

CPU12/CPU12X

Reference Manual

S12 & S12S

S12X & S12XS & S12XE

Microcontrollers

CPU12/CPU12X

v01.03

03/2008

freescale.com



CPU12/CPU12X

Reference Manual

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

<http://freescale.com>

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Revision Number	Date	Summary of Changes
v01.00	27 Sep. 2006	Initial version of the Generic CPU documentation
v01.01	17 Apr. 2007	Nomenclature update.
v01.02	08 Jan. 2008	Corrected minor errors in Instruction Glossary section.
v01.03	28 Mar. 2008	Corrected ABX/ABY cycle information in Instruction Glossary Section and Appendix A. Corrected H-Flag information for 16-Bit add instructions in Appendix A.



List of Paragraphs

List of Paragraphs5

Table of Contents7

Chapter 1 Introduction.....19

Chapter 2 Overview.....25

Chapter 3 Addressing Modes33

Chapter 4 Instruction Queue.....45

Chapter 5 Instruction Set Overview.....51

Chapter 6 Instruction Glossary75

Chapter 7 Exception Processing.....383

Chapter 8 Debugging Support.....391

Chapter 9 Fuzzy Logic Support.....399

Appendix A Instruction Reference.....427

Appendix B CPU12V0 - CPU12V1463

Appendix C CPU12 - CPU12X465

Appendix D CPU12XV0 - CPU12XV2.....469

Appendix E CPU12XV0 - CPU12XV1.....471

Appendix F High-Level Language Support.....473

Index479



Table of Contents

Chapter 1 Introduction

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

Chapter 2 Overview

2.1	Introduction	25
2.2	Programming Model	25
2.2.1	Accumulators	26
2.2.2	Index Registers	26
2.2.3	Stack Pointer	26
2.2.4	Program Counter	27
2.2.5	Condition Code Register	27
2.2.5.1	U Control Bit	28
2.2.5.2	IPL[2:0]	29
2.2.5.3	S Control Bit	29
2.2.5.4	X Mask Bit	29
2.2.5.5	H Status Bit	29
2.2.5.6	I Mask Bit	30
2.2.5.7	N Status Bit	30
2.2.5.8	Z Status Bit	30
2.2.5.9	V Status Bit	30
2.2.5.10	C Status Bit	30
2.3	Data Types	30
2.4	Memory Organization	31
2.5	Instruction Queue	31

Chapter 3 Addressing Modes

3.1	Introduction	33
-----	--------------------	----

3.2	Mode Summary	33
3.3	Effective Address	33
3.4	Inherent Addressing Mode	34
3.5	Immediate Addressing Mode	34
3.6	Direct Addressing Mode (CPU12)	35
3.7	Direct Addressing Mode (CPU12X)	35
3.8	Extended Addressing Mode	36
3.9	Relative Addressing Mode	36
3.10	Indexed Addressing Modes	37
3.10.1	5-Bit Constant Offset Indexed Addressing	38
3.10.2	9-Bit Constant Offset Indexed Addressing	39
3.10.3	16-Bit Constant Offset Indexed Addressing	39
3.10.4	16-Bit Constant Indirect Indexed Addressing	39
3.10.5	Auto Pre/Post Decrement/Increment Indexed Addressing	40
3.10.6	Accumulator Offset Indexed Addressing	41
3.10.7	Accumulator D Indirect Indexed Addressing	42
3.11	Global Addressing (CPU12X only)	42
3.12	Instructions Using Multiple Modes	42
3.12.1	Move Instructions	42
3.12.2	Bit Manipulation Instructions	43
3.13	Addressing More than 64KB	44

Chapter 4 Instruction Queue

4.1	Introduction	45
4.2	Queue Description	45
4.2.1	CPU12 Family Queue Implementation	45
4.2.2	Data Movement in the Queue	46
4.2.3	No Movement	46
4.2.4	Advance and Load from Data Bus	46
4.2.5	Changes in Execution Flow	46
4.2.6	Exceptions	46
4.2.7	Subroutines	47
4.2.8	Branches	47
4.2.8.1	Short Branches	48
4.2.8.2	Long Branches	48
4.2.8.3	Bit Condition Branches	48
4.2.8.4	Loop Primitives	49
4.2.9	Jumps	49

Chapter 5 Instruction Set Overview

5.1	Introduction	51
5.2	Instruction Set Description	51
5.3	Load and Store Instructions	51
5.4	Transfer and Exchange Instructions	53

5.5	Move Instructions	54
5.6	Addition and Subtraction Instructions	55
5.7	Binary-Coded Decimal Instructions	56
5.8	Decrement and Increment Instructions	56
5.9	Compare and Test Instructions	57
5.10	Boolean Logic Instructions	58
5.11	Clear, Complement, and Negate Instructions	58
5.12	Multiplication and Division Instructions	59
5.13	Bit Test and Manipulation Instructions	60
5.14	Shift and Rotate Instructions	60
5.15	Fuzzy Logic Instructions	61
5.15.1	Fuzzy Logic Membership Instruction (CPU12V0,CPU12XV0 only)	61
5.15.2	Fuzzy Logic Rule Evaluation Instructions (CPU12V0,CPU12XV0 only)	62
5.15.3	Fuzzy Logic Weighted Average Instruction (CPU12V0,CPU12XV0 only)	62
5.16	Maximum and Minimum Instructions	63
5.17	Multiply and Accumulate Instruction	64
5.18	Table Interpolation Instructions	64
5.19	Branch Instructions	65
5.19.1	Short Branch Instructions	65
5.19.2	Long Branch Instructions	66
5.19.3	Bit Condition Branch Instructions	67
5.20	Loop Primitive Instructions	68
5.21	Jump and Subroutine Instructions	68
5.22	Interrupt Instructions	69
5.23	Index Manipulation Instructions	70
5.24	Stacking Instructions	71
5.25	Pointer and Index Calculation Instructions	72
5.26	Condition Code Instructions	73
5.27	Stop and Wait Instructions	73
5.28	Background Mode and Null Operations	74

Chapter 6

Instruction Glossary

6.1	Introduction	75
6.2	Glossary Information	75
6.3	Condition Code Changes	77
6.4	Object Code Notation	77
6.5	Source Forms	78
6.6	Cycle-by-Cycle Execution	80
6.7	Glossary	83
	ABA — Add Accumulator B to Accumulator A	84
	ABX — Add Accumulator B to Index Register X	85
	ABY — Add Accumulator B to Index Register Y	86
	ADCA — Add with Carry to A	87

ADCB — Add with Carry to B	88
ADDA — Add without Carry to A	89
ADDB — Add without Carry to B	90
ADDD — Add Double Accumulator	91
ADDX — Add without Carry to X	92
ADDY — Add without Carry to Y	93
ADED — Add with Carry to D (A:B)	94
ADEX — Add with Carry to X	95
ADEY — Add with Carry to Y	96
ANDA — Logical AND A	97
ANDB — Logical AND B	98
ANDCC — Logical AND CCR with Mask	99
ANDX — Logical AND X	100
ANDY — Logical AND Y	101
ASL — Arithmetic Shift Left Memory (same as LSL)	102
ASLA — Arithmetic Shift Left A (same as LSLA)	103
ASLB — Arithmetic Shift Left B (same as LSLB)	104
ASLD — Arithmetic Shift Left Double Accumulator (same as LSLD)	105
ASLW — Arithmetic Shift Left Memory (16 Bit)	106
ASLX — Arithmetic Shift Left Index Register X	107
ASLY — Arithmetic Shift Left Index Register Y	108
ASR — Arithmetic Shift Right Memory	109
ASRA — Arithmetic Shift Right A	110
ASRB — Arithmetic Shift Right B	111
ASRW — Arithmetic Shift Right Memory (16 Bit)	112
ASRX — Arithmetic Shift Right Index Register X	113
ASRY — Arithmetic Shift Right Index Register Y	114
BCC — Branch if Carry Cleared (Same as BHS)	115
BCLR — Clear Bits in Memory	116
BCS — Branch if Carry Set (Same as BLO)	117
BEQ — Branch if Equal	118
BGE — Branch if Greater than or Equal to Zero	119
BGND — Enter Background Debug Mode	120
BGT — Branch if Greater than Zero	121
BHI — Branch if Higher	122
BHS — Branch if Higher or Same (Same as BCC)	123
BITA — Bit Test A	124
BITB — Bit Test B	125
BITX — Bit Test X	126
BITY — Bit Test Y	127
BLE — Branch if Less Than or Equal to Zero	128
BLO — Branch if Lower (Same as BCS)	129
BLS — Branch if Lower or Same	130
BLT — Branch if Less than Zero	131
BMI — Branch if Minus	132
BNE — Branch if Not Equal to Zero	133
BPL — Branch if Plus	134
BRA — Branch Always	135
BRCLR — Branch if Bits Cleared	136

BRN — Branch Never	137
BRSET — Branch if Bits Set	138
BSET — Set Bit(s) in Memory	139
BSR — Branch to Subroutine	140
BTAS — Bit(s) Test and Set in Memory	141
BVC — Branch if Overflow Cleared	142
BVS — Branch if Overflow Set	143
CALL — Call Subroutine in Expanded Memory	144
CBA — Compare Accumulators	145
CLC — Clear Carry	146
CLI — Clear Interrupt Mask	147
CLR — Clear Memory	148
CLRA — Clear A	149
CLRB — Clear B	150
CLRW — Clear Memory (16 Bit)	151
CLRX — Clear Index Register X	152
CLRY — Clear Index Register Y	153
CLV — Clear Two's Complement Overflow Bit	154
CPA — Compare A	155
CMPB — Compare B	156
COM — Complement Memory	157
COMA — Complement A	158
COMB — Complement B	159
COMW — Complement Memory (16 Bit)	160
COMX — Complement Index Register X	161
COMY — Complement Index Register Y	162
CPD — Compare Double Accumulator	163
CPED — Compare D to Memory with Borrow	164
CPES — Compare SP to Memory with Borrow	165
CPEX — Compare X to Memory with Borrow	166
CPEY — Compare Y to Memory with Borrow	167
CPS — Compare Stack Pointer	168
CPX — Compare Index Register X	169
CPY — Compare Index Register Y	170
DAA — Decimal Adjust A	171
DBEQ — Decrement and Branch if Equal to Zero	172
DBNE — Decrement and Branch if Not Equal to Zero	173
DEC — Decrement Memory	174
DECA — Decrement A	175
DECB — Decrement B	176
DECW — Decrement Memory (16 Bit)	177
DECX — Decrement Index Register X	178
DECY — Decrement Index Register Y	179
DES — Decrement Stack Pointer	180
DEX — Decrement Index Register X	181
DEY — Decrement Index Register Y	182
EDIV — Extended Divide 32-Bit by 16-Bit (Unsigned)	183
EDIVS — Extended Divide 32-Bit by 16-Bit (Signed)	184
EMACS — Extended Multiply and Accumulate 16-Bit by 16-Bit to 32-Bit (Signed)	185

EMAXD — Place Larger of Two Unsigned 16-Bit Values in Accumulator D	186
EMAXM — Place Larger of Two Unsigned 16-Bit Values in Memory	187
EMIND — Place Smaller of Two Unsigned 16-Bit Values in Accumulator D	188
EMINM — Place Smaller of Two Unsigned 16-Bit Values in Memory	189
EMUL — Extended Multiply 16-Bit by 16-Bit (Unsigned)	190
EMULS — Extended Multiply 16-Bit by 16-Bit (Signed)	191
EORA — Exclusive OR A	192
EORB — Exclusive OR B	193
EORX — Exclusive OR X	194
EORY — Exclusive OR Y	195
ETBL — Extended Table Lookup and Interpolate	196
EXG — Exchange Register Contents	197
EXG — Exchange Register Contents (Continued)	198
FDIV — Fractional Divide	199
GLDAA — Load Accumulator A from Global Memory	200
GLDAB — Load Accumulator B from Global Memory	201
GLDD — Load Double Accumulator D (A:B) from Global Memory	202
GLDS — Load Stack Pointer from Global Memory	203
GLDX — Load Stack Index Register X from Global Memory	204
GLDY — Load Stack Index Register Y from Global Memory	205
GSTAA — Store Accumulator A to Global Memory	206
GSTAB — Store Accumulator B to Global Memory	207
GSTD — Store Double Accumulator to Global Memory	208
GSTS — Store Stack Pointer to Global Memory	209
GSTX — Store Index Register X to Global Memory	210
GSTY — Store Index Register Y to Global Memory	211
IBEQ — Increment and Branch if Equal to Zero	212
IBNE — Increment and Branch if Not Equal to Zero	213
IDIV — Integer Divide	214
IDIVS — Integer Divide (Signed)	215
INC — Increment Memory	216
INCA — Increment A	217
INCB — Increment B	218
INCW — Increment Memory (16 Bit)	219
INCX — Increment Index Register X	220
INCY — Increment Index Register Y	221
INS — Increment Stack Pointer	222
INX — Increment Index Register X	223
INY — Increment Index Register Y	224
JMP — Jump	225
JSR — Jump to Subroutine	226
LBCC — Long Branch if Carry Cleared (Same as LBHS)	227
LBCS — Long Branch if Carry Set (Same as LBLO)	228
LBEQ — Long Branch if Equal	229
LBGE — Long Branch if Greater Than or Equal to Zero	230
LBGT — Long Branch if Greater Than Zero	231
LBHI — Long Branch if Higher	232
LBHS — Long Branch if Higher or Same (Same as LBCC)	233
LBLE — Long Branch if Less Than or Equal to Zero	234

LBLO — Long Branch if Lower (Same as LBCS)	235
LBSL — Long Branch if Lower or Same	236
LBLT — Long Branch if Less Than Zero	237
LBMI — Long Branch if Minus	238
LBNE — Long Branch if Not Equal to Zero	239
LBPL — Long Branch if Plus	240
LBRA — Long Branch Always	241
LBRN — Long Branch Never	242
LBVC — Long Branch if Overflow Cleared	243
LBVS — Long Branch if Overflow Set	244
LDAA — Load Accumulator A	245
LDAB — Load Accumulator B	246
LDD — Load Double Accumulator	247
LDS — Load Stack Pointer	248
LDX — Load Index Register X	249
LDY — Load Index Register Y	250
LEAS — Load Stack Pointer with Effective Address	251
LEAX — Load X with Effective Address	252
LEAY — Load Y with Effective Address	253
LSL — Logical Shift Left Memory (Same as ASL)	254
LSLA — Logical Shift Left A (Same as ASLA)	255
LSLB — Logical Shift Left B (Same as ASLB)	256
LSLD — Logical Shift Left Double (Same as ASLD)	257
LSLW — Logical Shift Left Memory (16 Bit, Same as ASLW)	258
LSLX — Logical Shift Left Index Register X (Same as ASLX)	259
LSLY — Logical Shift Left Index Register Y (Same as ASLY)	260
LSR — Logical Shift Right Memory	261
LSRA — Logical Shift Right A	262
LSRB — Logical Shift Right B	263
LSRD — Logical Shift Right Double	264
LSRW — Logical Shift Right Memory (16 Bit)	265
LSRX — Logical Shift Right Index Register X	266
LSRY — Logical Shift Right Index Register Y	267
MAXA — Place Larger of Two Unsigned 8-Bit Values in Accumulator A	268
MAXM — Place Larger of Two Unsigned 8-Bit Values in Memory	269
MEM — Determine Grade of Membership (Fuzzy Logic)	270
MINA — Place Smaller of Two Unsigned 8-Bit Values in Accumulator A	271
MINM — Place Smaller of Two Unsigned 8-Bit Values in Memory	272
MOVB — Immediate-to-Memory Byte Move (8 Bit)	273
MOVB — Memory-to-Memory Byte Move EXT Source (8 Bit)	274
MOVB — Memory-to-Memory Byte Move IDX Source (8 Bit)	275
MOVB — Memory-to-Memory Byte Move IDX1 Source (8 Bit)	276
MOVB — Memory-to-Memory Byte Move IDX2 Source (8 Bit)	277
MOVB — Memory-to-Memory Byte Move [D,IDX] Source (8 Bit)	278
MOVB — Memory-to-Memory Byte Move [IDX2] Source (8 Bit)	279
MOVW — Immediate-to-Memory Word Move (16 Bit)	280
MOVW — Memory-to-Memory Word Move EXT Source (16 Bit)	281
MOVW — Memory-to-Memory Word Move IDX Source (16 Bit)	282
MOVW — Memory-to-Memory Word Move IDX1 Source (16 Bit)	283

MOVW — Memory-to-Memory Word Move IDX2 Source (16 Bit)	284
MOVW — Memory-to-Memory Word Move [D,IDX] Source (16 Bit)	285
MOVW — Memory-to-Memory Word Move [IDX2] Source (16 Bit)	286
MUL — Multiply 8-Bit by 8-Bit (Unsigned)	287
NEG — Negate Memory	288
NEGA — Negate A	289
NEGB — Negate B	290
NEGW — Two's Complement Negate Memory (16 Bit)	291
NEGX — Negate Index Register X	292
NEGY — Negate Index Register Y	293
NOP — Null Operation	294
ORAA — Inclusive OR A	295
ORAB — Inclusive OR B	296
ORCC — Logical OR CCR with Mask	297
ORX — Logic OR X with Memory	298
ORY — Logic OR Y with Memory	299
PSHA — Push A onto Stack	300
PSHB — Push B onto Stack	301
PSHC — Push CCR onto Stack	302
PSHCW — Push CCRW onto Stack	303
PSHD — Push Double Accumulator onto Stack	304
PSHX — Push Index Register X onto Stack	305
PSHY — Push Index Register Y onto Stack	306
PULA — Pull A from Stack	307
PULB — Pull B from Stack	308
PULC — Pull Condition Code Register from Stack	309
PULCW — Pull Condition Code Register from Stack	310
PULD — Pull Double Accumulator from Stack	311
PULX — Pull Index Register X from Stack	312
PULY — Pull Index Register Y from Stack	313
REV — Fuzzy Logic Rule Evaluation	314
REV — Fuzzy Logic Rule Evaluation (Continued)	315
REVV — Fuzzy Logic Rule Evaluation (Weighted)	316
REVV — Fuzzy Logic Rule Evaluation (Weighted) (Continued)	317
ROL — Rotate Left Memory	318
ROLA — Rotate Left A	319
ROLB — Rotate Left B	320
ROLW — Rotate Memory Left through Carry (16 Bit)	321
ROLX — Rotate X Left through Carry	322
ROLY — Rotate Y Left through Carry	323
ROR — Rotate Right Memory	324
RORA — Rotate Right A	325
RORB — Rotate Right B	326
RORW — Rotate Memory Right through Carry (16 Bit)	327
RORX — Rotate X Right through Carry	328
RORY — Rotate Y Right through Carry	329
RTC — Return from Call	330
RTI — Return from Interrupt	331
RTS — Return from Subroutine	332

SBA — Subtract Accumulators	333
SBCA — Subtract with Carry from A	334
SBCB — Subtract with Carry from B	335
SBED — Subtract with Borrow from D	336
SBEX — Subtract with Borrow from X	337
SBEY — Subtract with Borrow from Y	338
SEC — Set Carry	339
SEI — Set Interrupt Mask	340
SEV — Set Two's Complement Overflow Bit	341
SEX — Sign Extend into 16-Bit Register	342
STAA — Store Accumulator A	343
STAB — Store Accumulator B	344
STD — Store Double Accumulator	345
STOP — Stop Processing	346
STS — Store Stack Pointer	348
STX — Store Index Register X	349
STY — Store Index Register Y	350
SUBA — Subtract A	351
SUBB — Subtract B	352
SUBD — Subtract Double Accumulator	353
SUBX — Subtract Memory from X	354
SUBY — Subtract Memory from Y	355
SWI — Software Interrupt	356
SYS — System Call Interrupt	357
TAB — Transfer from Accumulator A to Accumulator B	358
TAP — Transfer from Accumulator A to Condition Code Register	359
TBA — Transfer from Accumulator B to Accumulator A	360
TBEQ — Test and Branch if Equal to Zero	361
TBL — Table Lookup and Interpolate	362
TBNE — Test and Branch if Not Equal to Zero	363
TFR — Transfer Register Content to Another Register	364
TFR — Transfer Register Content to Another Register (Continued)	365
TPA — Transfer from Condition Code Register to Accumulator A	366
TRAP — Unimplemented Opcode Trap	367
TST — Test Memory	368
TSTA — Test A	369
TSTB — Test B	370
TSTW — Test Memory for Zero or Minus (16 Bit)	371
TSTX — Test X for Zero or Minus	372
TSTY — Test Y for Zero or Minus	373
TSX — Transfer from Stack Pointer to Index Register X	374
TSY — Transfer from Stack Pointer to Index Register Y	375
TXS — Transfer from Index Register X to Stack Pointer	376
TYS — Transfer from Index Register Y to Stack Pointer	377
WAI — Wait for Interrupt	378
WAV — Weighted Average	379
XGDX — Exchange Double Accumulator and Index Register X	380
XGDY — Exchange Double Accumulator and Index Register Y	381

Chapter 7 Exception Processing

7.1	Introduction	383
7.2	Types of Exceptions	383
7.3	Exception Priority	384
7.4	Resets	384
7.5	Interrupts	385
7.5.1	X-bit-Maskable Interrupt Request (XIRQ)	385
7.5.2	I-bit-Maskable Interrupt Requests	385
7.5.3	Interrupt Recognition	385
7.5.4	Return-from-Interrupt Instruction (RTI)	386
7.6	Unimplemented Opcode Trap	386
7.7	Software Interrupt Instruction (SWI)	387
7.8	System Call Instruction (SYS) (CPU12XV1 & CPU12XV2)	387
7.9	Exception Processing Flow	387
7.9.1	Vector Fetch	387
7.9.2	Reset Exception Processing	388
7.9.3	Interrupt and Unimplemented Opcode Trap Exception Processing	390

Chapter 8 Debugging Support

8.1	Introduction	391
8.2	External Reconstruction of the Queue	391
8.3	Instruction Queue Status Signals	391
8.3.1	CPU12 Timing Detail	392
8.3.2	CPU12X Timing Detail	393
8.3.3	Null	394
8.3.4	ALD — Advance and Load from Data Bus	394
8.3.5	INT — Interrupt Sequence Start	394
8.3.6	SEV — Start Instruction on Even Address	395
8.3.7	SOD — Start Instruction on Odd Address	395
8.4	Queue Reconstruction (for CPU12)	395
8.4.1	Queue Reconstruction Registers (for CPU12)	396
8.4.1.1	fetch_add Register	396
8.4.1.2	st1_add, st1_dat Registers	396
8.4.1.3	st2_add, st2_dat Registers	396
8.4.1.4	st3_add, st3_dat Registers	396
8.5	Instruction Tagging (CPU12)	396
8.6	Instruction Tagging (CPU12X)	397

Chapter 9 Fuzzy Logic Support

9.1	Introduction	399
9.2	Fuzzy Logic Basics	400
9.2.1	Fuzzification (MEM)	401
9.2.2	Rule Evaluation (REV and REVW)	402

9.2.3	Defuzzification (WAV)	404
9.3	Example Inference Kernel	405
9.4	MEM Instruction Details	406
9.4.1	Membership Function Definitions	406
9.4.2	Abnormal Membership Function Definitions	407
9.4.2.1	Abnormal Membership Function Case 1	409
9.4.2.2	Abnormal Membership Function Case 2	410
9.4.2.3	Abnormal Membership Function Case 3	410
9.5	REV and REVW Instruction Details	410
9.5.1	Unweighted Rule Evaluation (REV)	411
9.5.1.1	Set Up Prior to Executing REV	411
9.5.1.2	Interrupt Details	412
9.5.1.3	Cycle-by-Cycle Details for REV	412
9.5.2	Weighted Rule Evaluation (REWW)	415
9.5.2.1	Set Up Prior to Executing REWW	415
9.5.2.2	Interrupt Details	416
9.5.2.3	Cycle-by-Cycle Details for REWW	417
9.6	WAV Instruction Details	419
9.6.1	Set Up Prior to Executing WAV	419
9.6.2	WAV Interrupt Details	420
9.6.3	Cycle-by-Cycle Details for WAV and wavr	420
9.7	Custom Fuzzy Logic Programming	422
9.7.1	Fuzzification Variations	423
9.7.2	Rule Evaluation Variations	425
9.7.3	Defuzzification Variations	425

Appendix A Instruction Reference

A.1	Introduction	427
A.2	Stack and Memory Layout	428
A.3	Interrupt Vector Locations	429
A.4	Notation Used in Instruction Set Summary	429
A.5	Hexadecimal-to-Decimal Conversion	462
A.6	Decimal-to-Hexadecimal Conversion	462

Appendix B CPU12V0 - CPU12V1

B.1	Introduction	463
B.2	CPU12V1 Modifications	463
B.3	Source Code Compatibility	463
B.4	Programmer's Model and Stacking	463
B.5	Improved Performance	464
B.6	Bus Signatures - Access Details	464

Appendix C CPU12 - CPU12X

C.1	Introduction	465
C.2	CPU12X Modifications	465
C.3	Source Code Compatibility	465
C.3.1	Instruction Set	465
C.4	Programmer's Model and Stacking	467
C.5	Improved Performance	467
C.6	Bus Signatures - Access Details	468

Appendix D CPU12XV0 - CPU12XV2

D.1	Introduction	469
D.2	CPU12XV2 Modifications	469
D.3	Source Code Compatibility	469
D.4	Programmer's Model and Stacking	469
D.5	Performance	470
D.6	Bus Signatures - Access Details	470
D.7	Instruction Execution in User State	470

Appendix E CPU12XV0 - CPU12XV1

E.1	Introduction	471
E.2	CPU12XV1 Modifications	471
E.3	Source Code Compatibility	471
E.4	Programmer's Model and Stacking	471
E.5	Performance	471
E.6	Bus Signatures - Access Details	472

Appendix F High-Level Language Support

F.1	Introduction	473
F.2	Data Types	473
F.3	Parameters and Variables	473
F.4	Register Pushes and Pulls	474
F.5	Allocating and Deallocating Stack Space	474
F.6	Frame Pointer	474
F.7	Increment and Decrement Operators	475
F.8	Higher Math Functions	475
F.9	Conditional If Constructs	475
F.10	Case and Switch Statements	476
F.11	Pointers	476
F.12	Function Calls	476
F.13	Instruction Set Orthogonality	476

Chapter 1

Introduction

M68HC12, HCS12 and HCS12X represent three generations of 16-bit controllers with all of them being derived from the industry standard M68HC11. This manual describes the features and operation of the family of central processing units (CPU) used on HCS12 and HCS12X micro controllers. Detailed information about the M68HC12 is provided in the *CPU12RM/AD Rev. 3*.

The term “CPU12 Family” is used if the content applies to both CPU12 and CPU12X generations.

The CPU12 generation includes the different versions used on HCS12 micro controllers, CPU12V0 and CPU12V1. The term CPU12 is used if the content applies to both the CPU12V0 and CPU12V1 versions.

The CPU12X generation includes the different versions used on HCS12X micro controllers, CPU12XV0, CPU12XV1 and CPU12XV2. The term CPU12X is used if the content applies to the CPU12XV0, CPU12XV1 and CPU12XV2 versions.

1.1 CPU Versions

Table 1-1 gives a coarse overview about the main differences between the five CPU12/CPU12X versions described in this document. Additionally, more subtle differences exist which are explained in the following chapters of this manual.

Table 1-1. CPU12 Family generation and version differences

Support for:	CPU12 Family				
Generation	CPU12		CPU12X		
Version	CPU12V0	CPU12V1	CPU12XV0	CPU12XV1	CPU12XV2
Fuzzy Instructions (MEM, REV, REVW, WAV)	✓	–	✓	–	–
CPU12X Instruction Set Architecture	–	–	✓	✓	✓
Single-Cycle 16 bits Multiply Operation	–	–	✓	✓	✓
Interrupt Priority Levels	–	–	✓	✓	✓
SYS Instruction	–	–	–	✓	✓
User State	–	–	–	–	✓

1.2 Features

The CPU12 Family is a high-speed, 16-bit processing CPU family which features a programming model identical to that of the industry standard M68HC11 central processing unit CPU11. The CPU12 instruction

set is a proper superset of the M68HC11 instruction set, and M68HC11 source code is accepted by CPU12 assemblers with no changes. Likewise, the CPU12X instruction set is a proper superset of the CPU12 instruction set, and CPU12 source code is accepted by CPU12X assemblers with no changes¹.

- Full 16-bit data paths supports efficient arithmetic operation and high-speed math execution
- Supports instructions with odd byte counts, including many single-byte instructions. This allows much more efficient use of ROM space.
- An instruction queue buffers program information so the CPU12X has immediate access to at least three bytes of machine code at the start of every instruction.
- Extensive set of indexed addressing capabilities, including:
 - Using the stack pointer as an indexing register in all indexed operations
 - Using the program counter as an indexing register in all but auto increment/decrement mode
 - Accumulator offsets using A, B, or D accumulators
 - Automatic index predecrement, preincrement, postdecrement, and postincrement (by -8 to +8)

1. With the exception of the Fuzzy instructions MEM, REV, REVW and WAV which are not supported on all versions of CPU12X.

1.3 Symbols and Notation

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

1.3.1 Abbreviations for System Resources

A	—	Accumulator A
B	—	Accumulator B
D	—	Double accumulator D (A : B)
TMP1	—	Register for CPU internal use only
TMP2	—	Register for CPU internal use only
TMP3	—	Register for CPU internal use only
X	—	Index register X
Y	—	Index register Y
SP	—	Stack pointer
PC	—	Program counter
CCR	—	Condition code register
U	—	User state bit (CPU12XV2 only)
IPL[2:0]	—	Interrupt Priority Level (CPU12X only)
S	—	STOP instruction control bit
X	—	Non-maskable interrupt control bit
H	—	Half-carry status bit
I	—	Maskable interrupt control bit
N	—	Negative status bit
Z	—	Zero status bit
V	—	Two's complement overflow status bit
C	—	Carry/Borrow status bit

1.3.2 Memory and Addressing

- M — 8-bit memory location pointed to by the effective address of the instruction
- M : M+1 — 16-bit memory location. Consists of the contents of the location pointed to by the effective address concatenated with the contents of the location at the next higher memory address. The most significant byte is at location M.
- M~M+3 — 32-bit memory location. Consists of the contents of the effective address of the instruction concatenated with the contents of the next three higher memory locations. The most significant byte is at location M or M_(Y).
- M_(Y)~M_(Y+3)
- M_(X) — Memory locations pointed to by index register X
- M_(SP) — Memory locations pointed to by the stack pointer
- M_(Y+3) — Memory locations pointed to by index register Y plus 3
- PPAGE — Program overlay page (bank) number for extended memory (>64KBs).
- Page — Program overlay page
- X_H — High-order byte
- X_L — Low-order byte
- () — Content of register or memory location
- \$ — Hexadecimal value
- % — Binary value

1.3.3 Operators

- + — Addition
- — Subtraction
- — Logical AND
- + — Logical OR (inclusive)
- ⊕ — Logical exclusive OR
- × — Multiplication
- ÷ — Division
- \overline{M} — Negation. One's complement (invert each bit of M)
- : — Concatenate
Example: A : B means the 16-bit value formed by concatenating 8-bit accumulator A with 8-bit accumulator B.
A is in the high-order position.
- ⇒ — Transfer
Example: (A) ⇒ M means the content of accumulator A is transferred to memory location M.
- ⇔ — Exchange
Example: D ⇔ X means exchange the contents of D with those of X.

1.3.4 Definitions

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

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

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

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

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

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

ADDR is the mnemonic for address bus.

DATA is the mnemonic for data bus.

LSB means least significant bit or bits.

MSB means most significant bit or bits.

LSW means least significant word or words.

MSW means most significant word or words.

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

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

Chapter 2 Overview

2.1 Introduction

This section describes the CPU12 and the CPU12X programming models, register set, the data types used, and basic memory organization.

2.2 Programming Model

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

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

The CPU12X programming model shown in [Figure 2-1](#) features an additional high 8-bit portion of the condition code register (CCR_H) with the lower 8-bit portion identical to the CPU12 (CCR).

