rule input and stores the running result in an accumulator. The MAX operation stores the largest rule truth value in an accumulator or stores the previous fuzzy output value from a RAM location in 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 STAR12 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 STAR12. 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.

D.7.7 Fuzzy Logic Support

The STAR12 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. The STAR12 kernel uses about 50 bytes.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED "CONTROLLED COPY" IN RED

D.7.8 Table Lookup and Interpolation

The STAR12 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.

D.7.9 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 STAR12 allows all bit-manipulation instructions to work with direct, extended or indexed addressing modes.

D.7.10 Push and Pull D and CCR

The STAR12 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.

D.7.11 Compare SP

This instruction was added to the STAR12 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.

D.7.12 Support for Memory Expansion

Bank switching is a common method of expanding memory beyond the 64K byte limit of a CPU with a 64K byte physical 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 STAR12 Core includes a built-in bank switching scheme that eliminates many of the problems associated with external switching logic. The STAR12 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 STAR12 systems 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 systems, the STAR12 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 64K byte address space, thus providing for portable code.

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Core User Guide End Sheet

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

Core User Guide End Sheet

PRINTED VERSIONS ARE UNCONTROLLED EXCEPT WHEN STAMPED **"CONTROLLED COPY"** IN RED

FINAL PAGE OF
550 PAGES