- CPU12X — 16-bit condition code register (CCR_W = CCR_H:CCR)
- CPU12XV2 — User bit in CCR_H

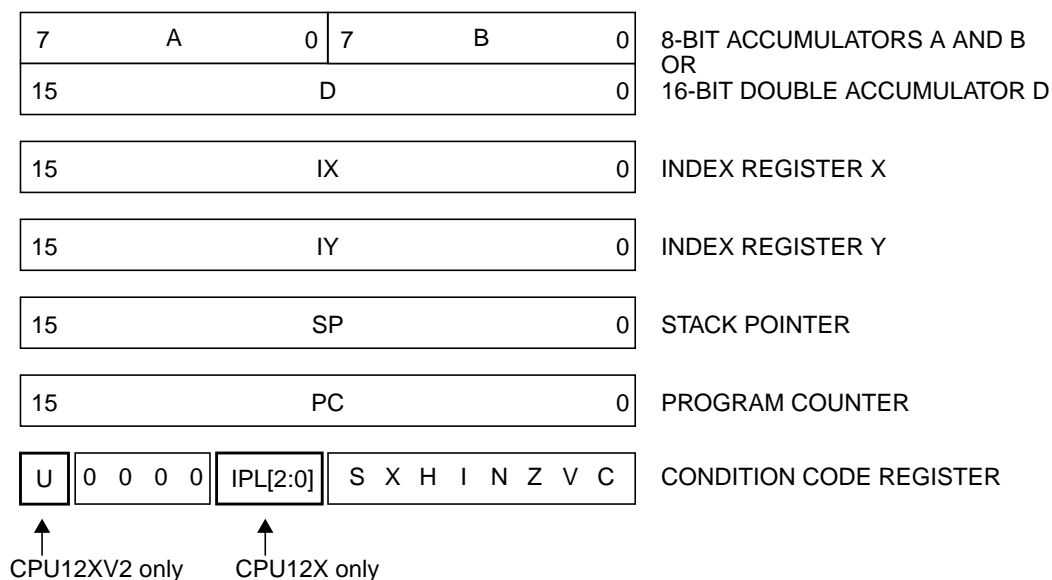


Figure 2-1. CPU12 Family Programming Model

2.2.1 Accumulators

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

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

2.2.2 Index Registers

16-bit index registers X and Y are used for indexed addressing. In the indexed addressing modes, the contents of an index register are added to 5-bit, 9-bit, or 16-bit constants or to the content of an accumulator to form the effective address of the instruction operand. The second index register is especially useful for moves and in cases where operands from two separate tables are used in a calculation. Additionally, the CPU12X provides a full range of arithmetic and logical operations for the X and Y registers.

2.2.3 Stack Pointer

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

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

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

When an interrupt occurs, the current instruction finishes execution. The address of the next instruction is calculated and pushed onto the stack, all the CPU registers are pushed onto the stack, the program counter is loaded with the address pointed to by the interrupt vector, and execution continues at that address. The stacked registers are referred to as an interrupt stack frame. CPU12X stack frame has increased by one byte compared to CPU12 stack frame.

NOTE

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

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

These instructions are removed on CPU12V1, CPU12XV1 and CPU12XV2

2.2.4 Program Counter

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

2.2.5 Condition Code Register

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

- Five status indicators
- Two interrupt masking bits
- STOP instruction control bit
- Interrupt Priority Level (CPU12X only)
- User state control bit (CPU12XV2 only)

The status bits reflect the results of CPU12 operation as it executes instructions. The five flags are:

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

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

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

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

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

The CPU12X extends this condition code register to a 16-Bit wide register. The lower byte is identical to the CPU12 version. The upper byte holds three bits reflecting the current processing level. These bits allow the nesting of interrupts, blocking interrupts of a lower priority. For details on interrupt processing refer to the Interrupt Block Guide.

The currently unused bits are reserved for future use and should be written to zero.

2.2.5.1 U Control Bit

Setting this bit brings the CPU12XV2 from Supervisor state (the default) into User state. In User state restrictions apply for the execution of several CPU12XV2 instructions:

1. Write access to the system control bits in the Condition Code Register (U, IPL[2:0], S, X, I) is blocked, i.e. any attempts to change these bits are ignored. This affects the following instructions:
 - ANDCC (including the alias instruction CLI)
 - ORCC (including the alias instruction SEI)
 - EXG with CCR, CCRH or CCRW
 - TFR with CCR, CCRH or CCRW as destination (including the alias instruction TAP)
 - PULC
 - PULCW
 - RTI
2. Instructions which would cause the CPU12XV2 to suspend instruction execution are treated as No-Operation instructions (NOP). This affects the following instructions:
 - STOP
 - WAI

Exceptions (see [Chapter 7, “Exception Processing”](#)) cause the CPU12XV2 to switch to Supervisor state, i.e. the U bit is automatically cleared when the CPU12XV2 starts exception processing. Executing the RTI instruction when exiting the exception handler restores the state of the U bit from the exception stack frame.

2.2.5.2 IPL[2:0]

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

2.2.5.3 S Control Bit

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

2.2.5.4 X Mask Bit

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

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

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

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

2.2.5.5 H Status Bit

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

2.2.5.6 I Mask Bit

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

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

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

2.2.5.7 N Status Bit

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

2.2.5.8 Z Status Bit

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

2.2.5.9 V Status Bit

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

2.2.5.10 C Status Bit

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

2.3 Data Types

The CPU12 uses these types of data:

- Bits
- 5-bit signed integers
- 8-bit signed and unsigned integers
- 8-bit, 2-digit binary-coded decimal numbers

- 9-bit signed integers
- 16-bit signed and unsigned integers
- 16-bit effective addresses
- 32-bit signed and unsigned integers

Negative integers are represented in two's complement form.

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

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

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

2.4 Memory Organization

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

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

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

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

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

2.5 Instruction Queue

The CPU12 Family uses an instruction queue to buffer program information. The mechanism is called a queue rather than a pipeline because a typical pipelined CPU executes more than one instruction at the same time, while the CPU12 Family always finishes executing an instruction before beginning to execute another. Refer to [Chapter 4, “Instruction Queue”](#) for more information.

Chapter 3 Addressing Modes

3.1 Introduction

Addressing modes determine how the CPU12 Family accesses memory locations to be operated upon. This section discusses the various modes and how they are used.

3.2 Mode Summary

Addressing modes are an implicit part of CPU12 Family instructions. Refer to [Appendix A, “Instruction Reference”](#) for the modes used by each instruction. All CPU12 Family addressing modes are shown in [Table 3-1](#).

No differences exist between CPU12 and CPU12X indexed modes. Instructions that use more than one mode are discussed in [Section 3.12, “Instructions Using Multiple Modes”](#).

3.3 Effective Address

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

Table 3-1. CPU12 Family Addressing Mode Summary

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

Table 3-1. CPU12 Family Addressing Mode Summary (continued)

Addressing Mode	Source Format	Abbreviation	Description
Indexed (pre-increment)	INST <i>opr</i> x3,+ <i>xy</i> s	IDX	Auto pre-increment x, y, or sp by 1 ~ 8
Indexed (post-decrement)	INST <i>opr</i> x3, <i>xy</i> s–	IDX	Auto post-decrement x, y, or sp by 1 ~ 8
Indexed (post-increment)	INST <i>opr</i> x3, <i>xy</i> s+	IDX	Auto post-increment x, y, or sp by 1 ~ 8
Indexed (accumulator offset)	INST <i>abd</i> , <i>xy</i> sp	IDX	Indexed with 8-bit (A or B) or 16-bit (D) accumulator offset from X, Y, SP, or PC
Indexed (9-bit offset)	INST <i>opr</i> x9, <i>xy</i> sp	IDX1	9-bit signed constant offset from X, Y, SP, or PC (lower 8 bits of offset in one extension byte)
Indexed (16-bit offset)	INST <i>opr</i> x16, <i>xy</i> sp	IDX2	16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (16-bit offset)	INST [<i>opr</i> x16, <i>xy</i> sp]	[IDX2]	Pointer to operand is found at... 16-bit constant offset from X, Y, SP, or PC (16-bit offset in two extension bytes)
Indexed-Indirect (D accumulator offset)	INST [D, <i>xy</i> sp]	[D,IDX]	Pointer to operand is found at... X, Y, SP, or PC plus the value in D

3.4 Inherent Addressing Mode

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

Examples:

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

3.5 Immediate Addressing Mode

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

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

Examples:

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

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

Example:

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

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

3.6 Direct Addressing Mode (CPU12)

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

Example:

```
LDAA          $55
```

This is a basic example of direct addressing. The value \$55 is taken to be the low-order half of an address in the range \$0000 through \$00FF. The high order half of the address is assumed to be 0. During execution of this instruction, the CPU12 combines the value \$55 from the instruction with the assumed value of \$00 to form the address \$0055, which is then used to access the data to be loaded into accumulator A.

Example:

```
LDX          $20
```

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

3.7 Direct Addressing Mode (CPU12X)

The Direct Page Register (DIRECT) (refer to Memory Controller Block Guide) determines the position of the direct page within the memory map. The direct addressing mode is used to access operands in the address range \$00 through \$FF in the direct page. Since these addresses always begin with the contents of the DIRECT register, only the eight low-order bits of the address need to be included in the instruction, which saves program space and execution time. A system can be optimized by placing the most commonly accessed data in this area of memory. The eight low-order bits of the operand address are supplied with the instruction, and the eight high-order bits of the address are assumed to be DIRECT.

3.8 Extended Addressing Mode

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

Example:

```
LDAA      $F03B
```

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

3.9 Relative Addressing Mode

The relative addressing mode is used only by branch instructions. Short and long conditional branch instructions use relative addressing mode exclusively, but branching versions of bit manipulation instructions (branch if bits set (BRSET) and branch if bits cleared (BRCLR)) use multiple addressing modes, including relative mode. Refer to [Section 3.12, “Instructions Using Multiple Modes”](#) for more information.

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

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

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

8-bit, 9-bit, and 16-bit offsets are signed two's complement numbers to support branching upward and downward in memory. The numeric range of short branch offset values is \$80 (–128) to \$7F (127). Loop primitive instructions support a 9-bit offset which allows a range of \$100 (–256) to \$0FF (255). The numeric range of long branch offset values is \$8000 (–32,768) to \$7FFF (32,767). If the offset is 0, the CPU12 executes the instruction immediately following the branch instruction, regardless of the test involved.

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

An offset that points to the opcode can cause a bit-condition branch to repeat execution until the specified bit condition is satisfied. Since bit-condition branches can consist of four, five, or six bytes depending on the addressing mode used to access the byte in memory, the offset value that sets up a loop can vary. For

instance, using an offset of \$FC with a BRCLR that accesses memory using an 8-bit indexed postbyte sets up a loop that executes until all the bits in the specified memory byte that correspond to 1s in the mask byte are cleared.

3.10 Indexed Addressing Modes

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

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

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

Major advantages of the CPU12 Family indexed addressing scheme are:

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

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

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

Table 3-2. Summary of Indexed Operations

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

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

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

3.10.1 5-Bit Constant Offset Indexed Addressing

This indexed addressing mode uses a 5-bit signed offset which is included in the instruction postbyte. This short offset is added to the base index register (X, Y, SP, or PC) to form the effective address of the memory location that will be affected by the instruction. This gives a range of -16 through +15 from the value in the base index register. Although other indexed addressing modes allow 9- or 16-bit offsets, those modes

also require additional extension bytes in the instruction for this extra information. The majority of indexed instructions in real programs use offsets that fit in the shortest 5-bit form of indexed addressing.

Examples:

LDAA	0, X
STAB	-8, Y

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

3.10.2 9-Bit Constant Offset Indexed Addressing

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

Examples:

LDAA	\$FF, X
LDAB	-20, Y

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

NOTE

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

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

3.10.3 16-Bit Constant Offset Indexed Addressing

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

3.10.4 16-Bit Constant Indirect Indexed Addressing

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

the location of a pointer to the address to be acted on. The square brackets distinguish this addressing mode from 16-bit constant offset indexing.

Example:

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

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

3.10.5 Auto Pre/Post Decrement/Increment Indexed Addressing

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

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

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

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

Examples:

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

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

For a “next-available” type of stack, push A onto stack (PSHA) is equivalent to store accumulator A (STAA) 1,SP– and pull A from stack (PULA) is equivalent to load accumulator A (LDAA) 1,SP.

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

Example:


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

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

3.10.6 Accumulator Offset Indexed Addressing

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

Example:

```
LDAA      B, X
```

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

Examples:

```
ABX
LDAA      0, X
```

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

3.10.7 Accumulator D Indirect Indexed Addressing

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

Examples:

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

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

Assume that the value in D is \$0002. The JMP instruction adds the values in D and PC to form the address of GO2. Next the CPU12 reads the address PLACE2 from memory at GO2 and jumps to PLACE2. The locations of PLACE1 through PLACE3 were known at the time of program assembly but the destination of the JMP depends upon the value in D computed during program execution.

3.11 Global Addressing (CPU12X only)

The CPU12 Core architecture limits the physical address space available to 64KB `addr[15:0]`. The CPU12X core architecture with the usage of the Global Page Index Register (refer to Memory Controller Block Guide) allows for integrating up to 8MB of memory `addr[22:0]` by using the seven global page index bits to page 64KB blocks into the memory map `addr[22:0]` is a result of concatenation between GPAGE and `addr[15:0]`.

New instructions started with the label G are created for this usage like (GLDAA, GSTAA,...).

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

GLDAA has the same addressing mode as LDAA with the only difference being the memory address (64KB) is replaced by the Global memory address (8MB).

This is the case for all Global instructions.

3.12 Instructions Using Multiple Modes

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

3.12.1 Move Instructions

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

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

In the CPU12 Move instructions do not support indirect modes, 9-bit, or 16-bit offset modes requiring extra extension bytes, while the CPU12X features all addressing modes for the source operand as well as for the destination operand. There are special considerations when using PC-relative addressing with move instructions.

PC-relative addressing uses the address of the location immediately following the last byte of object code for the current instruction as a reference point. The CPU12 normally corrects for queue offset and for instruction alignment so that queue operation is transparent to the user. However, in the CPU12X, move instructions using PC relative addressing pose a special problem:

- Some moves have object code that is too long to fit in the queue all at one time, so the PC value changes during execution.

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

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and the destination addressing mode is any of the three index addressing modes listed in [Table 3-3](#) below:

Table 3-3. Address Offsets for MOVB/MOVW using a PC-relative source address

Destination Address Mode	Offset for PC-relative source address
IDX1	+1
IDX2	+2
[IDX2]	+2

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

3.12.2 Bit Manipulation Instructions

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

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

The CPU12X includes BTAS (Bit Test And Set) that works by starting to test bits in memory location M, then set bits in memory location M. To test then set a bit, set the corresponding bit in the mask byte. All other bits in M are unchanged. BTAS is an atomic instruction and may be used to implement a semaphore.

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

3.13 Addressing More than 64KB

Some CPU12 Family devices incorporate hardware that supports addressing a larger memory space than the standard 64KB. The expanded memory system uses fast on-chip logic to implement a transparent bank-switching scheme (Section 3.11, “Global Addressing (CPU12X only)”).

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

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

The CPU12 Family systems require no external glue logic. Bank switching overhead is reduced by implementing control logic in the MCU. Interrupts do not need to be disabled during switching because switching tasks are incorporated in special instructions that greatly simplify program access to extended memory.

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

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

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

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

See specific device data sheet for more information on the memory layout of the particular device.

Chapter 4 Instruction Queue

4.1 Introduction

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

The material in this section is general. [Chapter 6, “Instruction Glossary”](#) contains detailed information concerning cycle-by-cycle execution of each instruction. [Chapter 8, “Debugging Support”](#) contains detailed information about tracking queue operation and instruction execution.

4.2 Queue Description

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

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

External pins, like IPIPE[1:0] for CPU12 and IQSTAT[3:0] for CPU12X, provide information about data movement in the queue and instruction execution. Decoding and use of these signals is discussed in [Chapter 8, “Debugging Support”](#).

4.2.1 CPU12 Family Queue Implementation

There are three 16-bit stages in the instruction queue. Instructions enter the queue at Stage_1 and shift out of Stage_3 as the CPU executes instructions and fetches new ones into Stage_1. Each byte in the queue is selectable. An opcode prediction algorithm determines the location of the next opcode in the instruction queue.

4.2.2 Data Movement in the Queue

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

4.2.3 No Movement

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

4.2.4 Advance and Load from Data Bus

The content of queue is advanced by one stage, and Stage_1 is loaded with a word of program information from the data bus. The information was requested two bus cycles earlier but has only become available this cycle, due to access delay.

4.2.5 Changes in Execution Flow

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

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

4.2.6 Exceptions

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

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

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

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

4.2.7 Subroutines

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

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

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

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

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

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

4.2.8 Branches

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

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

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

Branch instructions have two execution cases:

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

4.2.8.1 Short Branches

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

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

4.2.8.2 Long Branches

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

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

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

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

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

4.2.8.3 Bit Condition Branches

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

4.2.8.4 Loop Primitives

The loop primitive instructions test a counter value in a register or accumulator and branch to an address specified by a 9-bit relative offset contained in the instruction if a specified condition is met. There are auto-increment and auto-decrement versions of these instructions. The test and increment/decrement operations are performed on internal CPU registers, and require no additional program information. To shorten execution time, these branches perform one program word fetch in anticipation of the “taken” case. The data from this fetch is ignored if the branch is not taken, and the CPU does one program fetch and one optional fetch to refill the queue¹. If the branch is taken, the CPU finishes refilling the queue with two additional program word fetches at the new address.

4.2.9 Jumps

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

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

Chapter 5

Instruction Set Overview

5.1 Introduction

This section contains general information about the Central Processor Unit (CPU) instruction set. It is organized into instruction categories grouped by function.

5.2 Instruction Set Description

CPU12X instructions are a superset of the CPU12 instruction set. Code written for an CPU12 can be reassembled and run on a CPU12X with no changes. The CPU12X provides expanded functionality and increased code efficiency. There are two basic implementations of the CPU12 Family, the original CPU12 and the newer CPU12X. The CPU12X has an enhanced instruction set and for existing instructions there are small differences in cycle-by-cycle access details (the order of some bus cycles changed to accommodate differences in the way the instruction queue was implemented). These minor differences are transparent for most users.

In the CPU12 Family architecture, all memory and input/output (I/O) are mapped in a common 64KB address space (memory-mapped I/O). This allows the same set of instructions to be used to access memory, I/O, and control registers. General-purpose load, store, transfer, exchange, and move instructions facilitate movement of data to and from memory and peripherals.

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

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

Refer to [Chapter 6, “Instruction Glossary”](#) for detailed information about individual instructions. [Appendix A, “Instruction Reference”](#) contains quick-reference material, including an opcode map and postbyte encoding for indexed addressing, transfer/exchange instructions, and loop primitive instructions.

5.3 Load and Store Instructions

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

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

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

Table 5-1. Load and Store Instructions

Mnemonic	Function	Operation
Load Instructions		
GLDAA ¹	Global Load A	$(M) \Rightarrow A$
GLDAB ¹	Global Load B	$(M) \Rightarrow B$
GLDD ¹	Global Load D	$(M : M + 1) \Rightarrow (A:B)$
GLDS ¹	Global Load SP	$(M : M + 1) \Rightarrow SP_H:SP_L$
GLDX ¹	Global Load index register X	$(M : M + 1) \Rightarrow X_H:X_L$
GLDY ¹	Global Load index register Y	$(M : M + 1) \Rightarrow Y_H:Y_L$
LDAA	Load A	$(M) \Rightarrow A$
LDAB	Load B	$(M) \Rightarrow B$
LDD	Load D	$(M : M + 1) \Rightarrow (A:B)$
LDS	Load SP	$(M : M + 1) \Rightarrow SP_H:SP_L$
LDX	Load index register X	$(M : M + 1) \Rightarrow X_H:X_L$
LDY	Load index register Y	$(M : M + 1) \Rightarrow Y_H:Y_L$
LEAS	Load effective address into SP	Effective address \Rightarrow SP
LEAX	Load effective address into X	Effective address \Rightarrow X
LEAY	Load effective address into Y	Effective address \Rightarrow Y
Store Instructions		
GSTAA ¹	Global Store A	$(A) \Rightarrow M$
GSTAB ¹	Global Store B	$(B) \Rightarrow M$
GSTD ¹	Global Store D	$(A) \Rightarrow M, (B) \Rightarrow M + 1$
GSTS ¹	Global Store SP	$(SP_H:SP_L) \Rightarrow M : M + 1$
GSTX ¹	Global Store X	$(X_H:X_L) \Rightarrow M : M + 1$
GSTY ¹	Global Store Y	$(Y_H:Y_L) \Rightarrow M : M + 1$
STAA	Store A	$(A) \Rightarrow M$
STAB	Store B	$(B) \Rightarrow M$
STD	Store D	$(A) \Rightarrow M, (B) \Rightarrow M + 1$
STS	Store SP	$(SP_H:SP_L) \Rightarrow M : M + 1$
STX	Store X	$(X_H:X_L) \Rightarrow M : M + 1$
STY	Store Y	$(Y_H:Y_L) \Rightarrow M : M + 1$

¹ CPU12X only

5.4 Transfer and Exchange Instructions

Transfer instructions copy the content of a register or accumulator into another register or accumulator. Source content is not changed by the operation. Transfer register to register (TFR) is a universal transfer instruction, but other mnemonics are accepted for compatibility with CPU11. The transfer A to B (TAB) and transfer B to A (TBA) instructions affect the N, Z, and V condition code bits in the same way as CPU11 instructions. The TFR instruction does not affect the condition code bits.

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

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

Chapter 6, “[Instruction Glossary](#)” contains information concerning other transfers and exchanges between 8- and 16-bit registers.

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

Table 5-2. Transfer and Exchange Instructions

Mnemonic	Function	Operation
Transfer Instructions		
TAB	Transfer A to B	$(A) \Rightarrow B$
TAP	Transfer A to CCR	$(A) \Rightarrow \text{CCR}$
TBA	Transfer B to A	$(B) \Rightarrow A$
TFR	Transfer register to register	$(A, B, \text{CCR}, D, X, Y, \text{ or } SP) \Rightarrow A, B, \text{CCR}, D, X, Y, \text{ or } SP$
TPA	Transfer CCR to A	$(\text{CCR}) \Rightarrow A$
TSX	Transfer SP to X	$(SP) \Rightarrow X$
TSY	Transfer SP to Y	$(SP) \Rightarrow Y$
TXS	Transfer X to SP	$(X) \Rightarrow SP$
TYS	Transfer Y to SP	$(Y) \Rightarrow SP$
Exchange Instructions		
EXG	Exchange register to register	$(A, B, \text{CCR}, D, X, Y, \text{ or } SP) \Leftrightarrow (A, B, \text{CCR}, D, X, Y, \text{ or } SP)$
XGDX	Exchange D with X	$(D) \Leftrightarrow (X)$
XGDY	Exchange D with Y	$(D) \Leftrightarrow (Y)$
Sign Extension Instruction		
SEX	Sign extend 8-Bit operand	Sign-extended $(A, B, \text{ or } \text{CCR}) \Rightarrow D, X, Y, \text{ or } SP$
SEX ¹	Sign extend 16-Bit operand	Sign-extended $(D) \Rightarrow X, Y$

¹ CPU12X only

5.5 Move Instructions

Move instructions move (copy) data bytes or words from a source (M_1 or $M : M + 1_1$) to a destination (M_2 or $M : M + 1_2$) in memory. On CPU12V0 there are six combinations of immediate, extended, and indexed addressing allowed to specify source and destination addresses ($\text{IMM} \Rightarrow \text{EXT}$, $\text{IMM} \Rightarrow \text{IDX}$, $\text{EXT} \Rightarrow \text{EXT}$, $\text{EXT} \Rightarrow \text{IDX}$, $\text{IDX} \Rightarrow \text{EXT}$, $\text{IDX} \Rightarrow \text{IDX}$). On CPU12XV0 there are 42 combinations of immediate, extended, and indexed addressing allowed to specify source and destination addresses. Addressing mode combinations with immediate for the destination would not be useful.

Table 5-3 shows byte and word move instructions.

Table 5-3. Move Instructions

Mnemonic	Function	Operation
MOVB	Move byte (8-bit)	$(M_1) \Rightarrow M_2$

Table 5-3. Move Instructions

MOVW	Move word (16-bit)	$(M : M + 1_1) \Rightarrow M : M + 1_2$
------	--------------------	---

5.6 Addition and Subtraction Instructions

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

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

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

Table 5-4. Addition and Subtraction Instructions

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

Table 5-4. Addition and Subtraction Instructions (continued) (continued)

Mnemonic	Function	Operation
SUBD	Subtract memory from D (A:B)	$(A:B) - (M : M + 1) \Rightarrow A : B$
SUBX ¹	Subtract memory from X	$(X) - (M : M + 1) \Rightarrow X$
SUBY ¹	Subtract memory from Y	$(Y) - (M : M + 1) \Rightarrow Y$

¹ CPU12X only

5.7 Binary-Coded Decimal Instructions

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

Table 5-5. BCD Instructions

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

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

5.8 Decrement and Increment Instructions

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

Table 5-6. Decrement and Increment Instructions

Mnemonic	Function	Operation
Decrement Instructions		
DEC	Decrement memory	$(M) - \$01 \Rightarrow M$
DECA	Decrement A	$(A) - \$01 \Rightarrow A$
DECB	Decrement B	$(B) - \$01 \Rightarrow B$
DECW ¹	Decrement memory	$(M : M + 1) - \$0001 \Rightarrow M : M + 1$
DECX ¹	Decrement X	$(X) - \$0001 \Rightarrow X$
DECY ¹	Decrement Y	$(Y) - \$0001 \Rightarrow Y$

Table 5-6. Decrement and Increment Instructions

Mnemonic	Function	Operation
DES	Decrement SP	$(SP) - \$0001 \Rightarrow SP$
DEX	Decrement X	$(X) - \$0001 \Rightarrow X$
DEY	Decrement Y	$(Y) - \$0001 \Rightarrow Y$
Increment Instructions		
INC	Increment memory	$(M) + \$01 \Rightarrow M$
INCA	Increment A	$(A) + \$01 \Rightarrow A$
INCB	Increment B	$(B) + \$01 \Rightarrow B$
INCW ¹	Increment memory	$(M : M + 1) + \$0001 \Rightarrow M : M + 1$
INCX ¹	Increment X	$(X) + \$0001 \Rightarrow X$
INCY ¹	Increment Y	$(Y) + \$0001 \Rightarrow Y$
INS	Increment SP	$(SP) + \$0001 \Rightarrow SP$
INX	Increment X	$(X) + \$0001 \Rightarrow X$
INY	Increment Y	$(Y) + \$0001 \Rightarrow Y$

¹ CPU12X only

5.9 Compare and Test Instructions

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

Table 5-7. Compare and Test Instructions

Mnemonic	Function	Operation
Compare Instructions		
CBA	Compare A to B	$(A) - (B)$
CMPA	Compare A to memory	$(A) - (M)$
CMPB	Compare B to memory	$(B) - (M)$
CPD	Compare D to memory (16-bit)	$(A : B) - (M : M + 1)$
CPED ¹	Compare D to memory with borrow (16-bit)	$(A : B) - (M : M + 1) - C$
CPES ¹	Compare SP to memory with borrow (16-bit)	$(SP) - (M : M + 1) - C$
CPEX ¹	Compare X to memory with borrow (16-bit)	$(X) - (M : M + 1) - C$
CPEY ¹	Compare Y to memory with borrow (16-bit)	$(Y) - (M : M + 1) - C$
CPS	Compare SP to memory (16-bit)	$(SP) - (M : M + 1)$
CPX	Compare X to memory (16-bit)	$(X) - (M : M + 1)$
CPY	Compare Y to memory (16-bit)	$(Y) - (M : M + 1)$
Test Instructions		
TST	Test memory for zero or minus	$(M) - \$00$

Table 5-7. Compare and Test Instructions (continued)

Mnemonic	Function	Operation
Compare Instructions		
TSTA	Test A for zero or minus	(A) – \$00
TSTB	Test B for zero or minus	(B) – \$00
TSTW ¹	Test memory for zero or minus	(M : M + 1) – \$0000
TSTX ¹	Test X for zero or minus	(X) – \$0000
TSTY ¹	Test Y for zero or minus	(Y) – \$0000

¹ CPU12X only

5.10 Boolean Logic Instructions

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

Table 5-8. Boolean Logic Instructions

Mnemonic	Function	Operation
ANDA	AND A with memory	(A) • (M) ⇒ A
ANDB	AND B with memory	(B) • (M) ⇒ B
ANDCC	AND CCR with memory (clear CCR bits)	(CCR) • (M) ⇒ CCR
ANDX ¹	AND X with memory	(X) • (M : M + 1) ⇒ X
ANDY ¹	AND Y with memory	(Y) • (M : M + 1) ⇒ Y
EORA	Exclusive OR A with memory	(A) ⊕ (M) ⇒ A
EORB	Exclusive OR B with memory	(B) ⊕ (M) ⇒ B
EORX ¹	Exclusive OR X with memory	(X) ⊕ (M : M + 1) ⇒ X
EORY ¹	Exclusive OR Y with memory	(Y) ⊕ (M : M + 1) ⇒ Y
ORAA	OR A with memory	(A) + (M) ⇒ A
ORAB	OR B with memory	(B) + (M) ⇒ B
ORCC	OR CCR with memory (set CCR bits)	(CCR) + (M) ⇒ CCR
ORX ¹	OR X with memory	(X) + (M : M + 1) ⇒ X
ORY ¹	OR Y with memory	(Y) + (M : M + 1) ⇒ Y

¹ CPU12X only

5.11 Clear, Complement, and Negate Instructions

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

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

Mnemonic	Function	Operation
CLC	Clear C bit in CCR	$0 \Rightarrow C$
CLI	Clear I bit in CCR	$0 \Rightarrow I$
CLR	Clear memory	$\$00 \Rightarrow M$
CLRA	Clear A	$\$00 \Rightarrow A$
CLRB	Clear B	$\$00 \Rightarrow B$
CLRW ¹	Clear memory	$\$0000 \Rightarrow M : M + 1$
CLRX ¹	Clear X	$\$0000 \Rightarrow X$
CLRY ¹	Clear Y	$\$0000 \Rightarrow Y$
CLV	Clear V bit in CCR	$0 \Rightarrow V$
COM	One's complement memory	$\$FF - (M) \Rightarrow M$ or $(\bar{M}) \Rightarrow M$
COMA	One's complement A	$\$FF - (A) \Rightarrow A$ or $(\bar{A}) \Rightarrow A$
COMB	One's complement B	$\$FF - (B) \Rightarrow B$ or $(\bar{B}) \Rightarrow B$
COMW ¹	One's complement memory	$\$FFFF - (M : M + 1) \Rightarrow M : M + 1$ or $(\bar{M} : \bar{M} + 1) \Rightarrow M : M + 1$
COMX ¹	One's complement X	$\$FFFF - (X) \Rightarrow X$ or $(\bar{X}) \Rightarrow X$
COMY ¹	One's complement Y	$\$FFFF - (Y) \Rightarrow Y$ or $(\bar{Y}) \Rightarrow Y$
NEG	Two's complement memory	$\$00 - (M) \Rightarrow M$ or $(\bar{M}) + 1 \Rightarrow M$
NEGA	Two's complement A	$\$00 - (A) \Rightarrow A$ or $(\bar{A}) + 1 \Rightarrow A$
NEGB	Two's complement B	$\$00 - (B) \Rightarrow B$ or $(\bar{B}) + 1 \Rightarrow B$
NEGW ¹	Two's complement memory	$\$0000 - (M : M + 1) \Rightarrow M : M + 1$ or $(\bar{M} : \bar{M} + 1) + 1 \Rightarrow M : M + 1$
NEGX ¹	Two's complement X	$\$0000 - (X) \Rightarrow X$ or $(\bar{X}) + 1 \Rightarrow X$
NEGY ¹	Two's complement Y	$\$0000 - (Y) \Rightarrow Y$ or $(\bar{Y}) + 1 \Rightarrow Y$

¹ CPU12X only

5.12 Multiplication and Division Instructions

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

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

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

Table 5-10. Multiplication and Division Instructions

Mnemonic	Function	Operation
Multiplication Instructions		
EMUL	16 by 16 multiply (unsigned)	$(D) \times (Y) \Rightarrow Y : D$

Table 5-10. Multiplication and Division Instructions

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

5.13 Bit Test and Manipulation Instructions

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

Table 5-11. Bit Test and Manipulation Instructions

Mnemonic	Function	Operation
BCLR	Clear bits in memory	$(M) \bullet (\overline{mm}) \Rightarrow M$
BITA	Bit test A	$(A) \bullet (M)$
BITB	Bit test B	$(B) \bullet (M)$
BITX ¹	Bit test X	$(X) \bullet (M : M + 1)$
BITY ¹	Bit test Y	$(Y) \bullet (M : M + 1)$
BSET	Set bits in memory	$(M) + (mm) \Rightarrow M$

¹ CPU12X only

5.14 Shift and Rotate Instructions

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

Table 5-12. Shift and Rotate Instructions

Mnemonic	Function	Operation
Logical Shifts		
LSL LSLA LSLB	Logic shift left memory Logic shift left A Logic shift left B	
LSLD LSLW ¹ LSLX ¹ LSLY ¹	Logic shift left D Logic shift left memory Logic shift left X Logic shift left Y	
LSR LSRA LSRB	Logic shift right memory Logic shift right A Logic shift right B	
LSRD LSRW ¹ LSRX ¹ LSRY ¹	Logic shift right D Logic shift right memory Logic shift right X Logic shift right Y	
Arithmetic Shifts		
ASL ASLA ASLB	Arithmetic shift left memory Arithmetic shift left A Arithmetic shift left B	
ASLD ASLW ¹ ASLX ¹ ASLY ¹	Arithmetic shift left D Arithmetic shift left memory Arithmetic shift left X Arithmetic shift left Y	
ASR ASRA ASRB	Arithmetic shift right memory Arithmetic shift right A Arithmetic shift right B	
ASRW ¹ ASRX ¹ ASRY ¹	Arithmetic shift right memory Arithmetic shift right X Arithmetic shift right Y	
Rotates		
ROL ROLA ROLB	Rotate left memory through carry Rotate left A through carry Rotate left B through carry	
ROLW ¹ ROLX ¹ ROLY ¹	Rotate left memory through carry Rotate left X through carry Rotate left Y through carry	
ROR RORA RORB	Rotate right memory through carry Rotate right A through carry Rotate right B through carry	
RORW ¹ RORX ¹ RORY ¹	Rotate right memory through carry Rotate right X through carry Rotate right Y through carry	

¹ CPU12X only

5.15 Fuzzy Logic Instructions

The CPU12 Family instruction set includes instructions that support efficient processing of fuzzy logic operations. The descriptions of fuzzy logic instructions given here are functional overviews. [Table 5-13](#) summarizes the fuzzy logic instructions. Refer to [Chapter 9, “Fuzzy Logic Support”](#) for detailed discussion.

5.15.1 Fuzzy Logic Membership Instruction (CPU12V0,CPU12XV0 only)

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

5.15.2 Fuzzy Logic Rule Evaluation Instructions (CPU12V0,CPU12XV0 only)

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

5.15.3 Fuzzy Logic Weighted Average Instruction (CPU12V0,CPU12XV0 only)

The weighted average (WAV) instruction computes a sum-of-products and a sum-of-weights used for defuzzification. To be usable, the fuzzy outputs produced by rule evaluation must be defuzzified to produce a single output value which represents the combined effect of all of the fuzzy outputs. Fuzzy outputs correspond to the labels of a system output and each is defined by a membership function in the knowledge base. The 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 pseudo-instruction causes execution to resume at the point where it was interrupted.

Table 5-13. Fuzzy Logic Instructions

Mnemonic	Function	Operation
MEM	Membership function	$\mu(\text{grade}) \Rightarrow M_{(Y)}$ $(X) + 4 \Rightarrow X; (Y) + 1 \Rightarrow Y; A \text{ unchanged}$ <p>if $(A) < P1$ or $(A) > P2$, then $\mu = 0$, else $\mu = \text{MIN} [((A) - P1) \times S1, (P2 - (A)) \times S2, \\$FF]$ where: A = current crisp input value X points to a 4-byte data structure that describes a trapezoidal membership function as base intercept points and slopes (P1, P2, S1, S2) Y points at fuzzy input (RAM location)</p>
REV	MIN-MAX rule evaluation	<p>Find smallest rule input (MIN) Store to rule outputs unless fuzzy output is larger (MAX)</p> <p>Rules are unweighted</p> <p>Each rule input is an 8-bit offset from a base address in Y Each rule output is an 8-bit offset from a base address in Y \$FE separates rule inputs from rule outputs \$FF terminates the rule list</p> <p>REV can be interrupted</p>
REWV	MIN-MAX rule evaluation	<p>Find smallest rule input (MIN) Multiply by a rule weighting factor (optional) Store to rule outputs unless fuzzy output is larger (MAX)</p> <p>Each rule input is the 16-bit address of a fuzzy input Each rule output is the 16-bit address of a fuzzy output Address \$FFFE separates rule inputs from rule outputs \$FFFF terminates the rule list Weights are 8-bit values in a separate table</p> <p>REWV can be interrupted</p>
WAV	Calculates numerator (sum of products) and Denominator (Sum of Weights) for Weighted Average Calculation Results Are Placed in Correct Registers for EDIV immediately after WAV	$\sum_{i=1}^B S_i F_i \Rightarrow Y:D$ $\sum_{i=1}^B F_i \Rightarrow X$
wavr	Resumes execution of interrupted WAV instruction	Recover immediate results from stack rather than initializing them to 0.

5.16 Maximum and Minimum Instructions

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

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

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

Table 5-14. Minimum and Maximum Instructions

Mnemonic	Function	Operation
Minimum Instructions		
EMIND	MIN of two unsigned 16-bit values Result to Accumulator	$\text{MIN}((D), (M : M + 1)) \Rightarrow D$
EMINM	MIN of two unsigned 16-bit values Result to Memory	$\text{MIN}((D), (M : M + 1)) \Rightarrow M : M + 1$
MINA	MIN of two unsigned 8-bit values result to accumulator	$\text{MIN}((A), (M)) \Rightarrow A$
MINM	MIN of two unsigned 8-bit values result to memory	$\text{MIN}((A), (M)) \Rightarrow M$
Maximum Instructions		
EMAXD	MAX of two unsigned 16-bit values Result to Accumulator	$\text{MAX}((D), (M : M + 1)) \Rightarrow D$
EMAXM	MAX of two unsigned 16-bit values Result to Memory	$\text{MAX}((D), (M : M + 1)) \Rightarrow M : M + 1$
MAXA	MAX of two unsigned 8-bit values Result to Accumulator	$\text{MAX}((A), (M)) \Rightarrow A$
MAXM	MAX of two unsigned 8-bit values Result to Memory	$\text{MAX}((A), (M)) \Rightarrow M$

5.17 Multiply and Accumulate Instruction

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

Table 5-15. Multiply and Accumulate Instruction

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

5.18 Table Interpolation Instructions

The table interpolation instructions (TBL and ETBL) interpolate values from tables stored in memory. Any function that can be represented as a series of linear equations can be represented by a table of appropriate

size. Interpolation can be used for many purposes, including tabular fuzzy logic membership functions. TBL uses 8-bit table entries and returns an 8-bit result; ETBL uses 16-bit table entries and returns a 16-bit result. Use of indexed addressing mode provides great flexibility in structuring tables.

Consider each of the successive values stored in a table to be y-values for the endpoint of a line segment. The value in the B accumulator before instruction execution begins represents the change in x from the beginning of the line segment to the lookup point divided by total change in x from the beginning to the end of the line segment. B is treated as an 8-bit binary fraction with radix point left of the MSB, so each line segment is effectively divided into 256 smaller segments. During instruction execution, the change in y between the beginning and end of the segment (a signed byte for TBL or a signed word for ETBL) is multiplied by the content of the B accumulator to obtain an intermediate delta-y term. The result (stored in the A accumulator by TBL, and in the D accumulator by ETBL) is the y-value of the beginning point plus the signed intermediate delta-y value. Table 5-16 shows the table interpolation instructions.

Table 5-16. Table Interpolation Instructions

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

5.19 Branch Instructions

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

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

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

5.19.1 Short Branch Instructions

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

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

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

Table 5-17. Short Branch Instructions

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

5.19.2 Long Branch Instructions

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

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

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

Table 5-18. Long Branch Instructions

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

5.19.3 Bit Condition Branch Instructions

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

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

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

Table 5-19. Bit Condition Branch Instructions

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

5.20 Loop Primitive Instructions

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

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

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

Table 5-20. Loop Primitive Instructions

Mnemonic	Function	Equation or Operation
DBEQ	Decrement counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	$(\text{counter}) - 1 \Rightarrow \text{counter}$ If (counter) = 0, then branch; else continue to next instruction
DBNE	Decrement counter and branch if \neq 0 (counter = A, B, D, X, Y, or SP)	$(\text{counter}) - 1 \Rightarrow \text{counter}$ If (counter) not = 0, then branch; else continue to next instruction
IBEQ	Increment counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	$(\text{counter}) + 1 \Rightarrow \text{counter}$ If (counter) = 0, then branch; else continue to next instruction
IBNE	Increment counter and branch if \neq 0 (counter = A, B, D, X, Y, or SP)	$(\text{counter}) + 1 \Rightarrow \text{counter}$ If (counter) not = 0, then branch; else continue to next instruction
TBEQ	Test counter and branch if = 0 (counter = A, B, D, X, Y, or SP)	If (counter) = 0, then branch; else continue to next instruction
TBNE	Test counter and branch if \neq 0 (counter = A, B, D, X, Y, or SP)	If (counter) not = 0, then branch; else continue to next instruction

5.21 Jump and Subroutine Instructions

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

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

The call subroutine in expanded memory (CALL) instruction is intended for use with expanded memory. CALL stacks the value in the PPAGE register and the return address, then writes a new value to PPAGE to select the memory page where the subroutine resides. The page value is an immediate operand in all addressing modes except indexed indirect modes; in these modes, an operand points to locations in memory where the new page value and subroutine address are stored. The return from call (RTC) instruction is used to terminate subroutines in expanded memory. RTC unstacks the PPAGE value and the

return address so that execution resumes with the next instruction after CALL. For software compatibility, CALL and RTC execute correctly on devices that do not have expanded addressing capability. Table 5-21 summarizes the jump and subroutine instructions.

Table 5-21. Jump and Subroutine Instructions

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

5.22 Interrupt Instructions

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

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

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

The CPU12 Family uses a variation of the software interrupt for unimplemented opcode trapping. There are opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used in CPU12. 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 system call interrupt (SYS; CPU12XV1 and CPU12XV2 only) instruction behaves similar to an unimplemented opcode trap except that the interrupt vector used is \$FF12:\$FF13. This instruction is meant

to be used as a substitute for SWI in application code to better separate this from the debugging functionality of SWI.

The RTI instruction is used to terminate all exception handlers, including interrupt service routines. RTI first restores the CCRH (CPU12X only): CCR, B:A, X, Y, and the return address from the stack. If no other interrupt is pending, normal execution resumes with the instruction following the last instruction that executed prior to interrupt.

Table 5-22 is a summary of interrupt instructions.

Table 5-22. Interrupt Instructions

Mnemonic	Function	Operation
RTI	Return from interrupt	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H : CCR; (SP) - \$0000 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) - \$0002 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) - \$0004 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) - \$0006 \Rightarrow SP$ $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) - \$0008 \Rightarrow SP$
SWI	Software interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$
SYS ¹	System call interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$
TRAP	Unimplemented opcode interrupt	$SP - 2 \Rightarrow SP; RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; Y_H : Y_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; X_H : X_L \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; B : A \Rightarrow M_{(SP)} : M_{(SP+1)}$ $SP - 2 \Rightarrow SP; CCR_H : CCR \Rightarrow M_{(SP)} : M_{(SP+1)}$

¹ CPU12XV1 and CPU12XV2 only

5.23 Index Manipulation Instructions

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

Table 5-23. Index Manipulation Instructions

Mnemonic	Function	Operation
Addition Instructions		
ABX	Add B to X	$(B) + (X) \Rightarrow X$
ABY	Add B to Y	$(B) + (Y) \Rightarrow Y$
Compare Instructions		
CPES ¹	Compare SP to memory with borrow	$(SP) - (M : M + 1) - C$
CPEX ¹	Compare X to memory with borrow	$(X) - (M : M + 1) - C$
CPEY ¹	Compare Y to memory with borrow	$(Y) - (M : M + 1) - C$

Table 5-23. Index Manipulation Instructions (continued)

CPS	Compare SP to memory	$(SP) - (M : M + 1)$
CPX	Compare X to memory	$(X) - (M : M + 1)$
CPY	Compare Y to memory	$(Y) - (M : M + 1)$
Load Instructions		
GLDS ¹	Global Load SP from memory	$M : M + 1 \Rightarrow SP$
GLDX ¹	Global Load X from memory	$(M : M + 1) \Rightarrow X$
GLDY ¹	Global Load Y from memory	$(M : M + 1) \Rightarrow Y$
LDS	Load SP from memory	$(M : M + 1) \Rightarrow SP$
LDX	Load X from memory	$(M : M + 1) \Rightarrow X$
LDY	Load Y from memory	$(M : M + 1) \Rightarrow Y$
LEAS	Load effective address into SP	Effective address \Rightarrow SP
LEAX	Load effective address into X	Effective address \Rightarrow X
LEAY	Load effective address into Y	Effective address \Rightarrow Y
Store Instructions		
GSTS ¹	Global Store SP in memory	$(SP) \Rightarrow M : M + 1$
GSTX ¹	Global Store X in memory	$(X) \Rightarrow M : M + 1$
GSTY ¹	Global Store Y in memory	$(Y) \Rightarrow M : M + 1$
STS	Store SP in memory	$(SP) \Rightarrow M : M + 1$
STX	Store X in memory	$(X) \Rightarrow M : M + 1$
STY	Store Y in memory	$(Y) \Rightarrow M : M + 1$
Transfer Instructions		
TFR	Transfer register to register	$(A, B, CCR, D, X, Y, \text{ or } SP) \Rightarrow A, B, CCR, D, X, Y, \text{ or } SP$
TSX	Transfer SP to X	$(SP) \Rightarrow X$
TSY	Transfer SP to Y	$(SP) \Rightarrow Y$
TXS	transfer X to SP	$(X) \Rightarrow SP$
TYS	transfer Y to SP	$(Y) \Rightarrow SP$
Exchange Instructions		
EXG	Exchange register to register	$(A, B, CCR, D, X, Y, \text{ or } SP) \Leftrightarrow (A, B, CCR, D, X, Y, \text{ or } SP)$
XGDX	EXchange D with X	$(D) \Leftrightarrow (X)$
XGDY	EXchange D with Y	$(D) \Leftrightarrow (Y)$

¹ CPU12X only

5.24 Stacking Instructions

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

Table 5-24. Stacking Instructions

Mnemonic	Function	Operation
Stack Pointer Instructions		
CPES ¹	Compare SP to memory with borrow	$(SP) - (M : M + 1) - C$
CPS	Compare SP to memory	$(SP) - (M : M + 1)$
DES	Decrement SP	$(SP) - 1 \Rightarrow SP$
INS	Increment SP	$(SP) + 1 \Rightarrow SP$
GLDS ¹	Global Load SP	$(M : M + 1) \Rightarrow SP$
GSTS ¹	Global Store SP	$(SP) \Rightarrow M : M + 1$
LDS	Load SP	$(M : M + 1) \Rightarrow SP$
LEAS	Load effective address into SP	Effective address $\Rightarrow SP$
STS	Store SP	$(SP) \Rightarrow M : M + 1$
TSX	Transfer SP to X	$(SP) \Rightarrow X$
TSY	Transfer SP to Y	$(SP) \Rightarrow Y$
TXS	Transfer X to SP	$(X) \Rightarrow SP$
TYS	Transfer Y to SP	$(Y) \Rightarrow SP$
Stack Operation Instructions		
PSHA	Push A	$(SP) - 1 \Rightarrow SP; (A) \Rightarrow M_{(SP)}$
PSHB	Push B	$(SP) - 1 \Rightarrow SP; (B) \Rightarrow M_{(SP)}$
PSHC	Push CCR	$(SP) - 1 \Rightarrow SP; (CCR) \Rightarrow M_{(SP)}$
PSHCW ¹	Push CCR _H :CCR	$(SP) - 2 \Rightarrow SP; (CCR_H:CCR) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHD	Push D	$(SP) - 2 \Rightarrow SP; (A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHX	Push X	$(SP) - 2 \Rightarrow SP; (X) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PSHY	Push Y	$(SP) - 2 \Rightarrow SP; (Y) \Rightarrow M_{(SP)} : M_{(SP+1)}$
PULA	Pull A	$(M_{(SP)}) \Rightarrow A; (SP) + 1 \Rightarrow SP$
PULB	Pull B	$(M_{(SP)}) \Rightarrow B; (SP) + 1 \Rightarrow SP$
PULC	Pull CCR	$(M_{(SP)}) \Rightarrow CCR; (SP) + 1 \Rightarrow SP$
PULCW ¹	Pull CCR _H :CCR	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H:CCR; (SP) + 2 \Rightarrow SP$
PULD	Pull D	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B; (SP) + 2 \Rightarrow SP$
PULX	Pull X	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X; (SP) + 2 \Rightarrow SP$
PULY	Pull Y	$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y; (SP) + 2 \Rightarrow SP$

¹ CPU12X only

5.25 Pointer and Index Calculation Instructions

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

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

Table 5-25. Pointer and Index Calculation Instructions

Mnemonic	Function	Operation
LEAS	Load result of indexed addressing mode Effective Address Calculation into Stack Pointer	$r \pm \text{constant} \Rightarrow \text{SP}$ or $(r) + (\text{accumulator}) \Rightarrow \text{SP}$ $r = \text{X, Y, SP, or PC}$
LEAX	Load result of indexed addressing mode Effective Address Calculation into X Index Register	$r \pm \text{constant} \Rightarrow \text{X}$ or $(r) + (\text{accumulator}) \Rightarrow \text{X}$ $r = \text{X, Y, SP, or PC}$
LEAY	Load result of indexed addressing mode Effective Address Calculation into Y Index Register	$r \pm \text{constant} \Rightarrow \text{Y}$ or $(r) + (\text{accumulator}) \Rightarrow \text{Y}$ $r = \text{X, Y, SP, or PC}$

5.26 Condition Code Instructions

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

Table 5-26. Condition Code Instructions

Mnemonic	Function	Operation
ANDCC	Logical AND CCR with memory	$(\text{CCR}) \bullet (\text{M}) \Rightarrow \text{CCR}$
CLC	Clear C bit	$0 \Rightarrow \text{C}$
CLI	Clear I bit	$0 \Rightarrow \text{I}$
CLV	Clear V bit	$0 \Rightarrow \text{V}$
ORCC	Logical OR CCR with memory	$(\text{CCR}) + (\text{M}) \Rightarrow \text{CCR}$
PSHC	Push CCR onto stack	$(\text{SP}) - 1 \Rightarrow \text{SP}; \text{CCR} \Rightarrow \text{M}_{(\text{SP})}$
PSHCW	Push CCR _H :CCR onto stack	$(\text{SP}) - 2 \Rightarrow \text{SP}; (\text{CCR}_\text{H}:\text{CCR}) \Rightarrow \text{M}_{(\text{SP})}:\text{M}_{(\text{SP}+1)}$
PULC	Pull CCR from stack	$\text{M}_{(\text{SP})} \Rightarrow \text{CCR}; (\text{SP}) + 1 \Rightarrow \text{SP}$
PULCW	Pull CCR _H :CCR from stack	$\text{M}_{(\text{SP})}:\text{M}_{(\text{SP}+1)} \Rightarrow \text{CCR}_\text{H}:\text{CCR}; (\text{SP}) + 2 \Rightarrow \text{SP}$
SEC	Set C bit	$1 \Rightarrow \text{C}$
SEI	Set I bit	$1 \Rightarrow \text{I}$
SEV	Set V bit	$1 \Rightarrow \text{V}$
TAP	Transfer A to CCR	$(\text{A}) \Rightarrow \text{CCR}$
TPA	Transfer CCR to A	$(\text{CCR}) \Rightarrow \text{A}$

5.27 Stop and Wait Instructions

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

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

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

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

Table 5-27. Stop and Wait Instructions

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

5.28 Background Mode and Null Operations

Background debug mode (BDM) is a special CPU operating mode that is used for system development and debugging. Executing enter background debug mode (BGND) when BDM is enabled puts the CPU in this mode. For complete information, refer to [Chapter 8, “Debugging Support”](#).

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

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

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

Table 5-28. Background Mode and Null Operation Instructions

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

Chapter 6

Instruction Glossary

6.1 Introduction

This section is a comprehensive reference to the CPU12X instruction set.

CPU12 information is included for a reference point of view.

NOTE

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

6.2 Glossary Information

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

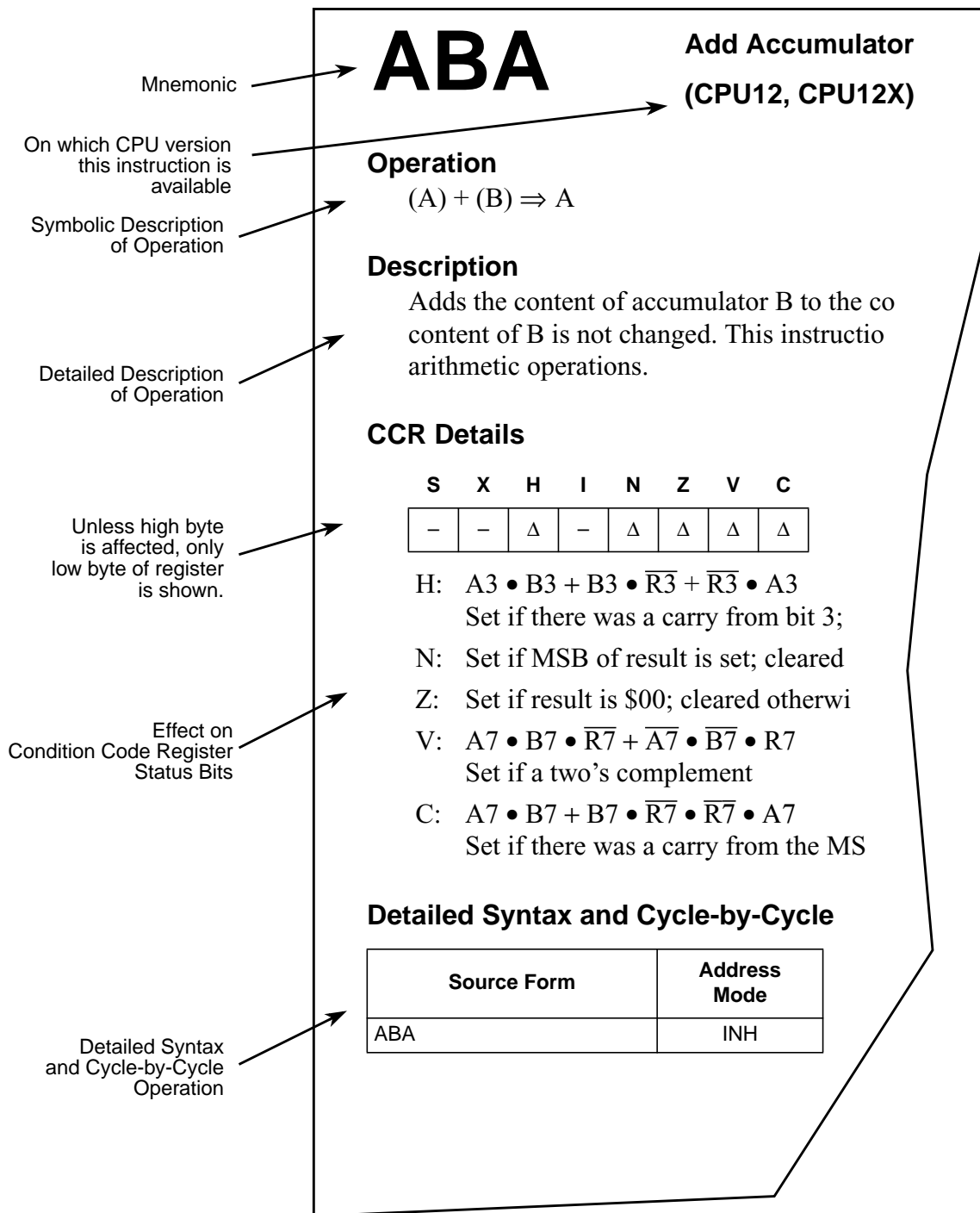


Figure 6-1. Example Glossary Page

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

6.3 Condition Code Changes

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

- — Status bit not affected by operation
- 0 — Status bit cleared by operation
- 1 — Status bit set by operation
- Δ — Status bit affected by operation
- ⇓ — Status bit may be cleared or remain set, but is not set by operation.
- ⇑ — Status bit may be set or remain cleared, but is not cleared by operation.
- ? — Status bit may be changed by operation, but the final state is not defined.
- ! — Status bit used for a special purpose

6.4 Object Code Notation

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

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

6.5 Source Forms

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

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

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

Groups of italic characters in the columns represent variable information to be supplied by the programmer. These groups can include any alphanumeric character or the underscore character, but cannot include a space or comma. For example, the groups *xysp* and *opr_{x0}_xysp* are both valid, but the two groups *opr_{x0} xysp* are not valid because there is a space between them. Permitted syntax is described here.

The definition of a legal label or expression varies from assembler to assembler. Assemblers also vary in the way CPU12 Family registers are specified. Refer to assembler documentation for detailed information. Recommended register designators are a, A, b, B, ccr, CCR, d, D, x, X, y, Y, sp, SP, pc, and PC.

- abc* — Any one legal register designator for accumulators A or B or the CCR
- abcdxys* — Any one legal register designator for accumulators A or B, the CCR, the double accumulator D, index registers X or Y, or the SP. Some assemblers may accept t2, T2, t3, or T3 codes in certain cases of transfer and exchange instructions, but these forms are intended for Freescale use only.
- abd* — Any one legal register designator for accumulators A or B or the double accumulator D
- abdxys* — Any one legal register designator for accumulators A or B, the double accumulator D, index register X or Y, or the SP
- dxys* — Any one legal register designation for the double accumulator D, index registers X or Y, or the SP
- msk8* — Any label or expression that evaluates to an 8-bit value. Some assemblers require a # symbol before this value.
- opr8i* — Any label or expression that evaluates to an 8-bit immediate value
- opr16i* — Any label or expression that evaluates to a 16-bit immediate value
- opr8a* — Any label or expression that evaluates to an 8-bit value. The instruction treats this 8-bit value as the low-order 8 bits of an address in the direct page of the 64KB address space (\$00xx).

- opr16a* — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64KB address space.
- opr0_xysp* — This word breaks down into one of the following alternative forms that assemble to an 8-bit indexed addressing postbyte code. These forms generate the same object code except for the value of the postbyte code, which is designated as *xb* in the object code columns of the glossary pages. As with the source forms, treat all commas, plus signs, and minus signs as literal syntax elements. The italicized words used in these forms are included in this key.
- opr5,xysp*
opr3,-xys
opr3,+xys
opr3,xys-
opr3,xys+
abd,xysp
- opr3* — Any label or expression that evaluates to a value in the range +1 to +8
- opr5* — Any label or expression that evaluates to a 5-bit value in the range -16 to +15
- opr9* — Any label or expression that evaluates to a 9-bit value in the range -256 to +255
- opr16* — Any label or expression that evaluates to a 16-bit value. Since the CPU12 Family has a 16-bit address bus, this can be either a signed or an unsigned value.
- page* — Any label or expression that evaluates to an 8-bit value. The CPU recognizes up to an 8-bit page value for memory expansion but not all MCUs from the CPU12 Family implement all of these bits. It is the programmer's responsibility to limit the page value to legal values for the intended MCU system. Some assemblers require a # symbol before this value.
- rel8* — Any label or expression that refers to an address that is within -128 to +127 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 8-bit signed offset and include it in the object code for this instruction.
- rel9* — Any label or expression that refers to an address that is within -256 to +255 locations from the next address after the last byte of object code for the current instruction. The assembler will calculate the 9-bit signed offset and include it in the object code for this instruction. The sign bit for this 9-bit value is encoded by the assembler as a bit in the looping postbyte (*lb*) of one of the loop control instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE. The remaining eight bits of the offset are included as an extra byte of object code.
- rel16* — Any label or expression that refers to an address anywhere in the 64KB address space. The assembler will calculate the 16-bit signed offset between this address and the next address after the last byte of object code for this instruction and include it in the object code for this instruction.
- trapnum* — Any label or expression that evaluates to an 8-bit number in the range \$30-\$39 or \$40-\$FF. Used for TRAP instruction.

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

6.6 Cycle-by-Cycle Execution

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

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

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

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

- f — Free cycle. This indicates a cycle where the CPU does not require use of the system buses. An f cycle is always one cycle of the system bus clock. These cycles can be used by a queue controller or the background debug system to perform single cycle accesses without disturbing the CPU.
- g — Read 8-bit PPAGE register. These cycles are used only with the CALL instruction to read the current value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.
- I — Read indirect pointer. Indexed indirect instructions use this 16-bit pointer from memory to address the operand for the instruction. These are always 16-bit reads but they can be either aligned or misaligned. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned access to a memory that is not designed for single-cycle misaligned access.
- i — Read indirect PPAGE value. These cycles are only used with indexed indirect versions of the CALL instruction, where the 8-bit value for the memory expansion page register of the CALL destination is fetched from an indirect memory location. These cycles are stretched only when controlled by a chip-select circuit that is programmed for slow memory.

- n — Write 8-bit PPAGE register. These cycles are used only with the CALL and RTC instructions to write the destination value of the PPAGE register and are not visible on the external bus. Since the PPAGE register is an internal 8-bit register, these cycles are never stretched.
- NA — Not available
- O — Optional cycle. Program information is always fetched as aligned 16-bit words. When an instruction consists of an odd number of bytes, and the first byte is misaligned, an O cycle is used to make an additional program word access (P) cycle that maintains queue order. In all other cases, the O cycle appears as a free (f) cycle. The \$18 prebyte for page two opcodes is treated as a special 1-byte instruction. If the prebyte is misaligned, the O cycle is used as a program word access for the prebyte; if the prebyte is aligned, the O cycle appears as a free cycle. If the remainder of the instruction consists of an odd number of bytes, another O cycle is required some time before the instruction is completed. If the O cycle for the prebyte is treated as a P cycle, any subsequent O cycle in the same instruction is treated as an f cycle; if the O cycle for the prebyte is treated as an f cycle, any subsequent O cycle in the same instruction is treated as a P cycle. Optional cycles used for program word accesses can be extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. Optional cycles used as free cycles are never stretched.
- P — Program word access. Program information is fetched as aligned 16-bit words. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the program is stored externally. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory.
- r — 8-bit data read. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.
- R — 16-bit data read. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the corresponding data is stored in external memory. There can be additional stretching when the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to memory that is not designed for single-cycle misaligned access.
- s — Stack 8-bit data. These cycles are stretched only when controlled by a chip-select circuit programmed for slow memory.
- S — Stack 16-bit data. These cycles are extended to two bus cycles if the MCU is operating with an 8-bit external data bus and the SP is pointing to external memory. There can be additional stretching if the address space is assigned to a chip-select circuit programmed for slow memory. These cycles are also stretched if they correspond to misaligned accesses to a memory that is not designed for single cycle misaligned access. The internal RAM is designed to allow single cycle misaligned word access.

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

Special Notation for Branch Taken/Not Taken Cases

- PPP/P** — Short branches require three cycles if taken, one cycle if not taken. Since the instruction consists of a single word containing both an opcode and an 8-bit offset, the not-taken case is simple — the queue advances, another program word fetch is made, and execution continues with the next instruction. The taken case requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is determined, then the CPU performs three program word fetches from that address.
- OPPP/OPO** — Long branches require four cycles if taken, three cycles if not taken. Optional cycles are required because all long branches are page two opcodes, and thus include the \$18 prebyte. The CPU treats the prebyte as a special 1-byte instruction. If the prebyte is misaligned, the optional cycle is used to perform a program word access; if the prebyte is aligned, the optional cycle is used to perform a free cycle. As a result, both the taken and not-taken cases use one optional cycle for the prebyte. In the not-taken case, the queue must advance so that execution can continue with the next instruction, and another optional cycle is required to maintain the queue. The taken case requires that the queue be refilled so that execution can continue at a new address. First, the effective address of the destination is determined, then the CPU performs three program word fetches from that address.

6.7 Glossary

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

ABA

Add Accumulator B to Accumulator A
(CPU12, CPU12X)

ABA

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

- H: $A3 \bullet B3 + B3 \bullet \overline{R3} + \overline{R3} \bullet A3$
Set if there was a carry from bit 3; cleared otherwise
- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $A7 \bullet B7 \bullet \overline{R7} + \overline{A7} \bullet \overline{B7} \bullet R7$
Set if a two's complement overflow resulted from the operation; cleared otherwise
- C: $A7 \bullet B7 + B7 \bullet \overline{R7} + \overline{R7} \bullet A7$
Set if there was a carry from the MSB of the result; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ABA	INH	18 06	all 00

ABX

Add Accumulator B to Index Register X
(CPU12, CPU12X)

ABX

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12V1, CPU12X	CPU12V0
ABX <i>translates to...</i> LEAX B,X	IDX	1A E5	P	Pf

ABY

Add Accumulator B to Index Register Y
(CPU12, CPU12X)

ABY

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12V1, CPU12X	CPU12V0
ABY <i>translates to...</i> LEAY B,Y	IDX	19 ED	P	Pf

ADCA

Add with Carry to A
(CPU12, CPU12X)

ADCA

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

$$H: A3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet A3$$

Set if there was a carry from bit 3; cleared otherwise

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

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

$$V: A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$$

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

$$C: A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ADCA #opr8i	IMM	89 ii	P
ADCA opr8a	DIR	99 dd	rPf
ADCA opr16a	EXT	B9 hh ll	rPO
ADCA oprx0_xysp	IDX	A9 xb	rPf
ADCA oprx9_xysp	IDX1	A9 xb ff	rPO
ADCA oprx16_xysp	IDX2	A9 xb ee ff	frPP
ADCA [D,xysp]	[D,IDX]	A9 xb	fIf rPf
ADCA [opr16,xysp]	[IDX2]	A9 xb ee ff	fIP rPf

ADCB

Add with Carry to B
(CPU12, CPU12X)

ADCB

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

$$H: B3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet B3$$

Set if there was a carry from bit 3; cleared otherwise

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

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

$$V: B7 \bullet M7 \bullet \overline{R7} + \overline{B7} \bullet \overline{M7} \bullet R7$$

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

$$C: B7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet B7$$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ADCB #opr8i	IMM	C9 ii	P
ADCB opr8a	DIR	D9 dd	rPf
ADCB opr16a	EXT	F9 hh ll	rPO
ADCB oprx0_xysp	IDX	E9 xb	rPf
ADCB oprx9_xysp	IDX1	E9 xb ff	rPO
ADCB oprx16_xysp	IDX2	E9 xb ee ff	frPP
ADCB [D,xysp]	[D,IDX]	E9 xb	fIfrPf
ADCB [oprx16,xysp]	[IDX2]	E9 xb ee ff	fIPrPf

ADDA

Add without Carry to A
(CPU12, CPU12X)

ADDA

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

$$H: A3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet A3$$

Set if there was a carry from bit 3; cleared otherwise

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

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

$$V: A7 \bullet M7 \bullet \overline{R7} + \overline{A7} \bullet \overline{M7} \bullet R7$$

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

$$C: A7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet A7$$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ADDA #opr8i	IMM	8B ii	P
ADDA opr8a	DIR	9B dd	rPf
ADDA opr16a	EXT	BB hh ll	rPO
ADDA oprx0_xysp	IDX	AB xb	rPf
ADDA oprx9_xysp	IDX1	AB xb ff	rPO
ADDA oprx16_xysp	IDX2	AB xb ee ff	frPP
ADDA [D,xysp]	[D,IDX]	AB xb	fIfrPf
ADDA [opr16,xysp]	[IDX2]	AB xb ee ff	fIPrPf

ADDB

Add without Carry to B
(CPU12, CPU12X)

ADDB

Operation

$$(B) + (M) \Rightarrow B$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	Δ	–	Δ	Δ	Δ	Δ

$$H: B3 \bullet M3 + M3 \bullet \overline{R3} + \overline{R3} \bullet B3$$

Set if there was a carry from bit 3; cleared otherwise

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

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

$$V: B7 \bullet M7 \bullet \overline{R7} + \overline{B7} \bullet \overline{M7} \bullet R7$$

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

$$C: B7 \bullet M7 + M7 \bullet \overline{R7} + \overline{R7} \bullet B7$$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ADDB #opr8i	IMM	CB ii	P
ADDB opr8a	DIR	DB dd	rPf
ADDB opr16a	EXT	FB hh ll	rPO
ADDB oprx0_xysp	IDX	EB xb	rPf
ADDB oprx9_xysp	IDX1	EB xb ff	rPO
ADDB oprx16_xysp	IDX2	EB xb ee ff	frPP
ADDB [D,xysp]	[D,IDX]	EB xb	fIf rPf
ADDB [opr16_xysp]	[IDX2]	EB xb ee ff	fIP rPf

ADDD

Add Double Accumulator (CPU12, CPU12X)

ADDD

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ADDD #opr16i	IMM	C3 jj kk	PO
ADDD opr8a	DIR	D3 dd	RPf
ADDD opr16a	EXT	F3 hh ll	RPO
ADDD oprx0_xysp	IDX	E3 xb	RPf
ADDD oprx9_xysp	IDX1	E3 xb ff	RPO
ADDD oprx16_xysp	IDX2	E3 xb ee ff	fRPP
ADDD [D,xysp]	[D,IDX]	E3 xb	fIfRPF
ADDD [opr16,xysp]	[IDX2]	E3 xb ee ff	fIPRPF

ADDX

Add without Carry to X
(CPU12X)

ADDX

Operation

$$(X) + (M : M + 1) \Rightarrow X$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

V: $X15 \bullet M15 \bullet \overline{R15} + \overline{X15} \bullet \overline{M15} \bullet R15$

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

C: $X15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet X15$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	CPU12X
ADDX #opr16i	IMM	18 8B jj kk		OPO
ADDX opr8a	DIR	18 9B dd		ORPf
ADDX opr16a	EXT	18 BB hh ll		ORPO
ADDX oprx0_xysp	IDX	18 AB xb		ORPf
ADDX oprx9_xysp	IDX1	18 AB xb ff		ORPO
ADDX oprx16_xysp	IDX2	18 AB xb ee ff		OfrPP
ADDX [D,xysp]	[D,IDX]	18 AB xb		OfIfRPf
ADDX [oprx16,xysp]	[IDX2]	18 AB xb ee ff		OfIPRPf

ADDY

Add without Carry to Y
(CPU12X)

ADDY

Operation

$$(Y) + (M : M + 1) \Rightarrow Y$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

V: $Y15 \bullet M15 \bullet \overline{R15} + \overline{Y15} \bullet \overline{M15} \bullet R15$

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

C: $Y15 \bullet M15 + M15 \bullet \overline{R15} + \overline{R15} \bullet Y15$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ADDY #opr16i	IMM	18 CB jj kk	OPO
ADDY opr8a	DIR	18 DB dd	ORPf
ADDY opr16a	EXT	18 FB hh ll	ORPO
ADDY oprx0_xysp	IDX	18 EB xb	ORPf
ADDY oprx9_xysp	IDX1	18 EB xb ff	ORPO
ADDY oprx16_xysp	IDX2	18 EB xb ee ff	OfrPP
ADDY [D,xysp]	[D,IDX]	18 EB xb	OfrRPf
ADDY [opr16,xysp]	[IDX2]	18 EB xb ee ff	OfrPRPf

ADED

Add with Carry to D (A:B)
(CPU12X)

ADED

Operation

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

Description

Adds the content of accumulator A : B to the content of memory location M : M + 1, then adds the value of the C bit and places the result in A : B.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ADED #opr16i	IMM	18 C3 jj kk	OPO
ADED opr8a	DIR	18 D3 dd	ORPf
ADED opr16a	EXT	18 F3 hh ll	ORPO
ADED oprx0_xysp	IDX	18 E3 xb	ORPf
ADED oprx9_xysp	IDX1	18 E3 xb ff	ORPO
ADED oprx16_xysp	IDX2	18 E3 xb ee ff	OfRPP
ADED [D,xysp]	[D,IDX]	18 E3 xb	OfIRPf
ADED [opr16_xysp]	[IDX2]	18 E3 xb ee ff	OfIPRPf

ADEX

Add with Carry to X (CPU12X)

ADEX

Operation

$$(X) + (M : M + 1) + C \Rightarrow X$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

V: $X_{15} \bullet M_{15} \bullet \overline{R_{15}} + \overline{X_{15}} \bullet \overline{M_{15}} \bullet R_{15}$

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ADEX #opr16i	IMM	18 89 jj kk	OPO
ADEX opr8a	DIR	18 99 dd	ORPf
ADEX opr16a	EXT	18 B9 hh ll	ORPO
ADEX oprx0_xysp	IDX	18 A9 xb	ORPf
ADEX oprx9_xysp	IDX1	18 A9 xb ff	ORPO
ADEX oprx16_xysp	IDX2	18 A9 xb ee ff	OfRPP
ADEX [D,xysp]	[D,IDX]	18 A9 xb	OfIFRPf
ADEX [opr16,xysp]	[IDX2]	18 A9 xb ee ff	OfIPRPf

ADEY

Add with Carry to Y
(CPU12X)

ADEY

Operation

$$(Y) + (M : M + 1) + C \Rightarrow Y$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ADEY #opr16i	IMM	18 C9 jj kk	OPO
ADEY opr8a	DIR	18 D9 dd	ORPf
ADEY opr16a	EXT	18 F9 hh ll	ORPO
ADEY oprx0_xysp	IDX	18 E9 xb	ORPf
ADEY oprx9_xysp	IDX1	18 E9 xb ff	ORPO
ADEY oprx16_xysp	IDX2	18 E9 xb ee ff	OFRPP
ADEY [D,xysp]	[D,IDX]	18 E9 xb	OfIFRPf
ADEY [opr16,xysp]	[IDX2]	18 E9 xb ee ff	OfIPRPf

ANDA

Logical AND A
(CPU12, CPU12X)

ANDA

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ANDA #opr8i	IMM	84 ii	P
ANDA opr8a	DIR	94 dd	rPf
ANDA opr16a	EXT	B4 hh ll	rPO
ANDA oprx0_xysp	IDX	A4 xb	rPf
ANDA oprx9_xysp	IDX1	A4 xb ff	rPO
ANDA oprx16_xysp	IDX2	A4 xb ee ff	frPP
ANDA [D,xysp]	[D,IDX]	A4 xb	fIf rPf
ANDA [opr16,xysp]	[IDX2]	A4 xb ee ff	fIP rPf

ANDB

Logical AND B
(CPU12, CPU12X)

ANDB

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ANDB #opr8i	IMM	C4 ii	P
ANDB opr8a	DIR	D4 dd	rPf
ANDB opr16a	EXT	F4 hh ll	rPO
ANDB oprx0_xysp	IDX	E4 xb	rPf
ANDB oprx9_xysp	IDX1	E4 xb ff	rPO
ANDB oprx16_xysp	IDX2	E4 xb ee ff	frPP
ANDB [D,xysp]	[D,IDX]	E4 xb	fIfrPf
ANDB [opr16_xysp]	[IDX2]	E4 xb ee ff	fIfrPf

ANDCC

Logical AND CCR with Mask
(CPU12, CPU12X)

ANDCC

Operation

$(CCR) \bullet (Mask) \Rightarrow CCR$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C	
↓	↓	↓	↓	↓	↓	↓	↓	supervisor state
–	–	↓	–	↓	↓	↓	↓	user state

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ANDCC #opr8i	IMM	10 ii	all P

ANDX

Logical AND X
(CPU12X)

ANDX

Operation

$$(X) \bullet (M : M + 1) \Rightarrow X$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ANDX #opr16i	IMM	18 84 jj kk	OPO
ANDX opr8a	DIR	18 94 dd	ORPf
ANDX opr16a	EXT	18 B4 hh ll	ORPO
ANDX oprx0_xysp	IDX	18 A4 xb	ORPf
ANDX oprx9_xysp	IDX1	18 A4 xb ff	ORPO
ANDX oprx16_xysp	IDX2	18 A4 xb ee ff	OFRPP
ANDX [D,xysp]	[D,IDX]	18 A4 xb	OfIFRPf
ANDX [opr16_xysp]	[IDX2]	18 A4 xb ee ff	OfIPRPf

ANDY

Logical AND Y
(CPU12X)

ANDY

Operation

$$(Y) \bullet (M : M + 1) \Rightarrow Y$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

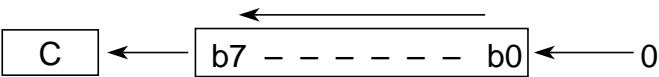
Source Form	Address Mode	Object Code	Access Detail	CPU12X
ANDY #opr16i	IMM	18 C4 jj kk		OPO
ANDY opr8a	DIR	18 D4 dd		ORPf
ANDY opr16a	EXT	18 F4 hh ll		ORPO
ANDY oprx0_xysp	IDX	18 E4 xb		ORPf
ANDY oprx9_xysp	IDX1	18 E4 xb ff		ORPO
ANDY oprx16_xysp	IDX2	18 E4 xb ee ff		OFRPP
ANDY [D,xysp]	[D,IDX]	18 E4 xb		OfIFRPf
ANDY [opr16,xysp]	[IDX2]	18 E4 xb ee ff		OfIPRPf

ASL

Arithmetic Shift Left Memory (same as LSL)
(CPU12, CPU12X)

ASL

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: M7
Set if the MSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

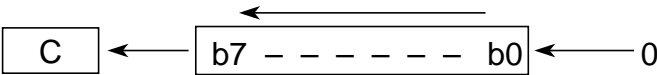
Source Form	Address Mode	Object Code	Access Detail
ASL <i>opr16a</i>	EXT	78 hh ll	rPwO
ASL <i>opr0_xysp</i>	IDX	68 xb	rPw
ASL <i>opr9,xysp</i>	IDX1	68 xb ff	rPwO
ASL <i>opr16,xysp</i>	IDX2	68 xb ee ff	frPwP
ASL [D,xysp]	[D,IDX]	68 xb	fIfrrPw
ASL [<i>opr16,xysp</i>]	[IDX2]	68 xb ee ff	fIPrrPw

ASLA

Arithmetic Shift Left A (same as LSLA)
(CPU12, CPU12X)

ASLA

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: A7
Set if the MSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

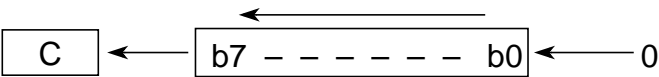
Source Form	Address Mode	Object Code	Access Detail
ASLA	INH	48	<div>all</div> <div>0</div>

ASLB

Arithmetic Shift Left B (same as LSLB)
(CPU12, CPU12X)

ASLB

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: B7
Set if the MSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

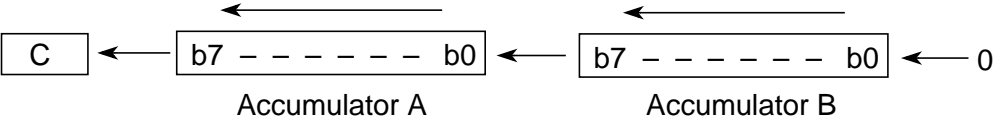
Source Form	Address Mode	Object Code	Access Detail
ASLB	INH	58	<div>all</div> <div>0</div>

ASLD

Arithmetic Shift Left Double Accumulator
(same as LSLD)
(CPU12, CPU12X)

ASLD

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$0000; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: D15
Set if the MSB of D was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

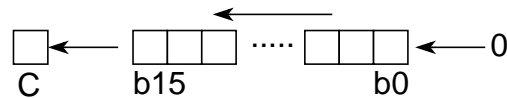
Source Form	Address Mode	Object Code	Access Detail
ASLD	INH	59	<div>all</div> <div>0</div>

ASLW

Arithmetic Shift Left Memory (16 Bit)
(CPU12X)

ASLW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of W.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

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

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

C: M15

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

Detailed Syntax and Cycle-by-Cycle Operation

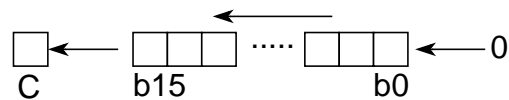
Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ASLW <i>opr16a</i>	EXT	18 78 hh ll	ORPWO
ASLW <i>opr0_xysp</i>	IDX	18 68 xb	ORPW
ASLW <i>opr9,xysp</i>	IDX1	18 68 xb ff	ORPWO
ASLW <i>opr16,xysp</i>	IDX2	18 68 xb ee ff	OfRPWP
ASLW [D,xysp]	[D,IDX]	18 68 xb	OfIfRPW
ASLW [<i>opr16,xysp</i>]	[IDX2]	18 68 xb ee ff	OfIPRPW

ASLX

Arithmetic Shift Left Index Register X (CPU12X)

ASLX

Operation



Description

Shifts all bits of index register X one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of X.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

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

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

C: X15

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

Detailed Syntax and Cycle-by-Cycle Operation

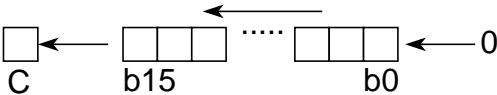
Source Form	Address Mode	Object Code	Access Detail
ASLX	INH	18 48	CPU12X 00

ASLY

Arithmetic Shift Left Index Register Y (CPU12X)

ASLY

Operation



Description

Shifts all bits of index register Y one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of Y.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

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

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

C: Y15

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

Detailed Syntax and Cycle-by-Cycle Operation

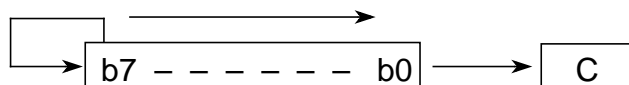
Source Form	Address Mode	Object Code	Access Detail
ASLY	INH	18 58	CPU12X 00

ASR

Arithmetic Shift Right Memory (CPU12, CPU12X)

ASR

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

C: M0

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

Detailed Syntax and Cycle-by-Cycle Operation

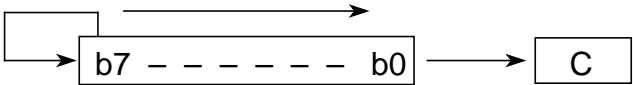
Source Form	Address Mode	Object Code	Access Detail
ASR <i>opr16a</i>	EXT	77 hh ll	rPwO
ASR <i>opr0_xysp</i>	IDX	67 xb	rPw
ASR <i>opr9_xysp</i>	IDX1	67 xb ff	rPwO
ASR <i>opr16_xysp</i>	IDX2	67 xb ee ff	frPwP
ASR [D, <i>xysp</i>]	[D,IDX]	67 xb	fIfPw
ASR [<i>opr16_xysp</i>]	[IDX2]	67 xb ee ff	fIPrPw

ASRA

Arithmetic Shift Right A
(CPU12, CPU12X)

ASRA

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: A0
Set if the LSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

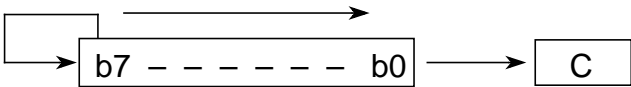
Source Form	Address Mode	Object Code	Access Detail
ASRA	INH	47	<div>all0</div>

ASRB

Arithmetic Shift Right B
(CPU12, CPU12X)

ASRB

Operation



Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

- N: Set if MSB of result is set; cleared otherwise
- Z: Set if result is \$00; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: B0
Set if the LSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

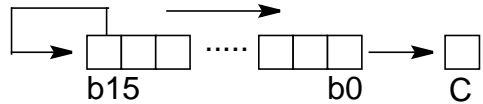
Source Form	Address Mode	Object Code	Access Detail
ASRB	INH	57	<div>all</div> <div>0</div>

ASRW

Arithmetic Shift Right Memory (16 Bit)
(CPU12X)

ASRW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one place to the right. Bit 15 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

C: M0

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

Detailed Syntax and Cycle-by-Cycle Operation

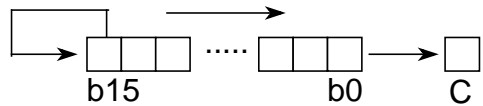
Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ASRW <i>opr16a</i>	EXT	18 77 hh ll	ORPWO
ASRW <i>opr0_xysp</i>	IDX	18 67 xb	ORPW
ASRW <i>opr9_xysp</i>	IDX1	18 67 xb ff	ORPWO
ASRW <i>opr16_xysp</i>	IDX2	18 67 xb ee ff	OfRPWP
ASRW [D, <i>xysp</i>]	[D,IDX]	18 67 xb	OfIFRPW
ASRW [<i>opr16_xysp</i>]	[IDX2]	18 67 xb ee ff	OfIPRPW

ASRX

Arithmetic Shift Right Index Register X (CPU12X)

ASRX

Operation



Description

Shifts all bits of index register X one place to the right. Bit 15 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

C: X0

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

Detailed Syntax and Cycle-by-Cycle Operation

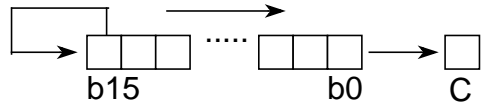
Source Form	Address Mode	Object Code	Access Detail
ASRX	INH	18 47	CPU12X 00

ASRY

Arithmetic Shift Right Index Register Y (CPU12X)

ASRY

Operation



Description

Shifts all bits of index register Y one place to the right. Bit 15 is held constant. Bit 0 is loaded into the C status bit. This operation effectively divides a two's complement value by two without changing its sign. The carry bit can be used to round the result.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

C: Y0

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ASRY	INH	18 57	CPU12X 00

BCC

Branch if Carry Cleared (Same as BHS)
(CPU12, CPU12X)

BCC

Operation

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

Simple branch

Description

Tests the C status bit and branches if $C = 0$.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BCC <i>rel8</i>	REL	24 rr	all PPP/P ¹

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

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

BCLR

Clear Bits in Memory
(CPU12, CPU12X)

BCLR

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode ¹	Object Code	Access Detail
BCLR <i>opr8a, msk8</i>	DIR	4D dd mm	rPwO
BCLR <i>opr16a, msk8</i>	EXT	1D hh ll mm	rPwP
BCLR <i>opr0_xysp, msk8</i>	IDX	0D xb mm	rPwO
BCLR <i>opr9_xysp, msk8</i>	IDX1	0D xb ff mm	rPwP
BCLR <i>opr16_xysp, msk8</i>	IDX2	0D xb ee ff mm	frPwPO

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

BCS

Branch if Carry Set (Same as BLO)
(CPU12, CPU12X)

BCS

Operation

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

Simple branch

Description

Tests the C status bit and branches if $C = 1$.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BCS <i>rel8</i>	REL	25 rr	all PPP/P ¹

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

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

BEQ

Branch if Equal
(CPU12, CPU12X)

BEQ

Operation

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

Simple branch

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BEQ <i>rel8</i>	REL	27 rr	all PPP/P ¹

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

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

BGE

Branch if Greater than or Equal to Zero
(CPU12, CPU12X)

BGE

Operation

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

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BGE <i>rel8</i>	REL	2C rr	all PPP/P ¹

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

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

BGND

Enter Background Debug Mode (CPU12, CPU12X)

BGND

Description

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

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

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

BDM is normally disabled to avoid accidental entry. While BDM is disabled, BGND executes as described, but the firmware causes execution to return to the user program. Refer to [Chapter 4, “Instruction Queue”](#) for more information concerning BDM.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BGND	INH	00	all VfPPP

BGT

Branch if Greater than Zero (CPU12, CPU12X)

BGT

Operation

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

For signed two's complement values

if (Accumulator) > (Memory), then branch

Description

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

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

CCR Details

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BGT <i>rel8</i>	REL	2E rr	all PPP/P ¹

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

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

BHI

Branch if Higher
(CPU12, CPU12X)

BHI

Operation

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

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BHI <i>rel8</i>	REL	22 rr	all PPP/P ¹

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

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

BHS

Branch if Higher or Same (Same as BCC)
(CPU12, CPU12X)

BHS

Operation

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

For unsigned values, if $(\text{Accumulator}) \geq (\text{Memory})$, then branch

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BHS <i>rel8</i>	REL	24 <i>rr</i>	all PPP/P ¹

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

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

BITA

Bit Test A
(CPU12, CPU12X)

BITA

Operation

(A) • (M)

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
BITA #opr8i	IMM	85 ii	P
BITA opr8a	DIR	95 dd	rPf
BITA opr16a	EXT	B5 hh ll	rPO
BITA oprx0_xysp	IDX	A5 xb	rPf
BITA oprx9_xysp	IDX1	A5 xb ff	rPO
BITA oprx16_xysp	IDX2	A5 xb ee ff	frPP
BITA [D,xysp]	[D,IDX]	A5 xb	fIfPrPf
BITA [opr16,xysp]	[IDX2]	A5 xb ee ff	fIPrPf

BITB

Bit Test B (CPU12, CPU12X)

BITB

Operation

(B) • (M)

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BITB #opr8i	IMM	C5 ii	P
BITB opr8a	DIR	D5 dd	rPf
BITB opr16a	EXT	F5 hh ll	rPO
BITB oprx0_xysp	IDX	E5 xb	rPf
BITB oprx9_xysp	IDX1	E5 xb ff	rPO
BITB oprx16_xysp	IDX2	E5 xb ee ff	frPP
BITB [D,xysp]	[D,IDX]	E5 xb	fIfrPf
BITB [opr16,xysp]	[IDX2]	E5 xb ee ff	fIPrPf

BITX

Bit Test X
(CPU12X)

BITX

Operation

$$(X) \bullet (M : M + 1)$$

Description

Performs bitwise logical AND on the content of index register X and the content of memory location M : M + 1 and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the index register and the memory location. Neither the content of the index register nor the content of the memory location is affected.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
BITX #opr16i	IMM	18 85 jj kk	OPO
BITX opr8a	DIR	18 95 dd	ORP _f
BITX opr16a	EXT	18 B5 hh ll	ORPO
BITX oprx0_xysp	IDX	18 A5 xb	ORP _f
BITX oprx9_xysp	IDX1	18 A5 xb ff	ORPO
BITX oprx16_xysp	IDX2	18 A5 xb ee ff	OFRPP
BITX [D,xysp]	[D,IDX]	18 A5 xb	OfIFRP _f
BITX [opr16,xysp]	[IDX2]	18 A5 xb ee ff	OfIPRP _f

BITY

Bit Test Y
(CPU12X)

BITY

Operation

$(Y) \bullet (M : M + 1)$

Description

Performs bitwise logical AND on the content of index register Y and the content of memory location M : M + 1 and modifies the condition codes accordingly. Each bit of the result is the logical AND of the corresponding bits of the index register and the memory location. Neither the content of the index register nor the content of the memory location is affected.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
BITY #opr16i	IMM	18 C5 jj kk	OPO
BITY opr8a	DIR	18 D5 dd	ORPf
BITY opr16a	EXT	18 F5 hh ll	ORPO
BITY oprx0_xysp	IDX	18 E5 xb	ORPf
BITY oprx9_xysp	IDX1	18 E5 xb ff	ORPO
BITY oprx16_xysp	IDX2	18 E5 xb ee ff	OFRPP
BITY [D,xysp]	[D,IDX]	18 E5 xb	OfIFRPf
BITY [opr16,xysp]	[IDX2]	18 E5 xb ee ff	OfIPRPf

BLE

Branch if Less Than or Equal to Zero
(CPU12, CPU12X)

BLE

Operation

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

For signed two's complement numbers

if $(\text{Accumulator}) \leq (\text{Memory})$, then branch

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BLE <i>rel</i> 8	REL	2F <i>rr</i>	all PPP/P ¹

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

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

BLO

Branch if Lower (Same as BCS)
(CPU12, CPU12X)

BLO

Operation

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

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BLO <i>relB</i>	REL	25 rr	all PPP/P ¹

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

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

BLS

Branch if Lower or Same
(CPU12, CPU12X)

BLS

Operation

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

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BLS <i>rel8</i>	REL	23 <i>rr</i>	all PPP/P ¹

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

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

BLT

Branch if Less than Zero (CPU12, CPU12X)

BLT

Operation

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

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BLT <i>rel</i> /8	REL	2D <i>rr</i>	all PPP/P ¹

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

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

BMI

Branch if Minus
(CPU12, CPU12X)

BMI

Operation

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

Simple branch

Description

Tests the N status bit and branches if $N = 1$.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BMI <i>rel8</i>	REL	2B <i>rr</i>	all PPP/P ¹

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

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

BNE

Branch if Not Equal to Zero
(CPU12, CPU12X)

BNE

Operation

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

Simple branch

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BNE <i>rel8</i>	REL	26 <i>rr</i>	all PPP/P ¹

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

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

BPL

Branch if Plus
(CPU12, CPU12X)

BPL

Operation

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

Simple branch

Description

Tests the N status bit and branches if $N = 0$.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BPL <i>rel/8</i>	REL	2A <i>rr</i>	all PPP/P ¹

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

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

BRA

Branch Always
(CPU12, CPU12X)

BRA

Operation

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

Description

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

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BRA <i>rel8</i>	REL	20 <i>rr</i>	<div>all</div> <div>PPP</div>

BRCLR

Branch if Bits Cleared
(CPU12, CPU12X)

BRCLR

Operation

If (M) • (Mask) = 0, then branch

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode ¹	Object Code	Access Detail
BRCLR <i>opr8a, msk8, rel8</i>	DIR	4F dd mm rr	rPPP
BRCLR <i>opr16a, msk8, rel8</i>	EXT	1F hh ll mm rr	rfPPP
BRCLR <i>opr0_xysp, msk8, rel8</i>	IDX	0F xb mm rr	rPPP
BRCLR <i>opr9_xysp, msk8, rel8</i>	IDX1	0F xb ff mm rr	rfPPP
BRCLR <i>opr16_xysp, msk8, rel8</i>	IDX2	0F xb ee ff mm rr	PrfPPP

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

BRN

Branch Never
(CPU12, CPU12X)

BRN

Operation

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

Description

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

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BRN <i>rel8</i>	REL	21 rr	all P

BRSET

Branch if Bits Set
(CPU12, CPU12X)

BRSET

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode ¹	Object Code	Access Detail
BRSET <i>opr8a, msk8, rel8</i>	DIR	4E dd mm rr	rPPP
BRSET <i>opr16a, msk8, rel8</i>	EXT	1E hh ll mm rr	rfPPP
BRSET <i>opr0_xysp, msk8, rel8</i>	IDX	0E xb mm rr	rPPP
BRSET <i>opr9_xysp, msk8, rel8</i>	IDX1	0E xb ff mm rr	rfPPP
BRSET <i>opr16_xysp, msk8, rel8</i>	IDX2	0E xb ee ff mm rr	PrfPPP

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

BSET

Set Bit(s) in Memory
(CPU12, CPU12X)

BSET

Operation

$$(M) \mid (\text{Mask}) \Rightarrow M$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode ¹	Object Code	Access Detail
BSET <i>opr8a, msk8</i>	DIR	4C dd mm	rPwO
BSET <i>opr16a, msk8</i>	EXT	1C hh ll mm	rPwP
BSET <i>opr0_xysp, msk8</i>	IDX	0C xb mm	rPwO
BSET <i>opr9_xysp, msk8</i>	IDX1	0C xb ff mm	rPwP
BSET <i>opr16_xysp, msk8</i>	IDX2	0C xb ee ff mm	frPwPO

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

BSR

Branch to Subroutine
(CPU12, CPU12X)

BSR

Operation

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

Description

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

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

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

Branches to a location determined by the branch offset.

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

CCR Details

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BSR <i>rel8</i>	REL	07 <i>rr</i>	all SPPP

BTAS

Bit(s) Test and Set in Memory (CPU12X)

BTAS

Operation

If $(M) \bullet (\text{Mask}) = 0$, then set Z, else clear Z

$(M) | (\text{Mask}) \Rightarrow M$

Description

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

NOTE

The CCR bits are affected by the test operation, $(M) \text{ AND } (\text{Mask})$, and not the result operation, $(M) \text{ OR } (\text{Mask})$.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode ¹	Object Code	Access Detail CPU12X
BTAS <i>opr8, msk8</i>	DIR	18 35 dd mm	OrPwO
BTAS <i>opr16a, msk8</i>	EXT	18 36 hh ll mm	OrPwP
BTAS <i>opr_x0_xysp, msk8</i>	IDX	18 37 xb mm	OrPwO
BTAS <i>opr_x9_xysp, msk8</i>	IDX1	18 37 xb ff mm	OrPwP
BTAS <i>opr_x16_xysp, msk8</i>	IDX2	18 37 xb ee ff mm	OfPwPO

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

BVC

Branch if Overflow Cleared (CPU12, CPU12X)

BVC

Operation

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

Simple branch

Description

Tests the V status bit and branches if $V = 0$.

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BVC <i>rel8</i>	REL	28 <i>rr</i>	all PPP/P ¹

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

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

BVS

Branch if Overflow Set (CPU12, CPU12X)

BVS

Operation

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

Simple branch

Description

Tests the V status bit and branches if $V = 1$.

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
BVS <i>rel8</i>	REL	29 <i>rr</i>	all PPP/P ¹

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

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

CALL

Call Subroutine in Expanded Memory (CPU12, CPU12X)

CALL

Operation

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

$(SP) - \$0001 \Rightarrow SP$; $(PPAGE) \Rightarrow M_{(SP)}$

page $\Rightarrow PPAGE$; Subroutine Address $\Rightarrow PC$

Description

Sets up conditions to return to normal program flow, then transfers control to a subroutine in expanded memory. Uses the address of the instruction following the CALL as a return address. For code compatibility, CALL also executes correctly in devices that do not have expanded memory capability. Decrements the SP by two, then stores the return address on the stack. The SP points to the high-order byte of the return address.

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

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

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

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

CCR Details

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CALL <i>opr16a, page</i>	EXT	4A hh ll pg	gnSsPPP
CALL <i>opr0_xysp, page</i>	IDX	4B xb pg	gnSsPPP
CALL <i>opr9_xysp, page</i>	IDX1	4B xb ff pg	gnSsPPP
CALL <i>opr16_xysp, page</i>	IDX2	4B xb ee ff pg	fgnSsPPP
CALL <i>[D,xysp]</i>	[D,IDX]	4B xb	fIignSsPPP
CALL <i>[opr16,xysp]</i>	[IDX2]	4B xb ee ff	fIignSsPPP

CBA

Compare Accumulators (CPU12, CPU12X)

CBA

Operation

$$(A) - (B)$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CBA	INH	18 17	all OO

CLC

Clear Carry
(CPU12, CPU12X)

CLC

Operation

0 \Rightarrow C bit

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	0

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLC <i>translates to...</i> ANDCC #\$FE	IMM	10 FE	all P

CLI

Clear Interrupt Mask (CPU12, CPU12X)

CLI

Operation

0 \Rightarrow I bit

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C	
–	–	–	0	–	–	–	–	supervisor state
–	–	–	–	–	–	–	–	user state

I: 0; cleared (supervisor mode only)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLI <i>translates to...</i> ANDCC #\$EF	IMM	10 EF	all P

CLR

Clear Memory
(CPU12, CPU12X)

CLR

Operation

$0 \Rightarrow M$

Description

All bits in memory location M are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CLR <i>opr16a</i>	EXT	79 hh ll	PwO
CLR <i>opr0_xysp</i>	IDX	69 xb	Pw
CLR <i>opr9,xysp</i>	IDX1	69 xb ff	PwO
CLR <i>opr16,xysp</i>	IDX2	69 xb ee ff	PwP
CLR [D, <i>xysp</i>]	[D,IDX]	69 xb	PIfw
CLR [<i>opr16,xysp</i>]	[IDX2]	69 xb ee ff	PIPw

CLRA

Clear A
(CPU12, CPU12X)

CLRA

Operation

$0 \Rightarrow A$

Description

All bits in accumulator A are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLRA	INH	87	all 0

CLRB

Clear B
(CPU12, CPU12X)

CLRB

Operation

$0 \Rightarrow B$

Description

All bits in accumulator B are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLRB	INH	C7	<div>all0</div>

CLR W

Clear Memory (16 Bit)
(CPU12X)

CLR W

Operation

$0 \Rightarrow M : M + 1$

Description

All bits in memory location $M : M + 1$ are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
CLR W <i>opr16a</i>	EXT	18 79 hh 11	OPWO
CLR W <i>opr0_xysp</i>	IDX	18 69 xb	OPW
CLR W <i>opr9_xysp</i>	IDX1	18 69 xb ff	OPWO
CLR W <i>opr16_xysp</i>	IDX2	18 69 xb ee ff	OPWP
CLR W [D, <i>xysp</i>]	[D, IDX]	18 69 xb	OPIfW
CLR W [<i>opr16_xysp</i>]	[IDX2]	18 69 xb ee ff	OPIPW

CLR X

Clear Index Register X
(CPU12X)

CLR X

Operation

0 ⇒ X

Description

All bits in index register X are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLR X	INH	18 87	CPU12X 00

CLRY

Clear Index Register Y
(CPU12X)

CLRY

Operation

$0 \Rightarrow Y$

Description

All bits in index register Y are cleared to 0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	1	0	0

N: 0; cleared

Z: 1; set

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLRY	INH	18 C7	CPU12X 00

CLV

Clear Two's Complement Overflow Bit
(CPU12, CPU12X)

CLV

Operation

0 ⇒ V bit

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	0	–

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
CLV <i>translates to...</i> ANDCC #\$FD	IMM	10 FD	all P

CMPA

Compare A
(CPU12, CPU12X)

CMPA

Operation

$$(A) - (M)$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

$$V: A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$$

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

$$C: \overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CMPA #opr8i	IMM	81 ii	P
CMPA opr8a	DIR	91 dd	rPf
CMPA opr16a	EXT	B1 hh ll	rPO
CMPA oprx0_xysp	IDX	A1 xb	rPf
CMPA oprx9_xysp	IDX1	A1 xb ff	rPO
CMPA oprx16_xysp	IDX2	A1 xb ee ff	frPP
CMPA [D,xysp]	[D,IDX]	A1 xb	fIfrrPf
CMPA [opr16,xysp]	[IDX2]	A1 xb ee ff	fIPrrPf

CMPB

Compare B
(CPU12, CPU12X)

CMPB

Operation

$(B) - (M)$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$

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

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CMPB #opr8i	IMM	C1 ii	P
CMPB opr8a	DIR	D1 dd	rPf
CMPB opr16a	EXT	F1 hh ll	rPO
CMPB oprx0_xysp	IDX	E1 xb	rPf
CMPB oprx9_xysp	IDX1	E1 xb ff	rPO
CMPB oprx16_xysp	IDX2	E1 xb ee ff	frPP
CMPB [D,xysp]	[D,IDX]	E1 xb	fIfrPf
CMPB [opr16,xysp]	[IDX2]	E1 xb ee ff	fIPrPf

COM

Complement Memory (CPU12, CPU12X)

COM

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

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

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

V: 0; cleared

C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
COM <i>opr16a</i>	EXT	71 hh ll	rPwO
COM <i>opr0_xysp</i>	IDX	61 xb	rPw
COM <i>opr9_xysp</i>	IDX1	61 xb ff	rPwO
COM <i>opr16_xysp</i>	IDX2	61 xb ee ff	frPwP
COM [D, <i>xysp</i>]	[D,IDX]	61 xb	fIfPw
COM [<i>opr16_xysp</i>]	[IDX2]	61 xb ee ff	fIPrPw

COMA

Complement A
(CPU12, CPU12X)

COMA

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

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

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

V: 0; cleared

C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

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

COMB

Complement B
(CPU12, CPU12X)

COMB

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

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

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

V: 0; cleared

C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
COMB	INH	51	all 0

COMW

Complement Memory (16 Bit)
(CPU12X)

COMW

Operation

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

Description

Replaces the content of memory location $M : M + 1$ with its one's complement. Each bit of $M : M + 1$ is complemented. Immediately after a COM operation on unsigned values, only the BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. After operation on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

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

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

V: 0; cleared

C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	CPU12X
COMW <i>opr16a</i>	EXT	18 71 hh ll		ORPWO
COMW <i>opr0_xysp</i>	IDX	18 61 xb		ORPW
COMW <i>opr9,xysp</i>	IDX1	18 61 xb ff		ORPWO
COMW <i>opr16,xysp</i>	IDX2	18 61 xb ee ff		OfRPWP
COMW [D,xysp]	[D,IDX]	18 61 xb		OfIfRPW
COMW [opr16,xysp]	[IDX2]	18 61 xb ee ff		OfIPRPW

COMX

Complement Index Register X
(CPU12X)

COMX

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

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

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

V: 0; cleared

C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
COMX	INH	18 41	00

COMY

Complement Index Register Y
(CPU12X)

COMY

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	1

- N: Set if MSB of result is set; cleared otherwise
Z: Set if result is \$0000; cleared otherwise
V: 0; cleared
C: 1; set (for M6800 compatibility)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
COMY	INH	18 51	CPU12X 00

CPD

Compare Double Accumulator (CPU12, CPU12X)

CPD

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CPD #opr16i	IMM	8C jj kk	PO
CPD opr8a	DIR	9C dd	RPf
CPD opr16a	EXT	BC hh ll	RPO
CPD oprx0_xysp	IDX	AC xb	RPf
CPD oprx9_xysp	IDX1	AC xb ff	RPO
CPD oprx16_xysp	IDX2	AC xb ee ff	fRPP
CPD [D,xysp]	[D,IDX]	AC xb	fIfRPf
CPD [opr16_xysp]	[IDX2]	AC xb ee ff	fIPRPf

CPED

Compare D to Memory with Borrow (CPU12X)

CPED

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
CPED #opr16i	IMM	18 8C jj kk	OPO
CPED opr8a	DIR	18 9C dd	ORPf
CPED opr16a	EXT	18 BC hh ll	ORPO
CPED oprx0_xysp	IDX	18 AC xb	ORPf
CPED oprx9_xysp	IDX1	18 AC xb ff	ORPO
CPED oprx16_xysp	IDX2	18 AC xb ee ff	OfRPP
CPED [D,xysp]	[D,IDX]	18 AC xb	OfIRPf
CPED [oprx16,xysp]	[IDX2]	18 AC xb ee ff	OfIRPf

CPES

Compare SP to Memory with Borrow (CPU12X)

CPES

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

V: $SP_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{SP_{15}} \bullet M_{15} \bullet R_{15}$

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
CPES #opr16i	IMM	18 8F jj kk	OPO
CPES opr8a	DIR	18 9F dd	ORPf
CPES opr16a	EXT	18 BF hh ll	ORPO
CPES oprx0_xysp	IDX	18 AF xb	ORPf
CPES oprx9_xysp	IDX1	18 AF xb ff	ORPO
CPES oprx16_xysp	IDX2	18 AF xb ee ff	OfRPP
CPES [D,xysp]	[D,IDX]	18 AF xb	OfIfRPf
CPES [opr16,xysp]	[IDX2]	18 AF xb ee ff	OfIPRPf

CPEX

Compare X to Memory with Borrow (CPU12X)

CPEX

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

$$V: X_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{X_{15}} \bullet M_{15} \bullet R_{15}$$

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
CPEX #opr16i	IMM	18 8E jj kk	OPO
CPEX opr8a	DIR	18 9E dd	ORPf
CPEX opr16a	EXT	18 BE hh ll	ORPO
CPEX oprx0_xysp	IDX	18 AE xb	ORPf
CPEX oprx9_xysp	IDX1	18 AE xb ff	ORPO
CPEX oprx16_xysp	IDX2	18 AE xb ee ff	OfRPP
CPEX [D,xysp]	[D,IDX]	18 AE xb	OfIfRPf
CPEX [opr16,xysp]	[IDX2]	18 AE xb ee ff	OfIPRPf

CPEY

Compare Y to Memory with Borrow (CPU12X)

CPEY

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
CPEY #opr16i	IMM	18 8D jj kk	OPO
CPEY opr8a	DIR	18 9D dd	ORPf
CPEY opr16a	EXT	18 BD hh ll	ORPO
CPEY oprx0_xysp	IDX	18 AD xb	ORPf
CPEY oprx9_xysp	IDX1	18 AD xb ff	ORPO
CPEY oprx16_xysp	IDX2	18 AD xb ee ff	OfRPP
CPEY [D,xysp]	[D,IDX]	18 AD xb	OfIfRPf
CPEY [opr16,xysp]	[IDX2]	18 AD xb ee ff	OfIPRPf

CPS

Compare Stack Pointer (CPU12, CPU12X)

CPS

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

V: $SP_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{SP_{15}} \bullet M_{15} \bullet R_{15}$

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CPS #opr16i	IMM	8F jj kk	PO
CPS opr8a	DIR	9F dd	RPf
CPS opr16a	EXT	BF hh ll	RPO
CPS oprx0_xysp	IDX	AF xb	RPf
CPS oprx9_xysp	IDX1	AF xb ff	RPO
CPS oprx16_xysp	IDX2	AF xb ee ff	fRPP
CPS [D,xysp]	[D,IDX]	AF xb	fIfRPf
CPS [oprx16,xysp]	[IDX2]	AF xb ee ff	fIPRPf

CPX

Compare Index Register X (CPU12, CPU12X)

CPX

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

V: $X15 \bullet \overline{M15} \bullet \overline{R15} + \overline{X15} \bullet M15 \bullet R15$

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

C: $\overline{X15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{X15}$

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CPX #opr16i	IMM	8E jj kk	PO
CPX opr8a	DIR	9E dd	RPf
CPX opr16a	EXT	BE hh ll	RPO
CPX oprx0_xysp	IDX	AE xb	RPf
CPX oprx9_xysp	IDX1	AE xb ff	RPO
CPX oprx16_xysp	IDX2	AE xb ee ff	fRPP
CPX [D,xysp]	[D,IDX]	AE xb	fIfRPf
CPX [oprx16,xysp]	[IDX2]	AE xb ee ff	fIPRPf

CPY

Compare Index Register Y (CPU12, CPU12X)

CPY

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
CPY #opr16i	IMM	8D jj kk	PO
CPY opr8a	DIR	9D dd	RPf
CPY opr16a	EXT	BD hh ll	RPO
CPY oprx0_xysp	IDX	AD xb	RPf
CPY oprx9_xysp	IDX1	AD xb ff	RPO
CPY oprx16_xysp	IDX2	AD xb ee ff	fRPP
CPY [D,xysp]	[D,IDX]	AD xb	fIfRPf
CPY [oprx16,xysp]	[IDX2]	AD xb ee ff	fIPRPf

DAA

Decimal Adjust A (CPU12, CPU12X)

DAA

Description

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

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	?	Δ

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

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

V: Undefined

C: Represents BCD carry. See bit table

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DAA	INH	18 07	all of O

DBEQ

Decrement and Branch if Equal to Zero
(CPU12, CPU12X)

DBEQ

Operation

$(\text{Counter}) - 1 \Rightarrow \text{Counter}$

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
DBEQ <i>abdxys, rel9</i>	REL	04 1b rr	all PPP/PPO

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

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

DBNE

Decrement and Branch if Not Equal to Zero
(CPU12, CPU12X)

DBNE

Operation

$(\text{Counter}) - 1 \Rightarrow \text{Counter}$

If $(\text{Counter}) \neq 0$, then $(\text{PC}) + \$0003 + \text{Rel} \Rightarrow \text{PC}$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
DBNE <i>abdxys, rel9</i>	REL	04 1b rr	all PPP/PPO

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

DEC

Decrement Memory (CPU12, CPU12X)

DEC

Operation

$$(M) - \$01 \Rightarrow M$$

Description

Subtract one from the content of memory location M.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DEC <i>opr16a</i>	EXT	73 hh ll	rPwO
DEC <i>opr0_xysp</i>	IDX	63 xb	rPw
DEC <i>opr9,xysp</i>	IDX1	63 xb ff	rPwO
DEC <i>opr16,xysp</i>	IDX2	63 xb ee ff	frPwP
DEC [D, <i>xysp</i>]	[D,IDX]	63 xb	fIfPw
DEC [<i>opr16,xysp</i>]	[IDX2]	63 xb ee ff	fIPrPw

DECA

Decrement A
(CPU12, CPU12X)

DECA

Operation

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

Description

Subtract one from the content of accumulator A.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DECA	INH	43	all 0

DECB

Decrement B
(CPU12, CPU12X)

DECB

Operation

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

Description

Subtract one from the content of accumulator B.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DECB	INH	53	<div>all</div> <div>0</div>

DECW

Decrement Memory (16 Bit)
(CPU12X)

DECW

Operation

$$(M : M + 1) - \$0001 \Rightarrow M : M + 1$$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

V: Set if there was a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if $(M : M + 1)$ was \$8000 before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
DECW <i>opr16a</i>	EXT	18 73 hh 11	ORPWO
DECW <i>opr0_xysp</i>	IDX	18 63 xb	ORPW
DECW <i>opr9_xysp</i>	IDX1	18 63 xb ff	ORPWO
DECW <i>opr16_xysp</i>	IDX2	18 63 xb ee ff	OfRPWP
DECW <i>[D,xysp]</i>	[D,IDX]	18 63 xb	OfIFRPW
DECW <i>[opr16,xysp]</i>	[IDX2]	18 63 xb ee ff18	OfIPRPW

DECX

Decrement Index Register X
(CPU12X)

DECX

Operation

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

Description

Subtract one from the content of index register X.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

V: Set if there was a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (X) was \$8000 before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DECX	INH	18 43	CPU12X 00

DECY

Decrement Index Register Y (CPU12X)

DECY

Operation

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

Description

Subtract one from the content of index register Y.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

V: Set if there was a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (Y) was \$8000 before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
DECY	INH	18 53	00

DES

Decrement Stack Pointer (CPU12, CPU12X)

DES

Operation

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

Description

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

CCR Details

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DES <i>translates to...</i> LEAS -1,SP	IDX	1B 9F	<div>all</div> <div>Pf</div>

DEX

Decrement Index Register X
(CPU12, CPU12X)

DEX

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	Δ	-	-

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DEX	INH	09	<div>all0</div>

DEY

Decrement Index Register Y
(CPU12, CPU12X)

DEY

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	Δ	-	-

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
DEY	INH	03	<div>all</div> <div>0</div>

EDIV

Extended Divide 32-Bit by 16-Bit (Unsigned) (CPU12, CPU12X)

EDIV

Operation

$(Y : D) \div (X) \Rightarrow Y$; Remainder $\Rightarrow D$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

C: Set if divisor was \$0000; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
EDIV	INH	11	all fffffffffffo

EDIVS

Extended Divide 32-Bit by 16-Bit (Signed)
(CPU12, CPU12X)

EDIVS

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
EDIVS	INH	18 14	all 0fffffffff0

EMACS

Extended Multiply and Accumulate
16-Bit by 16-Bit to 32-Bit (Signed)
(CPU12, CPU12X)

EMACS

Operation

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

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

Z: Set if result is \$00000000; cleared otherwise

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

Set if result > \$7FFFFFFF (+ overflow) or < \$80000000 (– underflow)

Indicates two's complement overflow

C: $M15 \bullet I15 + I15 \bullet \overline{R15} + \overline{M15} \bullet R15$

Set if there was a carry from bit 15 of the result; cleared otherwise

Indicates a carry from low word to high word of the result occurred

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Address Mode	Object Code	Access Detail		
			CPU12X	CPU12V1	CPU12V0
EMACS <i>opr16a</i>	Special	18 12 hh 11	ORRORRWWP	ORROfRRfWWP	ORROfffRRfWWP

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

EMAXD

Place Larger of Two Unsigned 16-Bit Values in Accumulator D (CPU12, CPU12X)

EMAXD

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
EMAXD <i>oprxd0_xyysp</i>	IDX	18 1A xb	ORPf
EMAXD <i>oprxd9_xyysp</i>	IDX1	18 1A xb ff	ORPO
EMAXD <i>oprxd16_xyysp</i>	IDX2	18 1A xb ee ff	OfRPP
EMAXD <i>[D,xyysp]</i>	[D,IDX]	18 1A xb	OfIfRPPf
EMAXD <i>[oprxd16,xyysp]</i>	[IDX2]	18 1A xb ee ff	OfIfRPPf

EMAXM

Place Larger of Two
Unsigned 16-Bit Values in Memory
(CPU12, CPU12X)

EMAXM

Operation

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

Description

Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger, and leaves the larger of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in D has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
EMAXM <i>opr_x0_xysp</i>	IDX	18 1E xb	all ORPW
EMAXM <i>opr_x9_xysp</i>	IDX1	18 1E xb ff	ORPWO
EMAXM <i>opr_x16_xysp</i>	IDX2	18 1E xb ee ff	OfRPWP
EMAXM [<i>D_xysp</i>]	[D,IDX]	18 1E xb	OfIfRPW
EMAXM [<i>opr_x16_xysp</i>]	[IDX2]	18 1E xb ee ff	OfIPRPW

EMIND

Place Smaller of Two Unsigned 16-Bit Values in Accumulator D (CPU12, CPU12X)

EMIND

Operation

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

Description

Subtracts an unsigned 16-bit value in memory from an unsigned 16-bit value in double accumulator D to determine which is larger, and leaves the smaller of the two values in D. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in D has been replaced by the value in memory.

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
EMIND <i>opr0_xysp</i>	IDX	18 1B xb	ORPf
EMIND <i>opr9_xysp</i>	IDX1	18 1B xb ff	ORPO
EMIND <i>opr16_xysp</i>	IDX2	18 1B xb ee ff	OfRPP
EMIND [D,xysp]	[D,IDX]	18 1B xb	OfIfRPf
EMIND [opr16,xysp]	[IDX2]	18 1B xb ee ff	OfIPRPf

EMINM

Place Smaller of Two
Unsigned 16-Bit Values in Memory
(CPU12, CPU12X)

EMINM

Operation

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

Description

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

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
EMINM <i>opr0_xysp</i>	IDX	18 1F xb	ORPW
EMINM <i>opr9_xysp</i>	IDX1	18 1F xb ff	ORPWO
EMINM <i>opr16_xysp</i>	IDX2	18 1F xb ee ff	OFRPWP
EMINM [D, <i>xysp</i>]	[D, IDX]	18 1F xb	OFRPWP
EMINM [<i>opr16_xysp</i>]	[IDX2]	18 1F xb ee ff	OFRPWP

EMUL

Extended Multiply 16-Bit by 16-Bit (Unsigned)
(CPU12, CPU12X)

EMUL

Operation

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

Description

An unsigned 16-bit value is multiplied by an unsigned 16-bit value to produce an unsigned 32-bit result. The first source operand must be loaded into 16-bit double accumulator D and the second source operand must be loaded into index register Y before executing the instruction. When the instruction is executed, the value in D is multiplied by the value in Y. The upper 16-bits of the 32-bit result are stored in Y and the low-order 16-bits of the result are stored in D.

The C status bit can be used to round the high-order 16 bits of the result.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

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

Z: Set if result is \$00000000; cleared otherwise

C: Set if bit 15 of the result is set; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
EMUL	INH	13	0	ff0

EMULS

Extended Multiply 16-Bit by 16-Bit (Signed) (CPU12, CPU12X)

EMULS

Operation

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

Description

A signed 16-bit value is multiplied by a signed 16-bit value to produce a signed 32-bit result. The first source operand must be loaded into 16-bit double accumulator D, and the second source operand must be loaded into index register Y before executing the instruction. When the instruction is executed, D is multiplied by the value Y. The 16 high-order bits of the 32-bit result are stored in Y and the 16 low-order bits of the result are stored in D.

The C status bit can be used to round the high-order 16 bits of the result.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ

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

Z: Set if result is \$00000000; cleared otherwise

C: Set if bit 15 of the result is set; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
EMULS	INH	18 13	OfO	OfO

EORA

Exclusive OR A
(CPU12, CPU12X)

EORA

Operation

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

Description

Performs the logical exclusive OR between the content of accumulator A and the content of memory location M. The result is placed in A. Each bit of A after the operation is the logical exclusive OR of the corresponding bits of M and A before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
EORA #opr8i	IMM	88 ii	P
EORA opr8a	DIR	98 dd	rPf
EORA opr16a	EXT	B8 hh ll	rPO
EORA oprx0_xysp	IDX	A8 xb	rPf
EORA oprx9_xysp	IDX1	A8 xb ff	rPO
EORA oprx16_xysp	IDX2	A8 xb ee ff	frPP
EORA [D,xysp]	[D,IDX]	A8 xb	fIfrrPf
EORA [opr16,xysp]	[IDX2]	A8 xb ee ff	fIPrrPf

EORB

Exclusive OR B
(CPU12, CPU12X)

EORB

Operation

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

Description

Performs the logical exclusive OR between the content of accumulator B and the content of memory location M. The result is placed in A. Each bit of A after the operation is the logical exclusive OR of the corresponding bits of M and B before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
EORB #opr8i	IMM	C8 ii	P
EORB opr8a	DIR	D8 dd	rPf
EORB opr16a	EXT	F8 hh ll	rPO
EORB oprx0_xysp	IDX	E8 xb	rPf
EORB oprx9_xysp	IDX1	E8 xb ff	rPO
EORB oprx16_xysp	IDX2	E8 xb ee ff	frPP
EORB [D,xysp]	[D,IDX]	E8 xb	fIfrPf
EORB [opr16,xysp]	[IDX2]	E8 xb ee ff	fIPrPf

EORX

Exclusive OR X
(CPU12X)

EORX

Operation

$$(X) \oplus (M : M + 1) \Rightarrow X$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
EORX #opr16i	IMM	18 88 jj kk	OPO
EORX opr8a	DIR	18 98 dd	ORPpf
EORX opr16a	EXT	18 B8 hh ll	ORPO
EORX oprx0_xysp	IDX	18 A8 xb	ORPpf
EORX oprx9_xysp	IDX1	18 A8 xb ff	ORPO
EORX oprx16_xysp	IDX2	18 A8 xb ee ff	OfRPP
EORX [D,xysp]	[D,IDX]	18 A8 xb	OfIfRPPf
EORX [opr16,xysp]	[IDX2]	18 A8 xb ee ff	OfIPRPf

EORY

Exclusive OR Y
(CPU12X)

EORY

Operation

$$(Y) \oplus (M : M + 1) \Rightarrow Y$$

Description

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
EORY #opr16i	IMM	18 C8 jj kk	OPO
EORY opr8a	DIR	18 D8 dd	ORPf
EORY opr16a	EXT	18 F8 hh ll	ORPO
EORY oprx0_xysp	IDX	18 E8 xb	ORPf
EORY oprx9_xysp	IDX1	18 E8 xb ff	ORPO
EORY oprx16_xysp	IDX2	18 E8 xb ee ff	OfRPP
EORY [D,xysp]	[D,IDX]	18 E8 xb	OfIfRPf
EORY [opr16,xysp]	[IDX2]	18 E8 xb ee ff	OfIPRPf

Operation

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

Description

ETBL linearly interpolates one of 256 result values that fall between each pair of data entries in a lookup table stored in memory. Data entries in the table represent the y values of endpoints of equally-spaced line segments. Table entries and the interpolated result are 16-bit values. The result is stored in the D accumulator.

Before executing ETBL, an index register points to the table entry corresponding to the x value (X1 that is closest to, but less than or equal to, the desired lookup point (XL, YL). This defines the left end of a line segment and the right end is defined by the next data entry in the table. Prior to execution, accumulator B holds a binary fraction (radix left of MSB) representing the ratio of $(XL - X1) \div (X2 - X1)$.

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

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

Where:

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

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

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

The intermediate value $[(B) \times (Y2 - Y1)]$ produces a 24-bit result with the radix point between bits 7 and 8. Any indexed addressing mode, except indirect modes or 9-bit and 16-bit offset modes, can be used to identify the first data point (X1,Y1). The second data point is the next table entry.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	–	Δ ¹

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

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

C: Set if result can be rounded up; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
ETBL <i>opr0_xysp</i>	IDX	18 3F xb	ORRffffP	ORRffffffP

EXG

Exchange Register Contents (CPU12, CPU12X)

EXG

Operation

See table

Description

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

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C	
—	—	—	—	—	—	—	—	—	—	—	—	—	—	
OR														
↑	—	—	—	—	Δ	Δ	↓	Δ	Δ	Δ	Δ	Δ	Δ	supervisor state
—	—	—	—	—	—	—	—	Δ	—	Δ	Δ	Δ	Δ	user state

None affected, unless the CCR (or CCRL, CCRH, CCRW) is the destination register. Condition codes take on the value of the corresponding source bits, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only in response to any reset or by recognition of an $\overline{\text{XIRQ}}$ interrupt.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
EXG <i>abcdxys,abcdxys</i>	INH	B7 eb	all P

¹ Legal coding for eb is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit. Values are in hexadecimal.

EXG

Exchange Register Contents (Continued) (CPU12, CPU12X)

EXG

	MS⇒	8	9	A	B	C	D	E	F
↓ LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR _L ⇔ A EXG CCR,A EXG CCR _L ,A	TMP3 _L ⇔ A \$00:A ⇒ TMP3 EXG A, TMP3	B ⇔ A EXG D,A	X _L ⇔ A \$00:A ⇒ X EXG X,A	Y _L ⇔ A \$00:A ⇒ Y EXG Y,A	SP _L ⇔ A \$00:A ⇒ SP EXG SP,A
1	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR _L ⇔ B EXG CCR,B EXG CCR _L ,B	TMP3 _L ⇔ B \$FF:B ⇒ TMP3 EXG B, TMP3	B ⇔ B \$FF ⇒ A EXG D,B	X _L ⇔ B \$FF:B ⇒ X EXG X,B	Y _L ⇔ B \$FF:B ⇒ Y EXG Y,B	SP _L ⇔ B \$FF:B ⇒ SP EXG SP,B
2	CCR	A ⇔ CCR _L EXG A, CCR EXG A,CCR _L	B ⇔ CCR _L EXG B,CCR EXG B,CCR _L	CCR _L ⇔ CCR _L EXG CCR,CCR EXG CCR _L ,CCR _L	TMP3 _L ⇔ CCR _L \$FF:CCR _L ⇒ TMP3 EXG, TMP3,CCR EXG TMP3,CCR _L	B ⇔ CCR _L \$FF:CCR _L ⇒ D EXG D,CCR EXG D,CCR _L	X _L ⇔ CCR _L \$FF:CCR _L ⇒ X EXG X,CCR EXG X,CCR _L	Y _L ⇔ CCR _L \$FF:CCR _L ⇒ Y EXG Y,CCR EXG Y,CCR _L	SP _L ⇔ CCR _L \$FF:CCR _L ⇒ SP EXG SP,CCR EXG SP,CCR _L
3	TMP2	\$00:A ⇒ TMP2 TMP2 _L ⇔ A EXG A, TMP2	\$00:B ⇒ TMP2 TMP2 _L ⇔ B EXG B, TMP2	\$00:CCR _L ⇒ TMP2 TMP2 _L ⇔ CCR EXG CCR, TMP2	TMP3 ⇔ TMP2 EXG TMP3, TMP2	D ⇔ TMP2 EXG D, TMP2	X ⇔ TMP2 EXG X, TMP2	Y ⇔ TMP2 EXG Y, TMP2	SP ⇔ TMP2 EXG SP, TMP2
4	D	\$00:A ⇒ D EXG A,D	\$00:B ⇒ D EXG B,D	\$00:CCR _L ⇒ D B ⇔ CCR _L EXG CCR,D EXG CCR _L ,D	TMP3 ⇔ D EXG TMP3,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
5	X	\$00:A ⇒ X X _L ⇔ A EXG A,X	\$00:B ⇒ X X _L ⇔ B EXG B,X	\$00:CCR _L ⇒ X X _L ⇔ CCR _L EXG CCR,X EXG CCR _L ,X	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
6	Y	\$00:A ⇒ Y Y _L ⇔ A EXG A,Y	\$00:B ⇒ Y Y _L ⇔ B EXG B,Y	\$00:CCR _L ⇒ Y Y _L ⇔ CCR _L EXG CCR,Y EXG CCR _L ,Y	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
7	SP	\$00:A ⇒ SP SP _L ⇔ A EXG A,SP	\$00:B ⇒ SP SP _L ⇔ B EXG B,SP	\$00:CCR _L ⇒ SP SP _L ⇔ CCR _L EXG CCR,SP EXG CCR _L ,SP	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP
8	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR _H ⇔ A EXG CCR _H ,A	TMP3 _H ⇔ A EXG TMP3 _H ,A	B ⇔ A EXG D,A	X _H ⇔ A EXG X _H ,A	Y _H ⇔ A EXG Y _H ,A	SP _H ⇔ A EXG SP _H ,A
9	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR _L ⇔ B EXG CCR _L ,B	TMP3 _L ⇔ B EXG TMP3 _L ,B	\$FF ⇒ A, B ⇒ B EXG D,B	X _L ⇔ B EXG X _L ,B	Y _L ⇔ B EXG Y _L ,B	SP _L ⇔ B EXG SP _L ,B
A	CCR	A ⇔ CCR _H EXG A,CCR _H	B ⇔ CCR _L EXG B,CCR _L	CCR _{H:L} ⇔ CCR _{H:L} EXG CCR _H ,CCR _L	TMP3 ⇔ CCR _{H:L} EXG TMP3,CCR _H	D ⇔ CCR _{H:L} EXG D,CCR _H	X ⇔ CCR _{H:L} EXG X,CCR _H	Y ⇔ CCR _{H:L} EXG Y,CCR _H	SP ⇔ CCR _{H:L} EXG SP,CCR _H
B	TMPx	A ⇔ TMP2 _H EXG A,TMP2 _H	B ⇔ TMP2 _L EXG B,TMP2 _L	CCR _{H:L} ⇔ TMP2 EXG CCR _H ,TMP2	TMP3 ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP1 EXG D,TMP1	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
C	D	\$00:A ⇒ D EXG A,D	\$00:B ⇒ D EXG B,D	CCR _{H:L} ⇔ D EXG CCR _H ,D	TMP1 ⇔ D EXG TMP1,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
D	X	A ⇔ X _H EXG A,X _H	B ⇔ X _L EXG B,X _L	CCR _{H:L} ⇔ X EXG CCR _H ,X	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
E	Y	A ⇔ Y _H EXG A,Y _H	B ⇔ Y _L EXG B,Y _L	CCR _{H:L} ⇔ Y EXG CCR _H ,Y	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
F	SP	A ⇔ SP _H EXG A,SP _H	B ⇔ SP _L EXG B,SP _L	CCR _{H:L} ⇔ SP EXG CCR _H ,SP	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP

Note: Encodings in the shaded area (LS = [8, F]) are only available on the CPU12X.

FDIV

Fractional Divide
(CPU12, CPU12X)

FDIV

Operation

$(D) \div (X) \Rightarrow X$; Remainder $\Rightarrow D$

Description

Divides an unsigned 16-bit numerator in double accumulator D by an unsigned 16-bit denominator in index register X, producing an unsigned 16-bit quotient in X and an unsigned 16-bit remainder in D. If both the numerator and the denominator are assumed to have radix points in the same positions, the radix point of the quotient is to the left of bit 15. The numerator must be less than the denominator. In the case of overflow (denominator is less than or equal to the numerator) or division by zero, the quotient is set to \$FFFF, and the remainder is indeterminate.

FDIV is equivalent to multiplying the numerator by 2^{16} and then performing 32 by 16-bit integer division. The result is interpreted as a binary-weighted fraction, which resulted from the division of a 16-bit integer by a larger 16-bit integer. A result of \$0001 corresponds to 0.000015, and \$FFFF corresponds to 0.9998. The remainder of an IDIV instruction can be resolved into a binary-weighted fraction by an FDIV instruction. The remainder of an FDIV instruction can be resolved into the next 16 bits of binary-weighted fraction by another FDIV instruction.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	Δ	Δ

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

V: 1 if $X \leq D$

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

C: $\overline{X15} \bullet \overline{X14} \bullet \overline{X13} \bullet \overline{X12} \bullet \dots \bullet \overline{X3} \bullet \overline{X2} \bullet \overline{X1} \bullet \overline{X0}$

Set if denominator was \$0000; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
FDIV	INH	18 11	all 0ffffffffff0

GLDAA

Load Accumulator A
from Global Memory
(CPU12X)

GLDAA

Operation

$G(M) \Rightarrow A$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GLDAA <i>opr8a</i>	DIR	18 96 dd	OrPf
GLDAA <i>opr16a</i>	EXT	18 B6 hh ll	OrPO
GLDAA <i>opr0_xysp</i>	IDX	18 A6 xb	OrPf
GLDAA <i>opr9_xysp</i>	IDX1	18 A6 xb ff	OrPO
GLDAA <i>opr16_xysp</i>	IDX2	18 A6 xb ee ff	OfrPP
GLDAA [D, <i>xysp</i>]	[D,IDX]	18 A6 xb	OfIfrPf
GLDAA [<i>opr16_xysp</i>]	[IDX2]	18 A6 xb ee ff	OfIfrPf

GLDAB

Load Accumulator B from Global Memory (CPU12X)

GLDAB

Operation

$G(M) \Rightarrow B$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
GLDAB <i>opr8a</i>	DIR	18 D6 dd	OrPf
GLDAB <i>opr16a</i>	EXT	18 F6 hh ll	OrPO
GLDAB <i>opr0_xysp</i>	IDX	18 E6 xb	OrPf
GLDAB <i>opr9_xysp</i>	IDX1	18 E6 xb ff	OrPO
GLDAB <i>opr16_xysp</i>	IDX2	18 E6 xb ee ff	OfrPP
GLDAB [D, <i>xysp</i>]	[D,IDX]	18 E6 xb	OfIfrrPf
GLDAB [<i>opr16_xysp</i>]	[IDX2]	18 E6 xb ee ff	OfIrrPf

GLDD

Load Double Accumulator D (A:B) from Global Memory (CPU12X)

GLDD

Operation

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GLDD <i>opr8a</i>	DIR	18 DC dd	ORPf
GLDD <i>opr16a</i>	EXT	18 FC hh ll	ORPO
GLDD <i>opr0_xysp</i>	IDX	18 EC xb	ORPf
GLDD <i>opr9_xysp</i>	IDX1	18 EC xb ff	ORPO
GLDD <i>opr16_xysp</i>	IDX2	18 EC xb ee ff	OFRPP
GLDD [<i>D</i> , <i>xysp</i>]	[D,IDX]	18 EC xb	OFRPf
GLDD [<i>opr16</i> , <i>xysp</i>]	[IDX2]	18 EC xb ee ff	OFRPPf

Operation

$$G(M : M + 1) \Rightarrow SP$$
Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
GLDS <i>opr8a</i>	DIR	18 DF dd	ORPf
GLDS <i>opr16a</i>	EXT	18 FF hh ll	ORPO
GLDS <i>opr0_xysp</i>	IDX	18 EF xb	ORPf
GLDS <i>opr9_xysp</i>	IDX1	18 EF xb ff	ORPO
GLDS <i>opr16_xysp</i>	IDX2	18 EF xb ee ff	OfRPP
GLDS <i>[D,xysp]</i>	[D,IDX]	18 EF xb	OfIfRPf
GLDS <i>[opr16,xysp]</i>	[IDX2]	18 EF xb ee ff	OfIPRPf

GLDX

Load Stack Index Register X from Global Memory (CPU12X)

GLDX

Operation

$G(M : M + 1) \Rightarrow X$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GLDX <i>opr8a</i>	DIR	18 DE dd	ORPf
GLDX <i>opr16a</i>	EXT	18 FE hh ll	ORPO
GLDX <i>opr0_xysp</i>	IDX	18 EE xb	ORPf
GLDX <i>opr9_xysp</i>	IDX1	18 EE xb ff	ORPO
GLDX <i>opr16_xysp</i>	IDX2	18 EE xb ee ff	OfRPP
GLDX [D, <i>xysp</i>]	[D,IDX]	18 EE xb	OfIfRPf
GLDX [<i>opr16_xysp</i>]	[IDX2]	18 EE xb ee ff	OfIPRPf

GLDY

Load Stack Index Register Y from Global Memory (CPU12X)

GLDY

Operation

$G(M : M + 1) \Rightarrow Y$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GLDY <i>opr8a</i>	DIR	18 DD dd	ORPf
GLDY <i>opr16a</i>	EXT	18 FD hh ll	ORPO
GLDY <i>opr0_xysp</i>	IDX	18 ED xb	ORPf
GLDY <i>opr9_xysp</i>	IDX1	18 ED xb ff	ORPO
GLDY <i>opr16_xysp</i>	IDX2	18 ED xb ee ff	OfRPP
GLDY [D, <i>xysp</i>]	[D,IDX]	18 ED xb	OfIfRPf
GLDY [<i>opr16_xysp</i>]	[IDX2]	18 ED xb ee ff	OfIPRPf

GSTAA

Store Accumulator A
to Global Memory
(CPU12X)

GSTAA

Operation

(A) \Rightarrow G(M)

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GSTAA <i>opr8a</i>	DIR	18 5A dd	OPw
GSTAA <i>opr16a</i>	EXT	18 7A hh ll	OPwO
GSTAA <i>opr0_xysp</i>	IDX	18 6A xb	OPw
GSTAA <i>opr9_xysp</i>	IDX1	18 6A xb ff	OPwO
GSTAA <i>opr16_xysp</i>	IDX2	18 6A xb ee ff	OPwP
GSTAA [D, <i>xysp</i>]	[D,IDX]	18 6A xb	OPIfw
GSTAA [<i>opr16_xysp</i>]	[IDX2]	18 6A xb ee ff	OPIPw

GSTAB

Store Accumulator B
to Global Memory
(CPU12X)

GSTAB

Operation

(B) \Rightarrow G(M)

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
GSTAB <i>opr8a</i>	DIR	18 5B dd	OPw
GSTAB <i>opr16a</i>	EXT	18 7B hh ll	OPwO
GSTAB <i>opr0_xysp</i>	IDX	18 6B xb	OPw
GSTAB <i>opr9_xysp</i>	IDX1	18 6B xb ff	OPwO
GSTAB <i>opr16_xysp</i>	IDX2	18 6B xb ee ff	OPwP
GSTAB [D, <i>xysp</i>]	[D,IDX]	18 6B xb	OPIfw
GSTAB [<i>opr16_xysp</i>]	[IDX2]	18 6B xb ee ff	OPIPw

GSTD

Store Double Accumulator to Global Memory (CPU12X)

GSTD

Operation

$(A) \Rightarrow G(M), (B) \Rightarrow G(M + 1)$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
GSTD <i>opr8a</i>	DIR	18 5C dd	OPW
GSTD <i>opr16a</i>	EXT	18 7C hh ll	OPWO
GSTD <i>opr0_xysp</i>	IDX	18 6C xb	OPW
GSTD <i>opr9_xysp</i>	IDX1	18 6C xb ff	OPWO
GSTD <i>opr16_xysp</i>	IDX2	18 6C xb ee ff	OPWP
GSTD [D, <i>xysp</i>]	[D,IDX]	18 6C xb	OPIfW
GSTD [<i>opr16_xysp</i>]	[IDX2]	18 6C xb ee ff	OPIPW

Operation

$$(SP) \Rightarrow G(M : M + 1)$$
Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	CPU12X
GSTS <i>opr8a</i>	DIR	18 5F dd		OPW
GSTS <i>opr16a</i>	EXT	18 7F hh ll		OPWO
GSTS <i>opr0_xysp</i>	IDX	18 6F xb		OPW
GSTS <i>opr9_xysp</i>	IDX1	18 6F xb ff		OPWO
GSTS <i>opr16_xysp</i>	IDX2	18 6F xb ee ff		OPWP
GSTS [<i>D,xysp</i>]	[D,IDX]	18 6F xb		OPIEW
GSTS [<i>opr16,xysp</i>]	[IDX2]	18 6F xb ee ff		OPIPW

GSTX

Store Index Register X to Global Memory
(CPU12X)

GSTX

Operation

$(X) \Rightarrow G(M : M + 1)$

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	CPU12X
GSTX <i>opr8a</i>	DIR	18 5E dd		OPW
GSTX <i>opr16a</i>	EXT	18 7E hh ll		OPWO
GSTX <i>opr0_xysp</i>	IDX	18 6E xb		OPW
GSTX <i>opr9_xysp</i>	IDX1	18 6E xb ff		OPWO
GSTX <i>opr16_xysp</i>	IDX2	18 6E xb ee ff		OPWP
GSTX [D, <i>xysp</i>]	[D,IDX]	18 6E xb		OPIEW
GSTX [<i>opr16_xysp</i>]	[IDX2]	18 6E xb ee ff		OPIPW

Operation

$$(Y) \Rightarrow G(M : M + 1)$$
Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

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

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

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
GSTY <i>opr8a</i>	DIR	18 5D dd	OPW
GSTY <i>opr16a</i>	EXT	18 7D hh ll	OPWO
GSTY <i>opr0_xysp</i>	IDX	18 6D xb	OPW
GSTY <i>opr9_xysp</i>	IDX1	18 6D xb ff	OPWO
GSTY <i>opr16_xysp</i>	IDX2	18 6D xb ee ff	OPWP
GSTY [D, <i>xysp</i>]	[D, IDX]	18 6D xb	OPIfW
GSTY [<i>opr16_xysp</i>]	[IDX2]	18 6D xb ee ff	OPIPW

IBEQ

Increment and Branch if Equal to Zero
(CPU12, CPU12X)

IBEQ

Operation

$(\text{Counter}) + 1 \Rightarrow \text{Counter}$

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

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
IBEQ <i>abdxys, rel9</i>	REL	04 1b rr	all PPP/PPO

¹ Encoding for 1b is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (IBEQ – 0) or not zero (IBNE – 1) versions, and bit 0 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 1:0 for IBEQ.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	IBEQ A, <i>rel9</i>	04 80 rr	04 90 rr
B	001	IBEQ B, <i>rel9</i>	04 81 rr	04 91 rr
D	100	IBEQ D, <i>rel9</i>	04 84 rr	04 94 rr
X	101	IBEQ X, <i>rel9</i>	04 85 rr	04 95 rr
Y	110	IBEQ Y, <i>rel9</i>	04 86 rr	04 96 rr
SP	111	IBEQ SP, <i>rel9</i>	04 87 rr	04 97 rr

IBNE

Increment and Branch if Not Equal to Zero (CPU12, CPU12X)

IBNE

Operation

(Counter) + 1 \Rightarrow Counter

If (Counter) not = 0, then (PC) + \$0003 + Rel \Rightarrow PC

Description

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
IBNE <i>abdxys, rel9</i>	REL	04 1b rr	all PPP/PP0

¹ Encoding for 1b is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (IBEQ – 0) or not zero (IBNE – 1) versions, and bit 0 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 1:0 for IBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	IBNE A, <i>rel9</i>	04 A0 rr	04 B0 rr
B	001	IBNE B, <i>rel9</i>	04 A1 rr	04 B1 rr
D	100	IBNE D, <i>rel9</i>	04 A4 rr	04 B4 rr
X	101	IBNE X, <i>rel9</i>	04 A5 rr	04 B5 rr
Y	110	IBNE Y, <i>rel9</i>	04 A6 rr	04 B6 rr
SP	111	IBNE SP, <i>rel9</i>	04 A7 rr	04 B7 rr

IDIV

Integer Divide (CPU12, CPU12X)

IDIV

Operation

$(D) \div (X) \Rightarrow X$; Remainder $\Rightarrow D$

Description

Divides an unsigned 16-bit dividend in double accumulator D by an unsigned 16-bit divisor in index register X, producing an unsigned 16-bit quotient in X, and an unsigned 16-bit remainder in D. If both the divisor and the dividend are assumed to have radix points in the same positions, the radix point of the quotient is to the right of bit 0. In the case of division by zero, C is set, the quotient is set to \$FFFF, and the remainder is indeterminate.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	0	Δ

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

V: 0; cleared

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
IDIV	INH	18 10	all 0fffffffffff0

IDIVS

Integer Divide (Signed) (CPU12, CPU12X)

IDIVS

Operation

$(D) \div (X) \Rightarrow X$; Remainder $\Rightarrow D$

Description

Performs signed integer division of a signed 16-bit numerator in double accumulator D by a signed 16-bit denominator in index register X, producing a signed 16-bit quotient in X, and a signed 16-bit remainder in D. If division by zero is attempted, the values in D and X are not changed, C is set, and the values of the N, Z, and V status bits are undefined.

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

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

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

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

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

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

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

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

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
IDIVS	INH	18 15	all O f f f f f f f f f f O

INC

Increment Memory (CPU12, CPU12X)

INC

Operation

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

Description

Add one to the content of memory location M.

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

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (M) was \$7F before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
INC <i>opr16a</i>	EXT	72 hh ll	rPwO
INC <i>opr0_xysp</i>	IDX	62 xb	rPw
INC <i>opr9_xysp</i>	IDX1	62 xb ff	rPwO
INC <i>opr16_xysp</i>	IDX2	62 xb ee ff	frPwP
INC [D, <i>xysp</i>]	[D,IDX]	62 xb	fIfPw
INC [<i>opr16,xysp</i>]	[IDX2]	62 xb ee ff	fIPrPw

INCA

Increment A
(CPU12, CPU12X)

INCA

Operation

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

Description

Add one to the content of accumulator A.

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

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

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

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

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (A) was \$7F before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
INCA	INH	42	<div>all</div> <div>O</div>

INCB

Increment B
(CPU12, CPU12X)

INCB

Operation

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

Description

Add one to the content of accumulator B.

The N, Z, and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (B) was \$7F before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
INCB	INH	52	all 0

INCW

Increment Memory (16 Bit) (CPU12X)

INCW

Operation

$$(M : M + 1) + \$0001 \Rightarrow M : M + 1$$

Description

Add one to the content of memory location $M : M + 1$.

The N, Z and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.

Two's complement overflow occurs if and only if $(M : M + 1)$ was \$7FFF before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Sourc Form	Address Mode	Object Code	Access Detail CPU12X
INCW <i>opr16a</i>	EXT	18 72 hh 1l	ORPWO
INCW <i>opr0_xysp</i>	IDX	18 62 xb	ORPW
INCW <i>opr9,xysp</i>	IDX1	18 62 xb ff	ORPWO
INCW <i>opr16,xysp</i>	IDX2	18 62 xb ee ff	OFRPWP
INCW [D,xysp]	[D,IDX]	18 62 xb	OFRFPW
INCW [<i>opr16,xysp</i>]	[IDX2]	18 62 xb ee ff	OFRPWP

INCX

Increment Index Register X (CPU12X)

INCX

Operation

$$(X) + \$0001 \Rightarrow X$$

Description

Add one to the content of index register X.

The N, Z and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (X) was \$7FFF before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
INCX	INH	18 42	00

INCY

Increment Index Register Y (CPU12X)

INCY

Operation

$(Y) + \$0001 \Rightarrow Y$

Description

Add one to the content of index register Y.

The N, Z and V status bits are set or cleared according to the results of the operation. The C status bit is not affected by the operation, thus allowing the INC instruction to be used as a loop counter in multiple-precision computations.

When operating on unsigned values, only BEQ, BNE, LBEQ, and LBNE branches can be expected to perform consistently. When operating on two's complement values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: Set if there is a two's complement overflow as a result of the operation; cleared otherwise.
Two's complement overflow occurs if and only if (Y) was \$7FFF before the operation.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
INCY	INH	18 52	00

INS

Increment Stack Pointer (CPU12, CPU12X)

INS

Operation

$(SP) + \$0001 \Rightarrow SP$

Description

Add one to the stack pointer SP. This instruction is assembled to LEAS 1,SP. The LEAS instruction does not affect condition codes as an INX or INY instruction would.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
INS <i>translates to...</i> LEAS 1,SP	IDX	1B 81	all Pf

INX

Increment Index Register X (CPU12, CPU12X)

INX

Operation

$(X) + \$0001 \Rightarrow X$

Description

Add one to index register X. LEAX 1,X can produce the same result but LEAX does not affect the Z status bit. Although the LEAX instruction is more flexible, INX requires only one byte of object code.

INX operation affects only the Z status bit.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is \$0000; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
INX	INH	08	<div>all</div> <div>O</div>

INY

Increment Index Register Y (CPU12, CPU12X)

INY

Operation

$(Y) + \$0001 \Rightarrow Y$

Description

Add one to index register Y. LEAY 1,Y can produce the same result but LEAY does not affect the Z status bit. Although the LEAY instruction is more flexible, INY requires only one byte of object code.

INY operation affects only the Z status bit.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	Δ	–	–

Z: Set if result is \$0000; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
INY	INH	02	<div>all</div> <div>0</div>

JMP

Jump
(CPU12, CPU12X)

JMP

Operation

Effective Address \Rightarrow PC

Description

Jumps to the instruction stored at the effective address. The effective address is obtained according to the rules for extended or indexed addressing.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
JMP <i>opr16a</i>	EXT	06 hh ll	PPP
JMP <i>opr0_xysp</i>	IDX	05 xb	PPP
JMP <i>opr9,xysp</i>	IDX1	05 xb ff	PPP
JMP <i>opr16,xysp</i>	IDX2	05 xb ee ff	fPPP
JMP [D, <i>xysp</i>]	[D,IDX]	05 xb	fIfPPP
JMP [<i>opr16,xysp</i>]	[IDX2]	05 xb ee ff	fIfPPP

JSR

Jump to Subroutine (CPU12, CPU12X)

JSR

Operation

$(SP) - \$0002 \Rightarrow SP$

$RTN_H : RTN_L \Rightarrow M_{(SP)} : M_{(SP+1)}$

Subroutine Address $\Rightarrow PC$

Description

Sets up conditions to return to normal program flow, then transfers control to a subroutine. Uses the address of the instruction following the JSR as a return address.

Decrements the SP by two to allow the two bytes of the return address to be stacked.

Stacks the return address. The SP points to the high order byte of the return address.

Calculates an effective address according to the rules for extended, direct, or indexed addressing.

Jumps to the location determined by the effective address.

Subroutines are normally terminated with an RTS instruction, which restores the return address from the stack.

For SP relative auto pre/post decrement/increment indexed addressing modes, the effective address of the jump is calculated first, then SP adjustments associated with the stacking operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
JSR <i>opr8a</i>	DIR	17 dd	SPPP
JSR <i>opr16a</i>	EXT	16 hh ll	SPPP
JSR <i>opr0_xysp</i>	IDX	15 xb	PPPS
JSR <i>opr9_xysp</i>	IDX1	15 xb ff	PPPS
JSR <i>opr16_xysp</i>	IDX2	15 xb ee ff	fPPPS
JSR [<i>D,xysp</i>]	[D,IDX]	15 xb	fIfPPPS
JSR [<i>opr16,xysp</i>]	[IDX2]	15 xb ee ff	fIfPPPS

LBCC

Long Branch if Carry Cleared
(Same as LBHS)
(CPU12, CPU12X)

LBCC

Operation

If $C = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the C status bit and branches if $C = 0$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBCC <i>rel16</i>	REL	18 24 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBCS

Long Branch if Carry Set (Same as LBLO)
(CPU12, CPU12X)

LBCS

Operation

If $C = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the C status bit and branches if $C = 1$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBCS <i>rel16</i>	REL	18 25 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBEQ

Long Branch if Equal
(CPU12, CPU12X)

LBEQ

Operation

If $Z = 1$, $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the Z status bit and branches if $Z = 1$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBEQ <i>rel</i> /16	REL	18 27 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBGE

Long Branch if Greater Than or Equal to Zero
(CPU12, CPU12X)

LBGE

Operation

If $N \oplus V = 0$, $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers, if (Accumulator) \geq Memory), then branch

Description

LBGE can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A.

See Section 3.9, "Relative Addressing Mode" for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBGE <i>rel16</i>	REL	18 2C qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBGT

Long Branch if Greater Than Zero
(CPU12, CPU12X)

LBGT

Operation

If $Z + (N \oplus V) = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers, If (Accumulator) > (Memory), then branch

Description

LBGT can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A.

See Section 3.9, "Relative Addressing Mode" for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBGT <i>rel16</i>	REL	18 2E qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBHI

Long Branch if Higher (CPU12, CPU12X)

LBHI

Operation

If $C + Z = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if (Accumulator) > (Memory), then branch

Description

LBHI can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than the value in M. After CBA or SBA, the branch occurs if the value in B is greater than the value in A. LBHI should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBHI <i>rel16</i>	REL	18 22 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBHS

Long Branch if Higher or Same
(Same as LBCC)
(CPU12, CPU12X)

LBHS

Operation

If $C = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if $(\text{Accumulator}) \geq (\text{Memory})$, then branch

Description

LBHS can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is greater than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is greater than or equal to the value in A. LBHS should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See Section 3.9, “Relative Addressing Mode” for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBHS <i>rel16</i>	REL	18 24 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLE

Long Branch if Less Than or Equal to Zero (CPU12, CPU12X)

LBLE

Operation

If $Z + (N \oplus V) = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers, if (Accumulator) \leq (Memory), then branch.

Description

LBLE can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is less than or equal to the value in A.

See [Section 3.9, "Relative Addressing Mode"](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBLE <i>rel16</i>	REL	18 2F qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLO

Long Branch if Lower (Same as LBSCS)
(CPU12, CPU12X)

LBLO

Operation

If $C = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if (Accumulator) < (Memory), then branch

Description

LBLO can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A. LBLO should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See Section 3.9, “Relative Addressing Mode” for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBLO <i>rel16</i>	REL	18 25 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLS

Long Branch if Lower or Same
(CPU12, CPU12X)

LBLS

Operation

If $C + Z = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

For unsigned binary numbers, if $(Accumulator) \leq (Memory)$, then branch

Description

LBLS can be used to branch after subtracting or comparing unsigned values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than or equal to the value in M. After CBA or SBA, the branch occurs if the value in B is less than or equal to the value in A. LBLS should not be used for branching after instructions that do not affect the C bit, such as increment, decrement, load, store, test, clear, or complement.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBLS <i>rel16</i>	REL	18 23 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBLT

Long Branch if Less Than Zero (CPU12, CPU12X)

LBLT

Operation

If $N \oplus V = 1$, $(PC) + \$0004 + Rel \Rightarrow PC$

For signed two's complement numbers, if (Accumulator) < (Memory), then branch

Description

LBLT can be used to branch after subtracting or comparing signed two's complement values. After CMPA, CMPB, CPD, CPS, CPX, CPY, SBCA, SBCB, SUBA, SUBB, or SUBD, the branch occurs if the CPU register value is less than the value in M. After CBA or SBA, the branch occurs if the value in B is less than the value in A.

See [Section 3.9, "Relative Addressing Mode"](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBLT <i>rel16</i>	REL	18 2D qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBMI

Long Branch if Minus (CPU12, CPU12X)

LBMI

Operation

If $N = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the N status bit and branches if $N = 1$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBMI <i>rel16</i>	REL	18 2B qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBNE

Long Branch if Not Equal to Zero
(CPU12, CPU12X)

LBNE

Operation

If $Z = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the Z status bit and branches if $Z = 0$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBNE <i>rel16</i>	REL	18 26 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBPL

Long Branch if Plus
(CPU12, CPU12X)

LBPL

Operation

If $N = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the N status bit and branches if $N = 0$.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBPL <i>rel16</i>	REL	18 2A qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBRA

Long Branch Always
(CPU12, CPU12X)

LBRA

Operation

$(PC) + \$0004 + Rel \Rightarrow PC$

Description

Unconditional branch to an address calculated as shown in the expression. Rel is a relative offset stored as a two's complement number in the second and third bytes of machine code corresponding to the long branch instruction.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the LBRA branch condition is always satisfied, the branch is always taken, and the instruction queue must always be refilled, so execution time is always the larger value.

See [Section 3.9, “Relative Addressing Mode”](#) for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBRA <i>rel16</i>	REL	18 20 qq rr	all OPPP

LBRN

Long Branch Never
(CPU12, CPU12X)

LBRN

Operation

$(PC) + \$0004 \Rightarrow PC$

Description

Never branches. LBRN is effectively a 4-byte NOP that requires three cycles to execute. LBRN is included in the instruction set to provide a complement to the LBRA instruction. The instruction is useful during program debug, to negate the effect of another branch instruction without disturbing the offset byte. A complement for LBRA is also useful in compiler implementations.

Execution time is longer when a conditional branch is taken than when it is not, because the instruction queue must be refilled before execution resumes at the new address. Since the LBRN branch condition is never satisfied, the branch is never taken, and the queue does not need to be refilled, so execution time is always the smaller value.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBRN <i>rel16</i>	REL	18 21 qq rr	<div>all</div> <div>OPO</div>

LBVC

Long Branch if Overflow Cleared
(CPU12, CPU12X)

LBVC

Operation

If $V = 0$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the V status bit and branches if $V = 0$.

LBVC causes a branch when a previous operation on two's complement binary values does not cause an overflow. That is, when LBVC follows a two's complement operation, a branch occurs when the result of the operation is valid.

See Section 3.9, "Relative Addressing Mode" for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBVC <i>rel16</i>	REL	18 28 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LBVS

Long Branch if Overflow Set
(CPU12, CPU12X)

LBVS

Operation

If $V = 1$, then $(PC) + \$0004 + Rel \Rightarrow PC$

Simple branch

Description

Tests the V status bit and branches if $V = 1$.

LBVS causes a branch when a previous operation on two's complement binary values causes an overflow. That is, when LBVS follows a two's complement operation, a branch occurs when the result of the operation is invalid.

See Section 3.9, "Relative Addressing Mode" for details of branch execution.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LBVS <i>rel16</i>	REL	18 29 qq rr	all OPPP/OPO ¹

¹ OPPP/OPO indicates this instruction takes four cycles to refill the instruction queue if the branch is taken and three cycles if the branch is not taken.

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	LBGT	18 2E	$Z + (N \oplus V) = 0$	$r \leq m$	LBLE	18 2F	Signed
$r \geq m$	LBGE	18 2C	$N \oplus V = 0$	$r < m$	LBLT	18 2D	Signed
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Signed
$r \leq m$	LBLE	18 2F	$Z + (N \oplus V) = 1$	$r > m$	LBGT	18 2E	Signed
$r < m$	LBLT	18 2D	$N \oplus V = 1$	$r \geq m$	LBGE	18 2C	Signed
$r > m$	LBHI	18 22	$C + Z = 0$	$r \leq m$	LBLS	18 23	Unsigned
$r \geq m$	LBHS/LBCC	18 24	$C = 0$	$r < m$	LBLO/LBCS	18 25	Unsigned
$r = m$	LBEQ	18 27	$Z = 1$	$r \neq m$	LBNE	18 26	Unsigned
$r \leq m$	LBLS	18 23	$C + Z = 1$	$r > m$	LBHI	18 22	Unsigned
$r < m$	LBLO/LBCS	18 25	$C = 1$	$r \geq m$	LBHS/LBCC	18 24	Unsigned
Carry	LBCS	18 25	$C = 1$	No Carry	LBCC	18 24	Simple
Negative	LBMI	18 2B	$N = 1$	Plus	LBPL	18 2A	Simple
Overflow	LBVS	18 29	$V = 1$	No Overflow	LBVC	18 28	Simple
$r = 0$	LBEQ	18 27	$Z = 1$	$r \neq 0$	LBNE	18 26	Simple
Always	LBRA	18 20	—	Never	LBRN	18 21	Unconditional

LDAA

Load Accumulator A
(CPU12, CPU12X)

LDAA

Operation

(M) \Rightarrow A

Description

Loads the content of memory location M into accumulator A. The condition codes are set according to the data.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
LDAA #opr8i	IMM	86 ii	P
LDAA opr8a	DIR	96 dd	rPf
LDAA opr16a	EXT	B6 hh ll	rPO
LDAA oprx0_xysp	IDX	A6 xb	rPf
LDAA oprx9_xysp	IDX1	A6 xb ff	rPO
LDAA oprx16_xysp	IDX2	A6 xb ee ff	frPP
LDAA [D,xysp]	[D,IDX]	A6 xb	fIfrPf
LDAA [opr16,xysp]	[IDX2]	A6 xb ee ff	fIPrPf

LDAB

Load Accumulator B
(CPU12, CPU12X)

LDAB

Operation

(M) \Rightarrow B

Description

Loads the content of memory location M into accumulator B. The condition codes are set according to the data.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
LDAB #opr8i	IMM	C6 ii	P
LDAB opr8a	DIR	D6 dd	rPf
LDAB opr16a	EXT	F6 hh ll	rPO
LDAB oprx0_xysp	IDX	E6 xb	rPf
LDAB oprx9_xysp	IDX1	E6 xb ff	rPO
LDAB oprx16_xysp	IDX2	E6 xb ee ff	frPP
LDAB [D,xysp]	[D,IDX]	E6 xb	fIfrPf
LDAB [opr16,xysp]	[IDX2]	E6 xb ee ff	fIPrPf

LDD

Load Double Accumulator (CPU12, CPU12X)

LDD

Operation

$(M : M+1) \Rightarrow A : B$

Description

Loads the contents of memory locations M and M+1 into double accumulator D. The condition codes are set according to the data. The information from M is loaded into accumulator A, and the information from M+1 is loaded into accumulator B.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
LDD #opr16i	IMM	CC jj kk	PO
LDD opr8a	DIR	DC dd	RPf
LDD opr16a	EXT	FC hh ll	RPO
LDD oprx0_xysp	IDX	EC xb	RPf
LDD oprx9_xysp	IDX1	EC xb ff	RPO
LDD oprx16_xysp	IDX2	EC xb ee ff	fRPP
LDD [D,xysp]	[D,IDX]	EC xb	fIFRPf
LDD [opr16,xysp]	[IDX2]	EC xb ee ff	fIPRPf

LDS

Load Stack Pointer (CPU12, CPU12X)

LDS

Operation

$(M : M + 1) \Rightarrow SP$

Description

Loads the most significant byte of the SP with the content of memory location $M : M + 1$, and loads the least significant byte of the SP with the content of the next byte of memory at $M : M + 1$.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LDS #opr16i	IMM	CF jj kk	PO
LDS opr8a	DIR	DF dd	RPf
LDS opr16a	EXT	FF hh ll	RPO
LDS oprx0_xysp	IDX	EF xb	RPf
LDS oprx9_xysp	IDX1	EF xb ff	RPO
LDS oprx16_xysp	IDX2	EF xb ee ff	fRPP
LDS [D,xysp]	[D,IDX]	EF xb	fIfRPf
LDS [opr16_xysp]	[IDX2]	EF xb ee ff	fIPRPf

LDX

Load Index Register X (CPU12, CPU12X)

LDX

Operation

$(M : M + 1) \Rightarrow X$

Description

Loads the most significant byte of index register X with the content of memory location M, and loads the least significant byte of X with the content of the next byte of memory at M + 1.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
LDX #opr16i	IMM	CE jj kk	PO
LDX opr8a	DIR	DE dd	RPf
LDX opr16a	EXT	FE hh ll	RPO
LDX oprx0_xysp	IDX	EE xb	RPf
LDX oprx9_xysp	IDX1	EE xb ff	RPO
LDX oprx16_xysp	IDX2	EE xb ee ff	fRPP
LDX [D,xysp]	[D,IDX]	EE xb	fIfRPf
LDX [opr16,xysp]	[IDX2]	EE xb ee ff	fIPRPf

LDY

Load Index Register Y (CPU12, CPU12X)

LDY

Operation

$(M : M + 1) \Rightarrow Y$

Description

Loads the most significant byte of index register Y with the content of memory location M, and loads the least significant byte of Y with the content of the next memory location at M + 1.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
LDY #opr16i	IMM	CD jj kk	PO
LDY opr8a	DIR	DD dd	RPf
LDY opr16a	EXT	FD hh ll	RPO
LDY oprx0_xysp	IDX	ED xb	RPf
LDY oprx9_xysp	IDX1	ED xb ff	RPO
LDY oprx16_xysp	IDX2	ED xb ee ff	fRPP
LDY [D,xysp]	[D,IDX]	ED xb	fIfRPf
LDY [opr16,xysp]	[IDX2]	ED xb ee ff	fIfRPf

LEAS

Load Stack Pointer with Effective Address (CPU12, CPU12X)

LEAS

Operation

Effective Address \Rightarrow SP

Description

Loads the stack pointer with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See [Section 3.10, “Indexed Addressing Modes”](#) for more details.

LEAS does not alter condition code bits. This allows stack modification without disturbing CCR bits changed by recent arithmetic operations.

Operation is a bit more complex when LEAS is used with auto-increment or auto-decrement operand specifications and the SP is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load index instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAS involves two different index registers and post-increment or post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAS 4,Y+. First S is loaded with the value of Y, then Y is incremented by 4.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12V1, CPU12X	CPU12V0
LEAS <i>opr0_xysp</i>	IDX	1B xb	P	Pf
LEAS <i>opr9,xysp</i>	IDX1	1B xb ff	PO	PO
LEAS <i>opr16,xysp</i>	IDX2	1B xb ee ff	PP	PP

LEAX

Load X with Effective Address (CPU12, CPU12X)

LEAX

Operation

Effective Address \Rightarrow X

Description

Loads index register X with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See [Section 3.10, “Indexed Addressing Modes”](#) for more details.

Operation is a bit more complex when LEAX is used with auto-increment or auto-decrement operand specifications and index register X is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load indexed instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAX involves two different index registers and post-increment and post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAX 4,Y+. First X is loaded with the value of Y, then Y is incremented by 4.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12V1, CPU12X	CPU12V0
LEAX <i>opr0_xysp</i>	IDX	1A xb	P	Pf
LEAX <i>opr9,xysp</i>	IDX1	1A xb ff	PO	PO
LEAX <i>opr16,xysp</i>	IDX2	1A xb ee ff	PP	PP

LEAY

Load Y with Effective Address
(CPU12, CPU12X)

LEAY

Operation

Effective Address \Rightarrow Y

Description

Loads index register Y with an effective address specified by the program. The effective address can be any indexed addressing mode operand address except an indirect address. Indexed addressing mode operand addresses are formed by adding an optional constant supplied by the program or an accumulator value to the current value in X, Y, SP, or PC. See [Section 3.10, “Indexed Addressing Modes”](#) for more details.

Operation is a bit more complex when LEAY is used with auto-increment or auto-decrement operand specifications and index register Y is the referenced index register. The index register is loaded with what would have gone out to the address bus in the case of a load indexed instruction. In the case of a pre-increment or pre-decrement, the modification is made before the index register is loaded. In the case of a post-increment or post-decrement, modification would have taken effect after the address went out on the address bus, so post-modification does not affect the content of the index register.

In the unusual case where LEAY involves two different index registers and post-increment and post-decrement, both index registers are modified as demonstrated by the following example. Consider the instruction LEAY 4,X+. First Y is loaded with the value of X, then X is incremented by 4.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

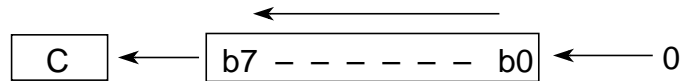
Source Form	Address Mode	Object Code	Access Detail	
			CPU12V1, CPU12X	CPU12V0
LEAY <i>opr0_xysp</i>	IDX	19 xb	P	Pf
LEAY <i>opr9,xysp</i>	IDX1	19 xb ff	PO	PO
LEAY <i>opr16,xysp</i>	IDX2	19 xb ee ff	PP	PP

LSL

Logical Shift Left Memory (Same as ASL) (CPU12, CPU12X)

LSL

Operation



Description

Shifts all bits of the memory location M one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of M.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M7

Set if the LSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LSL <i>opr16a</i>	EXT	78 hh ll	rPwO
LSL <i>opr16_xyysp</i>	IDX	68 xb	rPw
LSL <i>opr9_xyysp</i>	IDX1	68 xb ff	rPwO
LSL <i>opr16_xyysp</i>	IDX2	68 xb ee ff	frPwP
LSL [D, <i>xyysp</i>]	[D,IDX]	68 xb	fIfPrPw
LSL [<i>opr16_xyysp</i>]	[IDX2]	68 xb ee ff	fIPrPw

LSLA

Logical Shift Left A (Same as ASLA)
(CPU12, CPU12X)

LSLA

Operation



Description

Shifts all bits of accumulator A one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of A.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A7

Set if the LSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

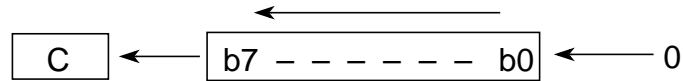
Source Form	Address Mode	Object Code	Access Detail
LSLA	INH	48	all 0

LSLB

Logical Shift Left B (Same as ASLB)
(CPU12, CPU12X)

LSLB

Operation



Description

Shifts all bits of accumulator B one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of B.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B7

Set if the LSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

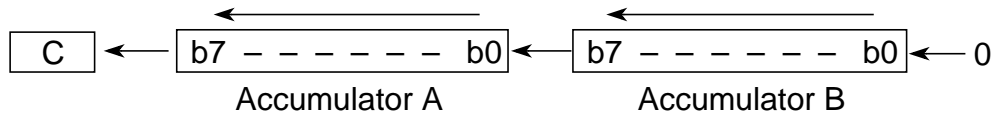
Source Form	Address Mode	Object Code	Access Detail
LSLB	INH	58	all 0

LSLD

Logical Shift Left Double (Same as ASLD)
(CPU12, CPU12X)

LSLD

Operation



Description

Shifts all bits of double accumulator D one place to the left. Bit 0 is loaded with 0. The C status bit is loaded from the most significant bit of accumulator A.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: D15

Set if the MSB of D was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

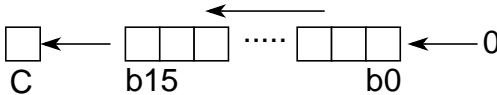
Source Form	Address Mode	Object Code	Access Detail
LSLD	INH	59	<div>all</div> <div>0</div>

LSLW

Logical Shift Left Memory (16 Bit, Same as ASLW) (CPU12X)

LSLW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of W.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M15

Set if the MSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

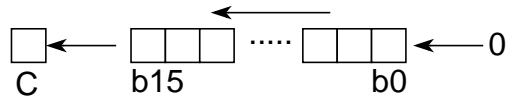
Source Form	Address Mode	Object Code	Access Detail
			CPU12X
LSLW <i>opr16a</i>	EXT	18 78 hh 11	ORPWO
LSLW <i>opr0_xysp</i>	IDX	18 68 xb	ORPW
LSLW <i>opr9_xysp</i>	IDX1	18 68 xb ff	ORPWO
LSLW <i>opr16_xysp</i>	IDX2	18 68 xb ee ff	OfRPWP
LSLW [D, <i>xysp</i>]	[D,IDX]	18 68 xb	OfIFRPW
LSLW [<i>opr16_xysp</i>]	[IDX2]	18 68 xb ee ff	OfIPRPW

LSLX

Logical Shift Left Index Register X (Same as ASLX) (CPU12X)

LSLX

Operation



Description

Shifts all bits of index register X one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of X.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X15

Set if the MSB of X was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

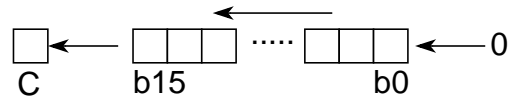
Source Form	Address Mode	Object Code	Access Detail
LSLX	INH	18 48	CPU12X 00

LSLY

Logical Shift Left Index Register Y (Same as ASLY) (CPU12X)

LSLY

Operation



Description

Shifts all bits of index register Y one bit position to the left. Bit 0 is loaded with a 0. The C status bit is loaded from the most significant bit of Y.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y15

Set if the MSB of Y was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

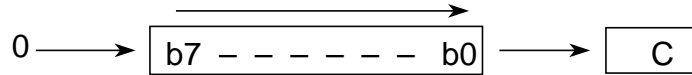
Source Form	Address Mode	Object Code	Access Detail
LSLY	INH	18 58	CPU12X 00

LSR

Logical Shift Right Memory (CPU12, CPU12X)

LSR

Operation



Description

Shifts all bits of memory location M one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of M.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0

Set if the LSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

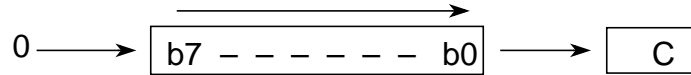
Source Form	Address Mode	Object Code	Access Detail
LSR <i>opr16a</i>	EXT	74 hh ll	rPwO
LSR <i>opr0_xysp</i>	IDX	64 xb	rPw
LSR <i>opr9_xysp</i>	IDX1	64 xb ff	rPwO
LSR <i>opr16_xysp</i>	IDX2	64 xb ee ff	frPwP
LSR [D, <i>xysp</i>]	[D,IDX]	64 xb	fIfrrPw
LSR [<i>opr16_xysp</i>]	[IDX2]	64 xb ee ff	fIPrrPw

LSRA

Logical Shift Right A
(CPU12, CPU12X)

LSRA

Operation



Description

Shifts all bits of accumulator A one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of A.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A0

Set if the LSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

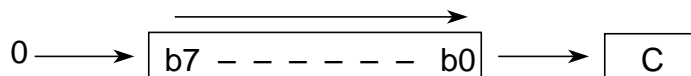
Source Form	Address Mode	Object Code	Access Detail
LSRA	INH	44	all O

LSRB

Logical Shift Right B
(CPU12, CPU12X)

LSRB

Operation



Description

Shifts all bits of accumulator B one place to the right. Bit 7 is loaded with 0. The C status bit is loaded from the least significant bit of B.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B0

Set if the LSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

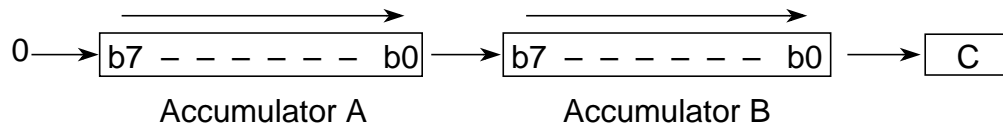
Source Form	Address Mode	Object Code	Access Detail
LSRB	INH	54	all 0

LSRD

Logical Shift Right Double (CPU12, CPU12X)

LSRD

Operation



Description

Shifts all bits of double accumulator D one place to the right. D15 (MSB of A) is loaded with 0. The C status bit is loaded from D0 (LSB of B).

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$0000; cleared otherwise

V: D0

Set if, after the shift operation, C is set; cleared otherwise

C: D0

Set if the LSB of D was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

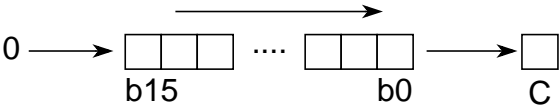
Source Form	Address Mode	Object Code	Access Detail
LSRD	INH	49	all 0

LSRW

Logical Shift Right Memory (16 Bit) (CPU12X)

LSRW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of $M : M + 1$.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	0	Δ	Δ	Δ

- N: 0; cleared
- Z: Set if result is \$0000; cleared otherwise
- V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)
- C: M0
Set if the LSB of $M : M + 1$ was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

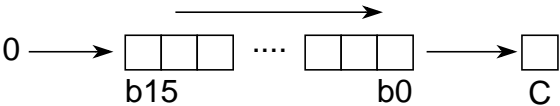
Source Form	Address Mode	Object Code	Access Detail
LSRW <i>opr16a</i>	EXT	18 74 hh 11	ORPWO
LSRW <i>opr0_xysp</i>	IDX	18 64 xb	ORPW
LSRW <i>opr9_xysp</i>	IDX1	18 64 xb ff	ORPWO
LSRW <i>opr16_xysp</i>	IDX2	18 64 xb ee ff	OfRPWP
LSRW [D,xysp]	[D,IDX]	18 64 xb	OfIFRPW
LSRW [<i>opr16_xysp</i>]	[IDX2]	18 64 xb ee ff	OfIFRPW

LSRX

Logical Shift Right Index Register X (CPU12X)

LSRX

Operation



Description

Shifts all bits of index register X one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of X.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X0
Set if the LSB of X was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

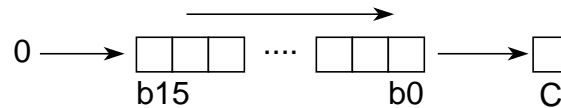
Source Form	Address Mode	Object Code	Access Detail
LSRX	INH	18 44	CPU12X 00

LSRY

Logical Shift Right Index Register Y (CPU12X)

LSRY

Operation



Description

Shifts all bits of index register Y one place to the right. Bit 15 is loaded with 0. The C status bit is loaded from the least significant bit of Y.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	0	Δ	Δ	Δ

N: 0; cleared

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y0

Set if the LSB of Y was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
LSRY	INH	18 54	CPU12X 00

MAXA

Place Larger of Two Unsigned 8-Bit Values
in Accumulator A
(CPU12, CPU12X)

MAXA

Operation

$\text{MAX}((A), (M)) \Rightarrow A$

Description

Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the larger of the two values in A. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in A has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the largest value in a list of values.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
MAXA <i>opr</i> x0_ <i>xy</i> sp	IDX	18 18 xb	OrPf
MAXA <i>opr</i> x9_ <i>xy</i> sp	IDX1	18 18 xb ff	OrPO
MAXA <i>opr</i> x16_ <i>xy</i> sp	IDX2	18 18 xb ee ff	OfPrPP
MAXA [D_ <i>xy</i> sp]	[D,IDX]	18 18 xb	OfIfPrPf
MAXA [<i>opr</i> x16_ <i>xy</i> sp]	[IDX2]	18 18 xb ee ff	OfIPrPf

MAXM

Place Larger of Two Unsigned 8-Bit Values
in Memory
(CPU12, CPU12X)

MAXM

Operation

$\text{MAX}((A), (M)) \Rightarrow M$

Description

Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the larger of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in accumulator A has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
MAXM <i>oprX0_xysp</i>	IDX	18 1C xb	all OrPw
MAXM <i>oprX9_xysp</i>	IDX1	18 1C xb ff	OrPwO
MAXM <i>oprX16_xysp</i>	IDX2	18 1C xb ee ff	OfrPwP
MAXM [D, <i>xysp</i>]	[D,IDX]	18 1C xb	OfIfrPw
MAXM [<i>oprX16_xysp</i>]	[IDX2]	18 1C xb ee ff	OfIPrPw

MEM

Determine Grade of Membership (Fuzzy Logic) (CPU12V0, CPU12XV0)

MEM

Operation

Grade of Membership $\Rightarrow M_{(Y)}$
(Y) + \$0001 $\Rightarrow Y$
(X) + \$0004 $\Rightarrow X$

Description

Before executing MEM, initialize A, X, and Y. Load A with the current crisp value of a system input variable. Load Y with the fuzzy input RAM location where the grade of membership is to be stored. Load X with the first address of a 4-byte data structure that describes a trapezoidal membership function. The data structure consists of:

- Point_1 — The x-axis starting point for the leading side (at M_X)
- Point_2 — The x-axis position of the rightmost point (at M_{X+1})
- Slope_1 — The slope of the leading side (at M_{X+2})
- Slope_2 — The slope of the trailing side (at M_{X+3}); the right side slopes up and to the left from Point_2

A Slope_1 or Slope_2 value of \$00 is a special case in which the membership function either starts with a grade of \$FF at input = Point_1, or ends with a grade of \$FF at input = Point_2 (infinite slope). During execution, the value of A remains unchanged. X is incremented by four and Y is incremented by one.

CCR Details

S	X	H	I	N	Z	V	C
—	—	?	—	?	?	?	?

H, N, Z, V, and C may be altered by this instruction.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12V0, CPU12XV0
MEM	Special	01	RRfOw

Operation

$\text{MIN}((A), (M)) \Rightarrow A$

Description

Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger, and leaves the smaller of the two values in accumulator A. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 0, the value in accumulator A has been replaced by the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand. Auto increment/decrement variations of indexed addressing facilitate finding the smallest value in a list of values.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
MINA <i>opr0_xysp</i>	IDX	18 19 xb	OrPf
MINA <i>opr9_xysp</i>	IDX1	18 19 xb ff	OrPO
MINA <i>opr16_xysp</i>	IDX2	18 19 xb ee ff	OfrPP
MINA [D, <i>xysp</i>]	[D,IDX]	18 19 xb	OfIfrrPf
MINA [<i>opr16_xysp</i>]	[IDX2]	18 19 xb ee ff	OfIPrrPf

MINM

Place Smaller of Two Unsigned 8-Bit Values in Memory (CPU12, CPU12X)

MINM

Operation

$$\text{MIN}((A), (M)) \Rightarrow M$$

Description

Subtracts an unsigned 8-bit value in memory from an unsigned 8-bit value in accumulator A to determine which is larger and leaves the smaller of the two values in the memory location. The Z status bit is set when the result of the subtraction is zero (the values are equal), and the C status bit is set when the subtraction requires a borrow (the value in memory is larger than the value in the accumulator). When C = 1, the value in accumulator A has replaced the value in memory.

The unsigned value in memory is accessed by means of indexed addressing modes, which allow a great deal of flexibility in specifying the address of the operand.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Condition codes reflect internal subtraction ($R = A - M$)

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
MINM <i>opr0_xysp</i>	IDX	18 1D xb	all OrPw
MINM <i>opr9_xysp</i>	IDX1	18 1D xb ff	OrPwO
MINM <i>opr16_xysp</i>	IDX2	18 1D xb ee ff	OfrPwP
MINM [<i>D,xysp</i>]	[D,IDX]	18 1D xb	OfIfrPw
MINM [<i>opr16,xysp</i>]	[IDX2]	18 1D xb ee ff	OfIPrPw

MOVB

Immediate-to-Memory Byte Move (8 Bit)
(CPU12, CPU12X)

MOVB

Operation

⇒ M

Description

Moves the immediate value # to memory location M.

Move byte instructions specify the source first and destination second in the object code for an immediate value source and an extended addressing mode destination. Move byte instructions using immediate values for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for CPU12 and HC12 compatibility.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVB #opr8i, opr16a ¹	EXT	18 0B ii hh ll	OPwP	OPwP
MOVB #opr8i, oprx0_xyysp ¹	IDX	18 08 xb ² ii	OPwO	OPwO
MOVB #opr8i, oprx9_xyysp ¹	IDX1	18 08 xb ² ff ii	OPwP	NA
MOVB #opr8i, oprx16_xyysp ¹	IDX2	18 08 xb ² ee ff ii	OPPwO	NA
MOVB #opr8i, [D_xyysp] ¹	[D,IDX]	18 08 xb ² ii	OPIOw	NA
MOVB #opr8i, [opr16_xyysp] ¹	[IDX2]	18 08 xb ² ee ff ii	OPIOwP	NA

¹ The first operand in the source code statement specifies the source for the move.

² The IDX destination code is listed before the source for backwards compatibility.

MOVB

Memory-to-Memory Byte Move
EXT Source (8 Bit)
(CPU12, CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

EXT Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for an extended addressing mode source and an extended addressing mode destination. Move byte instructions using extended addressing for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for CPU12 and HC12 compatibility.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVB <i>opr16a</i> , <i>opr16a</i> ¹	EXT	18 0C hh 1l hh 1l	OPrPwO	OrPwPO
MOVB <i>opr16a</i> , <i>opr0_xysp</i> ¹	IDX	18 09 xb ² hh 1l	OPrPw	OPrPw
MOVB <i>opr16a</i> , <i>opr9_xysp</i> ¹	IDX1	18 09 xb ² ff hh 1l	OPrPwO	NA
MOVB <i>opr16a</i> , <i>opr16_xysp</i> ¹	IDX2	18 09 xb ² ee ff hh 1l	OPPrPw	NA
MOVB <i>opr16a</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 09 xb ² hh 1l	OPrIPw	NA
MOVB <i>opr16a</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 09 xb ² ee ff hh 1l	OPPrIPw	NA

¹ The first operand in the source code statement specifies the source for the move.

² The IDX destination code is listed before the source for backwards compatibility.

MOVB

Memory-to-Memory Byte Move
IDX Source (8 Bit)
(CPU12, CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

IDX Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVB <i>opr</i> x0_ysp, <i>opr</i> 16a ¹	EXT	18 0D xb hh ll	OrPPw	OrPwP
MOVB <i>opr</i> x0_ysp, <i>opr</i> x0_ysp ¹	IDX	18 0A xb xb	OrPOw	OrPwO
MOVB <i>opr</i> x0_ysp, <i>opr</i> x9_ysp ¹	IDX1	18 0A xb xb ff	OrPPw	NA
MOVB <i>opr</i> x0_ysp, <i>opr</i> x16_ysp ¹	IDX2	18 0A xb xb ee ff	OrPOPw	NA
MOVB <i>opr</i> x0_ysp, [<i>D_ysp</i>] ¹	[D,IDX]	18 0A xb xb	OrPIOw	NA
MOVB <i>opr</i> x0_ysp, [<i>opr</i> x16_ysp] ¹	[IDX2]	18 0A xb xb ee ff	OrPPIOw	NA

¹ The first operand in the source code statement specifies the source for the move.

MOVB

Memory-to-Memory Byte Move
IDX1 Source (8 Bit)
(CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

IDX1 Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVB <i>opr9_xysp</i> , <i>opr16a</i> ¹	EXT	18 0D xb ff hh ll	OPrOPw
MOVB <i>opr9_xysp</i> , <i>opr0_xysp</i> ¹	IDX	18 0A xb ff xb	OPrOOw
MOVB <i>opr9_xysp</i> , <i>opr9_xysp</i> ¹	IDX1	18 0A xb ff xb ff	OPrOPw
MOVB <i>opr9_xysp</i> , <i>opr16_xysp</i> ¹	IDX2	18 0A xb ff xb ee ff	OPrOOPw
MOVB <i>opr9_xysp</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 0A xb ff xb	OPrOIOw
MOVB <i>opr9_xysp</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 0A xb ff xb ee ff	OPrOPIow

¹ The first operand in the source code statement specifies the source for the move.

MOVB

Memory-to-Memory Byte Move
IDX2 Source (8 Bit)
(CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

IDX2 Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVB <i>opr16_xysp</i> , <i>opr16a</i> ¹	EXT	18 0D xb ee ff hh ll	OPrPPw
MOVB <i>opr16_xysp</i> , <i>opr0_xysp</i> ¹	IDX	18 0A xb ee ff xb	OPrPOw
MOVB <i>opr16_xysp</i> , <i>opr9_xysp</i> ¹	IDX1	18 0A xb ee ff xb ff	OPrPPw
MOVB <i>opr16_xysp</i> , <i>opr16_xysp</i> ¹	IDX2	18 0A xb ee ff xb ee ff	OPrPOPw
MOVB <i>opr16_xysp</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 0A xb ee ff xb	OPrPIOw
MOVB <i>opr16_xysp</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 0A xb ee ff xb ee ff	OPrPPIOw

¹ The first operand in the source code statement specifies the source for the move.

MOVB

Memory-to-Memory Byte Move
[D,IDX] Source (8 Bit)
(CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

[D,IDX] Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed. Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVB [D_xysp], opr16a ¹	EXT	18 0D xb hh ll	OIPrfPw
MOVB [D_xysp], oprx0_xysp ¹	IDX	18 0A xb xb	OIPrfOw
MOVB [D_xysp], oprx9_xysp ¹	IDX1	18 0A xb xb ff	OIPrfPw
MOVB [D_xysp], oprx16_xysp ¹	IDX2	18 0A xb xb ee ff	OIPrfOPw
MOVB [D_xysp], [D_xysp] ¹	[D,IDX]	18 0A xb xb	OIPrfIOw
MOVB [D_xysp], [opr16_xysp] ¹	[IDX2]	18 0A xb xb ee ff	OIPrfPIOw

¹ The first operand in the source code statement specifies the source for the move.

MOVB

Memory-to-Memory Byte Move
[IDX2] Source (8 Bit)
(CPU12X)

MOVB

Operation

$(M_1) \Rightarrow M_2$

[IDX2] Source \Rightarrow Address Mode Destination

Description

Moves the content of one 8-bit memory location to another 8-bit memory location. The content of the source memory location is not changed.

Move byte instructions specify the source first and destination second in the object code for all indexed addressing mode sources. For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVB instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVB [opr16_xysp], opr16a ¹	EXT	18 0D xb ee ff hh ll	OPIPrfPw
MOVB [opr16_xysp], oprx0_xysp ¹	IDX	18 0A xb ee ff xb	OPIPrfOw
MOVB [opr16_xysp], oprx9_xysp ¹	IDX1	18 0A xb ee ff xb ff	OPIPrfPw
MOVB [opr16_xysp], oprx16_xysp ¹	IDX2	18 0A xb ee ff xb ee ff	OPIPrfOPw
MOVB [opr16_xysp], [D_xysp] ¹	[D,IDX]	18 0A xb ee ff xb	OPIPrfIOw
MOVB [opr16_xysp], [opr16_xysp] ¹	[IDX2]	18 0A xb ee ff xb ee ff	OPIPrfPIOw

¹ The first operand in the source code statement specifies the source for the move.

MOVW

Immediate-to-Memory Word Move (16 Bit)
(CPU12, CPU12X)

MOVW

Operation

$\# \Rightarrow M : M + 1$

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed.

Move word instructions specify the source first and destination second in the object code for an immediate value source and an extended addressing mode destination. Move word instructions using immediate values for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for CPU12V0 and HC12 compatibility.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVW #opr16i, opr16a ¹	EXT	18 03 jj kk hh ll	OPPWO	OPWPO
MOVW #opr16i, oprx0_xysp ¹	IDX	18 00 xb ² jj kk	OPWP	OPPW
MOVW #opr16i, oprx9_xysp ¹	IDX1	18 00 xb ² ff jj kk	OPPWO	NA
MOVW #opr16i, oprx16_xysp ¹	IDX2	18 00 xb ² ee ff jj kk	OPPWP	NA
MOVW #opr16i, [D_xysp]	[D,IDX]	18 00 xb ² jj kk	OPIPW	NA
MOVW #opr16i, [opr16_xysp] ¹	[IDX2]	18 00 xb ² ee ff jj kk	OPIPWP	NA

¹ The first operand in the source code statement specifies the source for the move.

² The IDX destination code is listed before the source for backwards compatibility.

MOVW

Memory-to-Memory Word Move
EXT Source (16 Bit)
(CPU12, CPU12X)

MOVW

Operation

$(M : M + 1_1) \Rightarrow M : M + 1_2$

EXT Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed.

Move word instructions specify the source first and destination second in the object code for an extended addressing mode source and an extended addressing mode destination. Move word instructions using extended addressing for the source and indexed addressing modes for the destination have the destination index code (xb) specified before the source value for CPU12V0 and HC12 compatibility.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVW <i>opr16a</i> , <i>opr16a</i> ¹	EXT	18 04 hh ₂ 11 hh 11	OPRPWO	ORPWPO
MOVW <i>opr16a</i> , <i>opr0_xysp</i> ¹	IDX	18 01 xb ₂ hh 11	OPRPW	OPRPW
MOVW <i>opr16a</i> , <i>opr9_xysp</i> ¹	IDX1	18 01 xb ₂ ff hh 11	OPRPWO	NA
MOVW <i>opr16a</i> , <i>opr16_xysp</i> ¹	IDX2	18 01 xb ₂ ee ff hh 11	OPPRPW	NA
MOVW <i>opr16a</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 01 xb ₂ hh 11	OPRIPW	NA
MOVW <i>opr16a</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 01 xb ₂ ee ff hh 11	OPPRIPW	NA

¹ The first operand in the source code statement specifies the source for the move.

² The IDX destination code is listed before the source for backwards compatibility.

MOVW

Memory-to-Memory Word Move
IDX Source (16 Bit)
(CPU12, CPU12X)

MOVW

Operation

$(M : M + 1_1) \Rightarrow M : M + 1_2$

IDX Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed.

Move word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
MOVW <i>opr0_xysp, opr16a</i> ¹	EXT	18 05 xb hh ll	ORPPW	ORPWP
MOVW <i>opr0_xysp, opr0_xysp</i> ¹	IDX	18 02 xb xb	ORPOW	ORPWO
MOVW <i>opr0_xysp, opr9_xysp</i> ¹	IDX1	18 02 xb xb ff	ORPPW	NA
MOVW <i>opr0_xysp, opr16_xysp</i> ¹	IDX2	18 02 xb xb ee ff	ORPOPW	NA
MOVW <i>opr0_xysp, [D_xysp]</i> ¹	[D,IDX]	18 02 xb xb	ORPIOW	NA
MOVW <i>opr0_xysp, [opr16_xysp]</i> ¹	[IDX2]	18 02 xb xb ee ff	ORPPIOW	NA

¹ The first operand in the source code statement specifies the source for the move.

MOVW

Memory-to-Memory Word Move
IDX1 Source (16 Bit)
(CPU12X)

MOVW

Operation

$(M : M + 1_1) \Rightarrow M : M + 1_2$

IDX1 Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed. Move word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVW <i>opr9_xysp</i> , <i>opr16a</i> ¹	EXT	18 05 xb ff hh ll	OPROPW
MOVW <i>opr9_xysp</i> , <i>opr0_xysp</i> ¹	IDX	18 02 xb ff xb	OPROOW
MOVW <i>opr9_xysp</i> , <i>opr9_xysp</i> ¹	IDX1	18 02 xb ff xb ff	OPROPW
MOVW <i>opr9_xysp</i> , <i>opr16_xysp</i> ¹	IDX2	18 02 xb ff xb ee ff	OPROOPW
MOVW <i>opr9_xysp</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 02 xb ff xb	OPROIOW
MOVW <i>opr9_xysp</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 02 xb ff xb ee ff	OPROPIOW

¹ The first operand in the source code statement specifies the source for the move.

MOVW

Memory-to-Memory Word Move
IDX2 Source (16 Bit)
(CPU12X)

MOVW

Operation

$$(M : M + 1_1) \Rightarrow M : M + 1_2$$

IDX2 Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed. Move word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVW <i>opr16_xysp</i> , <i>opr16a</i> ¹	EXT	18 05 xb ee ff hh ll	OPRPPW
MOVW <i>opr16_xysp</i> , <i>opr0_xysp</i> ¹	IDX	18 02 xb ee ff xb	OPRPOW
MOVW <i>opr16_xysp</i> , <i>opr9_xysp</i> ¹	IDX1	18 02 xb ee ff xb ff	OPRPPW
MOVW <i>opr16_xysp</i> , <i>opr16_xysp</i> ¹	IDX2	18 02 xb ee ff xb ee ff	OPRPOPW
MOVW <i>opr16_xysp</i> , [<i>D_xysp</i>] ¹	[D,IDX]	18 02 xb ee ff xb	OPRPIOW
MOVW <i>opr16_xysp</i> , [<i>opr16_xysp</i>] ¹	[IDX2]	18 02 xb ee ff xb ee ff	OPRPPIOW

¹ The first operand in the source code statement specifies the source for the move.

MOVW

Memory-to-Memory Word Move
[D,IDX] Source (16 Bit)
(CPU12X)

MOVW

Operation

$(M : M + 1_1) \Rightarrow M : M + 1_2$

[D,IDX] Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed. Move word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVW [D_xysp], opr16a ¹	EXT	18 05 xb hh ll	OIPRfPW
MOVW [D_xysp], oprx0_xysp ¹	IDX	18 02 xb xb	OIPRfOW
MOVW [D_xysp], oprx9_xysp ¹	IDX1	18 02 xb xb ff	OIPRfPW
MOVW [D_xysp], oprx16_xysp ¹	IDX2	18 02 xb xb ee ff	OIPRfOPW
MOVW [D_xysp], [D_xysp] ¹	[D,IDX]	18 02 xb xb	OIPRfIOW
MOVW [D_xysp], [opr16_xysp] ¹	[IDX2]	18 02 xb xb ee ff	OIPRfPIOW

¹ The first operand in the source code statement specifies the source for the move.

MOVW

Memory-to-Memory Word Move
[IDX2] Source (16 Bit)
(CPU12X)

MOVW

Operation

$(M : M + 1_1) \Rightarrow M : M + 1_2$

[IDX2] Source \Rightarrow Address Mode Destination

Description

Moves the content of one 16-bit location in memory to another 16-bit location in memory. The content of the source memory location is not changed. Move word instructions specify the source first and destination second in the object code for all indexed addressing mode sources.

For auto pre/post decrement/increment indexed addressing modes, the effective address of the source is calculated first and the source index register is updated appropriately, then the destination effective address is calculated.

A PC offset must be applied to the source address when using PC relative index addressing for the source operand and any of the three destination index addressing modes listed below:

IDX1: +1

IDX2: +2

[IDX2]: +2

These offsets compensate for the variable instruction length and are needed to identify the location of the instruction immediately following the MOVW instruction.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form ¹	Destination Address Mode	Object Code	Access Detail CPU12X
MOVW [opr16_xysp], opr16a ¹	EXT	18 05 xb ee ff hh ll	OPIPRfPW
MOVW [opr16_xysp], opr0_xysp ¹	IDX	18 02 xb ee ff xb	OPIPRfOW
MOVW [opr16_xysp], oprx9_xysp ¹	IDX1	18 02 xb ee ff xb ff	OPIPRfPW
MOVW [opr16_xysp], oprx16_xysp ¹	IDX2	18 02 xb ee ff xb ee ff	OPIPRfOPW
MOVW [opr16_xysp], [D_xysp] ¹	[D,IDX]	18 02 xb ee ff xb	OPIPRfIOW
MOVW [opr16_xysp], [opr16_xysp]1	[IDX2]	18 02 xb ee ff xb ee ff	OPIPRfPIOW

¹ The first operand in the source code statement specifies the source for the move.

MUL

Multiply 8-Bit by 8-Bit (Unsigned)
(CPU12, CPU12X)

MUL

Operation

$$(A) \times (B) \Rightarrow A : B$$

Description

Multiplies the 8-bit unsigned binary value in accumulator A by the 8-bit unsigned binary value in accumulator B and places the 16-bit unsigned result in double accumulator D. The carry flag allows rounding the most significant byte of the result through the sequence MUL, ADCA #0.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	Δ

C: R7

Set if bit 7 of the result (B bit 7) is set; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
MUL	INH	12	all 0

NEG

Negate Memory (CPU12, CPU12X)

NEG

Operation

$$0 - (M) = (\overline{M}) + 1 \Rightarrow M$$

Description

Replaces the content of memory location M with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise.

Z: Set if result is \$00; cleared otherwise.

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$

Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise. Two's complement overflow occurs if and only if $(M) = \$80$

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$

Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when $(M) = \$00$.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
NEG <i>opr16a</i>	EXT	70 hh 11	rPwO
NEG <i>opr0_xysp</i>	IDX	60 xb	rPw
NEG <i>opr9_xysp</i>	IDX1	60 xb ff	rPwO
NEG <i>opr16_xysp</i>	IDX2	60 xb ee ff	frPwP
NEG [D, <i>xysp</i>]	[D,IDX]	60 xb	fIfPw
NEG [<i>opr16_xysp</i>]	[IDX2]	60 xb ee ff	fIPrPw

NEGA

Negate A
(CPU12, CPU12X)

NEGA

Operation

$$0 - (A) = (\overline{A}) + 1 \Rightarrow A$$

Description

Replaces the content of accumulator A with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$

Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise

Two's complement overflow occurs if and only if $(A) = \$80$

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$

Set if there is a borrow in the implied subtraction from zero; cleared otherwise

Set in all cases except when $(A) = \$00$

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
NEGA	INH	40	all O

NEGB

Negate B
(CPU12, CPU12X)

NEGB

Operation

$$0 - (B) = (\overline{B}) + 1 \Rightarrow B$$

Description

Replaces the content of accumulator B with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $R7 \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$

Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise

Two's complement overflow occurs if and only if $(B) = \$80$

C: $R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$

Set if there is a borrow in the implied subtraction from zero; cleared otherwise

Set in all cases except when $(B) = \$00$

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
NEGB	INH	50	all 0

NEGW Two's Complement Negate Memory (16 Bit) (CPU12X) NEGW

Operation

$0 - (M : M + 1) \Rightarrow M : M + 1$ equivalent to $(\overline{M : M + 1}) + 1 \Rightarrow M : M + 1$

Description

Replaces the content of memory location $M : M + 1$ with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise.

Z: Set if result is \$0000; cleared otherwise.

V: $R15 \bullet \overline{R14} \bullet \overline{R13} \bullet \overline{R12} \bullet \overline{R11} \bullet \overline{R10} \bullet \overline{R9} \bullet \overline{R8} \bullet \overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise. Two's complement overflow occurs if and only if $(M : M + 1) = \$8000$

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when $(M : M + 1) = \$0000$.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
NEGW <i>opr16a</i>	EXT	18 70 hh ll	ORPWO
NEGW <i>opr0_xysp</i>	IDX	18 60 xb	ORPW
NEGW <i>opr9_xysp</i>	IDX1	18 60 xb ff	ORPWO
NEGW <i>opr16_xysp</i>	IDX2	18 60 xb ee ff	OfRPWP
NEGW [D, <i>xysp</i>]	[D,IDX]	18 60 xb	OfIfRPW
NEGW [<i>opr16_xysp</i>]	[IDX2]	18 60 xb ee ff	OfIPRPW

NEGX

Negate Index Register X (CPU12X)

NEGX

Operation

$0 - (X) \Rightarrow X$ equivalent to $(\overline{X}) + 1 \Rightarrow X$

Description

Replaces the content of index register X with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise.

Z: Set if result is \$0000; cleared otherwise.

V: $R15 \bullet \overline{R14} \bullet \overline{R13} \bullet \overline{R12} \bullet \overline{R11} \bullet \overline{R10} \bullet \overline{R9} \bullet \overline{R8} \bullet \overline{R7} \bullet \overline{R6} \bullet \overline{R5} \bullet \overline{R4} \bullet \overline{R3} \bullet \overline{R2} \bullet \overline{R1} \bullet \overline{R0}$
Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise. Two's complement overflow occurs if and only if $(X) = \$8000$

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when $(X) = \$0000$.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
NEGX	INH	18 40	00

NEGY

Negate Index Register Y (CPU12X)

NEGY

Operation

$0 - (Y) \Rightarrow Y$ equivalent to $(\bar{Y}) + 1 \Rightarrow Y$

Description

Replaces the content of index register Y with its two's complement.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise.

Z: Set if result is \$0000; cleared otherwise.

V: $R15 \bullet \bar{R14} \bullet \bar{R13} \bullet \bar{R12} \bullet \bar{R11} \bullet \bar{R10} \bullet \bar{R9} \bullet \bar{R8} \bullet \bar{R7} \bullet \bar{R6} \bullet \bar{R5} \bullet \bar{R4} \bullet \bar{R3} \bullet \bar{R2} \bullet \bar{R1} \bullet \bar{R0}$
Set if there is a two's complement overflow from the implied subtraction from zero; cleared otherwise. Two's complement overflow occurs if and only if $(Y) = \$8000$

C: $R15 + R14 + R13 + R12 + R11 + R10 + R9 + R8 + R7 + R6 + R5 + R4 + R3 + R2 + R1 + R0$
Set if there is a borrow in the implied subtraction from zero; cleared otherwise. Set in all cases except when $(Y) = \$0000$.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
NEGY	INH	18 50	00

NOP

Null Operation
(CPU12, CPU12X)

NOP

Operation

No operation

Description

This single-byte instruction increments the PC and does nothing else. No other CPU12 registers are affected. NOP is typically used to produce a time delay, although some software disciplines discourage CPU12 frequency-based time delays. During debug, NOP instructions are sometimes used to temporarily replace other machine code instructions, thus disabling the replaced instruction(s).

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
NOP	INH	A7	all 0

ORAA

Inclusive OR A
(CPU12, CPU12X)

ORAA

Operation

$$(A) \mid (M) \Rightarrow A$$

Description

Performs bitwise logical inclusive OR between the content of accumulator A and the content of memory location M and places the result in A. Each bit of A after the operation is the logical inclusive OR of the corresponding bits of M and of A before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ORAA #opr8i	IMM	8A ii	P
ORAA opr8a	DIR	9A dd	rPf
ORAA opr16a	EXT	BA hh ll	rPO
ORAA oprx0_xysp	IDX	AA xb	rPf
ORAA oprx9_xysp	IDX1	AA xb ff	rPO
ORAA oprx16_xysp	IDX2	AA xb ee ff	frPP
ORAA [D,xysp]	[D,IDX]	AA xb	fIf rPf
ORAA [opr16,xysp]	[IDX2]	AA xb ee ff	fIP rPf

ORAB

Inclusive OR B
(CPU12, CPU12X)

ORAB

Operation

$(B) \mid (M) \Rightarrow B$

Description

Performs bitwise logical inclusive OR between the content of accumulator B and the content of memory location M. The result is placed in B. Each bit of B after the operation is the logical inclusive OR of the corresponding bits of M and of B before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
ORAB #opr8i	IMM	CA ii	P
ORAB opr8a	DIR	DA dd	rPf
ORAB opr16a	EXT	FA hh ll	rPO
ORAB oprx0_xysp	IDX	EA xb	rPf
ORAB oprx9_xysp	IDX1	EA xb ff	rPO
ORAB oprx16_xysp	IDX2	EA xb ee ff	frPP
ORAB [D,xysp]	[D,IDX]	EA xb	fIfrrPf
ORAB [opr16,xysp]	[IDX2]	EA xb ee ff	fIPrrPf

ORCC

Logical OR CCR with Mask (CPU12, CPU12X)

ORCC

Operation

$$(CCR) \mid (M) \Rightarrow CCR$$

Description

Performs bitwise logical inclusive OR between the content of memory location M and the content of the CCR and places the result in the CCR. Each bit of the CCR after the operation is the logical OR of the corresponding bits of M and of CCR before the operation. To set one or more bits, set the corresponding bit of the mask equal to 1. Bits corresponding to 0s in the mask are not changed by the ORCC operation.

CCR Details

S	X	H	I	N	Z	V	C	
↑↑	—	↑↑	↑↑	↑↑	↑↑	↑↑	↑↑	supervisor state
—	—	↑↑	—	↑↑	↑↑	↑↑	↑↑	user state

Condition code bits are set if the corresponding bit was 1 before the operation or if the corresponding bit in the instruction-provided mask is 1. The X interrupt mask cannot be set by any software instruction.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ORCC #opr8i	IMM	14 ii	all P

ORX

Logic OR X with Memory (CPU12X)

ORX

Operation

$$(X) \mid (M : M + 1) \Rightarrow X$$

Description

Performs bitwise logical inclusive OR between the content of index register X and the content of memory location M : M + 1 and places the result in X. Each bit of X after the operation is the logical inclusive OR of the corresponding bits of M : M + 1 and of X before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
ORX #opr16i	IMM	18 8A jj kk	OPO
ORX opr8a	DIR	18 9A dd	ORPf
ORX opr16a	EXT	18 BA hh ll	ORPO
ORX oprx0_xyyp	IDX	18 AA xb	ORPf
ORX oprx9_xyyp	IDX1	18 AA xb ff	ORPO
ORX oprx16_xyyp	IDX2	18 AA xb ee ff	OFRPP
ORX [D,xyyp]	[D,IDX]	18 AA xb	OfIFRPf
ORX [opr16_xyyp]	[IDX2]	18 AA xb ee ff	OfIPRPf

ORY

Logic OR Y with Memory (CPU12X)

ORY

Operation

$$(Y) \mid (M : M + 1) \Rightarrow Y$$

Description

Performs bitwise logical inclusive OR between the content of index register Y and the content of memory location M : M + 1 and places the result in Y. Each bit of Y after the operation is the logical inclusive OR of the corresponding bits of M : M + 1 and of Y before the operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
ORY #opr16i	IMM	18 CA jj kk	OPO
ORY opr8a	DIR	18 DA dd	ORPf
ORY opr16a	EXT	18 FA hh ll	ORPO
ORY oprx0_xysp	IDX	18 EA xb	ORPf
ORY oprx9_xysp	IDX1	18 EA xb ff	ORPO
ORY oprx16_xysp	IDX2	18 EA xb ee ff	OfrPP
ORY [D,xysp]	[D,IDX]	18 EA xb	OfIfRPf
ORY [opr16,xysp]	[IDX2]	18 EA xb ee ff	OfIPRPf

PSHA

Push A onto Stack
(CPU12, CPU12X)

PSHA

Operation

$(SP) - \$0001 \Rightarrow SP$

$(A) \Rightarrow M_{(SP)}$

Description

Stacks the content of accumulator A. The stack pointer is decremented by one. The content of A is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHA	INH	36	all Os

PSHB

Push B onto Stack
(CPU12, CPU12X)

PSHB

Operation

$(SP) - \$0001 \Rightarrow SP$

$(B) \Rightarrow M_{(SP)}$

Description

Stacks the content of accumulator B. The stack pointer is decremented by one. The content of B is then stored at the address the SP points to.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHB	INH	37	all Os

PSHC

Push CCR onto Stack
(CPU12, CPU12X)

PSHC

Operation

$(SP) - \$0001 \Rightarrow SP$

$(CCR) \Rightarrow M_{(SP)}$

Description

Stacks the content of the condition codes register. The stack pointer is decremented by one. The content of the CCR is then stored at the address to which the SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHC	INH	39	all Os

PSHCW

Push CCRW onto Stack
(CPU12X)

PSHCW

Operation

$(SP) - 2 \Rightarrow SP; (CCR_H:CCR_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$

Description

Stacks the content of the condition codes register. The stack pointer is decremented by two. The content of the CCR is then stored at the address to which the SP points.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
–	0	0	0	0	–	–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHCW	INH	18 39	CPU12X OOS

PSHD

Push Double Accumulator onto Stack
(CPU12, CPU12X)

PSHD

Operation

$(SP) - \$0002 \Rightarrow SP$

$(A : B) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description

Stacks the content of double accumulator D. The stack pointer is decremented by two, then the contents of accumulators A and B are stored at the location to which the SP points.

After PSHD executes, the SP points to the stacked value of accumulator A. This stacking order is the opposite of the order in which A and B are stacked when an interrupt is recognized. The interrupt stacking order is backward-compatible with the M6800, which had no 16-bit accumulator.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHD	INH	3B	all OS

PSHX

Push Index Register X onto Stack
(CPU12, CPU12X)

PSHX

Operation

$(SP) - \$0002 \Rightarrow SP$

$(X_H : X_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description

Stacks the content of index register X. The stack pointer is decremented by two. The content of X is then stored at the address to which the SP points. After PSHX executes, the SP points to the stacked value of the high-order half of X.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHX	INH	34	all OS

PSHY

Push Index Register Y onto Stack
(CPU12, CPU12X)

PSHY

Operation

$(SP) - \$0002 \Rightarrow SP$

$(Y_H : Y_L) \Rightarrow M_{(SP)} : M_{(SP+1)}$

Description

Stacks the content of index register Y. The stack pointer is decremented by two. The content of Y is then stored at the address to which the SP points. After PSHY executes, the SP points to the stacked value of the high-order half of Y.

Push instructions are commonly used to save the contents of one or more CPU registers at the start of a subroutine. Complementary pull instructions can be used to restore the saved CPU registers just before returning from the subroutine.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PSHY	INH	35	<div>allOS</div>

PULA

Pull A from Stack
(CPU12, CPU12X)

PULA

Operation

$(M_{(SP)}) \Rightarrow A$
 $(SP) + \$0001 \Rightarrow SP$

Description

Accumulator A is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULA	INH	32	<div>all</div> <div>uf0</div>

PULB

Pull B from Stack
(CPU12, CPU12X)

PULB

Operation

$(M_{(SP)}) \Rightarrow B$
 $(SP) + \$0001 \Rightarrow SP$

Description

Accumulator B is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine, to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C
—	—	—	—	—	—	—	—

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULB	INH	33	<div>all</div> <div>uf0</div>

PULC

Pull Condition Code Register from Stack (CPU12, CPU12X)

PULC

Operation

$(M_{(SP)}) \Rightarrow CCR$
 $(SP) + \$0001 \Rightarrow SP$

Description

The condition code register is loaded from the address indicated by the stack pointer. The SP is then incremented by one.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C	
Δ	\Downarrow	Δ	Δ	Δ	Δ	Δ	Δ	supervisor state
—	—	Δ	—	Δ	Δ	Δ	Δ	user state

Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an \overline{XIRQ} interrupt.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULC	INH	38	<div>all</div> <div>ufo</div>

PULCW

Pull Condition Code Register
from Stack
(CPU12X)

PULCW

Operation

$$(M_{(SP)}:M_{(SP+1)}) \Rightarrow CCR_H:CCR_L; (SP) + 2 \Rightarrow SP$$

Description

The condition code register is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C	
↑	0	0	0	0	Δ	Δ	↓	Δ	Δ	Δ	Δ	Δ	Δ	supervisor state
–	0	0	0	0	–	–	–	Δ	–	Δ	Δ	Δ	Δ	user state

Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an \overline{XIRQ} interrupt.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULCW	INH	18 38	CPU12X OUfO

PULD

Pull Double Accumulator from Stack
(CPU12, CPU12X)

PULD

Operation

$$(M_{(SP)} : M_{(SP+1)}) \Rightarrow A : B$$
$$(SP) + \$0002 \Rightarrow SP$$

Description

Double accumulator D is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

The order in which A and B are pulled from the stack is the opposite of the order in which A and B are pulled when an RTI instruction is executed. The interrupt stacking order for A and B is backward-compatible with the M6800, which had no 16-bit accumulator.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULD	INH	3A	<div>all</div> <div>UFO</div>

Operation

$$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L$$

$$(SP) + \$0002 \Rightarrow SP$$

Description

Index register X is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULX	INH	30	all UfO

PULY

Pull Index Register Y from Stack
(CPU12, CPU12X)

PULY

Operation

$$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L$$
$$(SP) + \$0002 \Rightarrow SP$$

Description

Index register Y is loaded from the address indicated by the stack pointer. The SP is then incremented by two.

Pull instructions are commonly used at the end of a subroutine to restore the contents of CPU registers that were pushed onto the stack before subroutine execution.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
PULY	INH	31	all UfO

Operation

MIN-MAX Rule Evaluation

Description

Performs an unweighted evaluation of a list of rules, using fuzzy input values to produce fuzzy outputs. REV can be interrupted, so it does not adversely affect interrupt latency.

The REV instruction uses an 8-bit offset from a base address stored in index register Y to determine the address of each fuzzy input and fuzzy output. For REV to execute correctly, each rule in the knowledge base must consist of a table of 8-bit antecedent offsets followed by a table of 8-bit consequent offsets. The value \$FE marks boundaries between antecedents and consequents and between successive rules. The value \$FF marks the end of the rule list. REV can evaluate any number of rules with any number of inputs and outputs.

Beginning with the address pointed to by the first rule antecedent, REV evaluates each successive fuzzy input value until it encounters an \$FE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. Then, beginning with the address pointed to by the first rule consequent, the truth value is compared to each successive fuzzy output value until another \$FE separator is encountered; if the truth value is greater than the current output value, it is written to the output. Operation is similar to that of a MAXM instruction. Rules are processed until an \$FF terminator is encountered.

Before executing REV, perform these set up operations.

- X must point to the first 8-bit element in the rule list.
- Y must point to the base address for fuzzy inputs and fuzzy outputs.
- A must contain the value \$FF, and the CCR V bit must = 0.
(LDAA #\$FF places the correct value in A and clears V.)
- Clear fuzzy outputs to 0s.

Index register X points to the element in the rule list that is being evaluated. X is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, X points to the next address after the \$FF separator at the end of the rule list.

Index register Y points to the base address for the fuzzy inputs and fuzzy outputs. The value in Y does not change during execution.

Accumulator A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A, and writes the smaller of the two to A. When all antecedents have been evaluated, A contains the smallest input value. This is the truth value used during consequent processing. Accumulator A must be initialized to \$FF for the MIN function to evaluate the inputs of the first rule correctly. For subsequent rules, the value \$FF is written to A when an \$FE marker is encountered. At the end of execution, accumulator A holds the truth value for the last rule.

The V status bit signals whether antecedents (0) or consequents (1) are being processed. V must be initialized to 0 for processing to begin with the antecedents of the first rule. Once execution begins, the value of V is automatically changed as \$FE separators are encountered. At the end of execution, V should equal 1, because the last element before the \$FF end marker should be a rule consequent. If V is equal to 0 at the end of execution, the rule list is incorrect.

Fuzzy outputs must be cleared to \$00 before processing begins in order for the MAX algorithm used during consequent processing to work correctly. Residual output values would cause incorrect comparison.

Refer to [Chapter 9, “Fuzzy Logic Support”](#) for details.

CCR Details

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	?

V: 1; Normally set, unless rule structure is erroneous

H, N, Z, and C may be altered by this instruction

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail ¹
			CPU12V0, CPU12XV0
REV (replace comma if interrupted)	Special	18 3A	Orf(t,tx)0 ff + Orf(t,

¹ The 3-cycle loop in parentheses is executed once for each element in the rule list. When an interrupt occurs, there is a 2-cycle exit sequence, a 4-cycle re-entry sequence, then execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

Operation

MIN-MAX Rule Evaluation with Optional Rule Weighting

Description

REVW performs either weighted or unweighted evaluation of a list of rules, using fuzzy inputs to produce fuzzy outputs. REVW can be interrupted, so it does not adversely affect interrupt latency. For REVW to execute correctly, each rule in the knowledge base must consist of a table of 16-bit antecedent pointers followed by a table of 16-bit consequent pointers. The value \$FFFE marks boundaries between antecedents and consequents, and between successive rules. The value \$FFFF marks the end of the rule list. REVW can evaluate any number of rules with any number of inputs and outputs.

Setting the C status bit enables weighted evaluation. To use weighted evaluation, a table of 8-bit weighting factors, one per rule, must be stored in memory. Index register Y points to the weighting factors.

Beginning with the address pointed to by the first rule antecedent, REVW evaluates each successive fuzzy input value until it encounters an \$FFFE separator. Operation is similar to that of a MINA instruction. The smallest input value is the truth value of the rule. Next, if weighted evaluation is enabled, a computation is performed, and the truth value is modified. Then, beginning with the address pointed to by the first rule consequent, the truth value is compared to each successive fuzzy output value until another \$FFFE separator is encountered; if the truth value is greater than the current output value, it is written to the output. Operation is similar to that of a MAXM instruction. Rules are processed until an \$FFFF terminator is encountered.

Perform these set up operations before execution:

- X must point to the first 16-bit element in the rule list.
- A must contain the value \$FF, and the CCR V bit must = 0 (LDAA #\$FF places the correct value in A and clears V).
- Clear fuzzy outputs to 0s.
- Set or clear the CCR C bit. When weighted evaluation is enabled, Y must point to the first item in a table of 8-bit weighting factors.

Index register X points to the element in the rule list that is being evaluated. X is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, X points to the address after the \$FFFF separator at the end of the rule list.

Index register Y points to the weighting factor being used. Y is automatically updated so that execution can resume correctly if the instruction is interrupted. When execution is complete, Y points to the last weighting factor used. When weighting is not used ($C = 0$), Y is not changed.

Accumulator A holds intermediate results. During antecedent processing, a MIN function compares each fuzzy input to the value stored in A and writes the smaller of the two to A. When all antecedents have been evaluated, A contains the smallest input value. For unweighted evaluation, this is the truth value used during consequent processing. For weighted evaluation, the value in A is multiplied by the quantity (Rule Weight + 1) and the upper eight bits of the result replace the content of A. Accumulator A must be initialized to \$FF for the MIN function to evaluate the inputs of the first rule correctly. For subsequent rules, the value \$FF is automatically written to A when an \$FFFE marker is encountered. At the end of execution, accumulator A holds the truth value for the last rule.

The V status bit signals whether antecedents (0) or consequents (1) are being processed. V must be initialized to 0 for processing to begin with the antecedents of the first rule. Once execution begins, the value of V is automatically changed as \$FFFE separators are encountered. At the end of execution, V should equal 1, because the last element before the \$FF end marker should be a rule consequent. If V is equal to 0 at the end of execution, the rule list is incorrect. Fuzzy outputs must be cleared to \$00 before processing begins in order for the MAX algorithm used during consequent processing to work correctly. Residual output values would cause incorrect comparison.

Refer to Chapter 9, “Fuzzy Logic Support” for details.

CCR Details

S	X	H	I	N	Z	V	C
–	–	?	–	?	?	Δ	!

V: 1; Normally set, unless rule structure is erroneous

C: Selects weighted (1) or unweighted (0) rule evaluation

H, N, Z, and C may be altered by this instruction

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail ¹ CPU12V0, CPU12XV0
REVW (add 2 at end of ins if wts) (replace comma if interrupted)	Special	18 3B	$ORf(t, Tx)O(r, RfRf) ffff + ORf(t,$

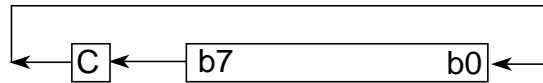
¹ The 3-cycle loop in parentheses expands to five cycles for separators when weighting is enabled. The loop is executed once for each element in the rule list. When an interrupt occurs, there is a 2-cycle exit sequence, a 4-cycle re-entry sequence, then execution resumes with a prefetch of the last antecedent or consequent being processed at the time of the interrupt.

ROL

Rotate Left Memory (CPU12, CPU12X)

ROL

Operation



Description

Shifts all bits of memory location M one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of M. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, ROL HIGH could be used where LOW, MID and HIGH refer to the low-order, middle and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M7

Set if the MSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

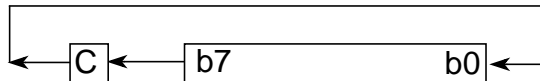
Source Form	Address Mode	Object Code	Access Detail
ROL <i>opr16a</i>	EXT	75 hh ll	rPwO
ROL <i>opr0_xysp</i>	IDX	65 xb	rPw
ROL <i>opr9,xysp</i>	IDX1	65 xb ff	rPwO
ROL <i>opr16,xysp</i>	IDX2	65 xb ee ff	frPwP
ROL [D, <i>xysp</i>]	[D,IDX]	65 xb	fIfrrPw
ROL [<i>opr16,xysp</i>]	[IDX2]	65 xb ee ff	fIPrrPw

ROLA

Rotate Left A
(CPU12, CPU12X)

ROLA

Operation



Description

Shifts all bits of accumulator A one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of A. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, and ROL HIGH could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A7

Set if the MSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

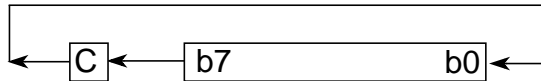
Source Form	Address Mode	Object Code	Access Detail
ROLA	INH	45	<div>all</div> <div>0</div>

ROLB

Rotate Left B
(CPU12, CPU12X)

ROLB

Operation



Description

Shifts all bits of accumulator B one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of B. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the left, the sequence ASL LOW, ROL MID, and ROL HIGH could be used where LOW, MID, and HIGH refer to the low-order, middle and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B7

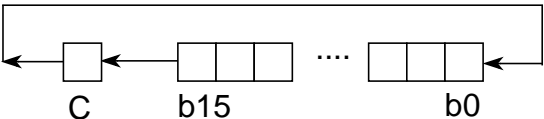
Set if the MSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
ROLB	INH	55	<div>all</div> <div>0</div>

ROLW Rotate Memory Left through Carry (16 Bit) (CPU12X) ROLW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of $M : M + 1$. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M15

Set if the MSB of $M : M + 1$ was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

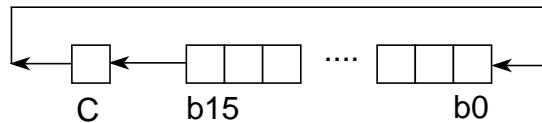
Source Form	Address Mode	Object Code	Access Detail
			CPU12X
ROLW <i>opr16a</i>	EXT	18 75 hh ll	ORPWO
ROLW <i>opr0_xysp</i>	IDX	18 65 xb	ORPW
ROLW <i>opr9_xysp</i>	IDX1	18 65 xb ff	ORPWO
ROLW <i>opr16_xysp</i>	IDX2	18 65 xb ee ff	OFRPWP
ROLW [D, <i>xysp</i>]	[D,IDX]	18 65 xb	OFRPWP
ROLW [<i>opr16_xysp</i>]	[IDX2]	18 65 xb ee ff	FOIPRPW

ROLX

Rotate X Left through Carry
(CPU12X)

ROLX

Operation



Description

Shifts all bits of index register X one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of X. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X15

Set if the MSB of X was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

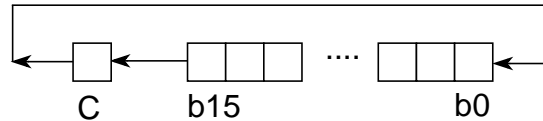
Source Form	Address Mode	Object Code	Access Detail
ROLX	INH	18 45	CPU12X 00

ROLY

Rotate Y Left through Carry
(CPU12X)

ROLY

Operation



Description

Shifts all bits of index register Y one place to the left. Bit 0 is loaded from the C status bit. The C bit is loaded from the most significant bit of Y. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y15

Set if the MSB of Y was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

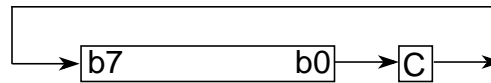
Source Form	Address Mode	Object Code	Access Detail
ROLY	INH	18 55	CPU12X 00

ROR

Rotate Right Memory
(CPU12, CPU12X)

ROR

Operation



Description

Shifts all bits of memory location M one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of M. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0
Set if the LSB of M was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

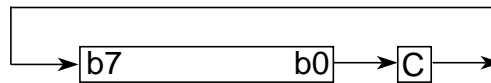
Source Form	Address Mode	Object Code	Access Detail
ROR <i>opr16a</i>	EXT	76 hh ll	rPwO
ROR <i>opr0_xysp</i>	IDX	66 xb	rPw
ROR <i>opr9_xysp</i>	IDX1	66 xb ff	rPwO
ROR <i>opr16_xysp</i>	IDX2	66 xb ee ff	frPwP
ROR [D, <i>xysp</i>]	[D,IDX]	66 xb	fIfPw
ROR [<i>opr16_xysp</i>]	[IDX2]	66 xb ee ff	fIPwP

RORA

Rotate Right A
(CPU12, CPU12X)

RORA

Operation



Description

Shifts all bits of accumulator A one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of A. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: A0

Set if the LSB of A was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

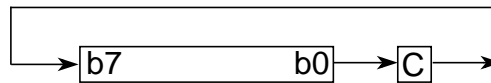
Source Form	Address Mode	Object Code	Access Detail
RORA	INH	46	<div>all</div> <div>0</div>

RORB

Rotate Right B
(CPU12, CPU12X)

RORB

Operation



Description

Shifts all bits of accumulator B one place to the right. Bit 7 is loaded from the C status bit. The C bit is loaded from the least significant bit of B. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple bytes. For example, to shift a 24-bit value one bit to the right, the sequence LSR HIGH, ROR MID, and ROR LOW could be used where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: B0

Set if the LSB of B was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

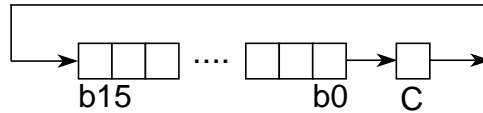
Source Form	Address Mode	Object Code	Access Detail
RORB	INH	56	<div>all</div> <div>0</div>

RORW

Rotate Memory Right through Carry
(16 Bit)
(CPU12X)

RORW

Operation



Description

Shifts all bits of memory location $M : M + 1$ one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of $M : M + 1$. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \bar{C}] + [\bar{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: M0

Set if the LSB of $M : M + 1$ was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

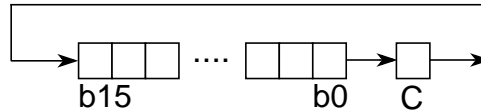
Source Form	Address Mode	Object Code	Access Detail	CPU12X
RORW <i>opr16a</i>	EXT	18 76 hh 11		ORPWO
RORW <i>opr0_xysp</i>	IDX	18 66 xb		ORPW
RORW <i>opr9_xysp</i>	IDX1	18 66 xb ff		ORPWO
RORW <i>opr16_xysp</i>	IDX2	18 66 xb ee ff		OFRPWP
RORW [D,xysp]	[D,IDX]	18 66 xb		OFIfrPW
RORW [opr16,xysp]	[IDX2]	18 66 xb ee ff		OFIrPW

RORX

Rotate X Right through Carry
(CPU12X)

RORX

Operation



Description

Shifts all bits of index register X one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of X. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)

Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: X0

Set if the LSB of X was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

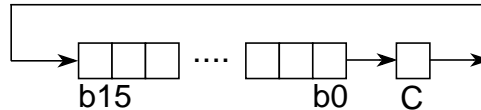
Source Form	Address Mode	Object Code	Access Detail
RORX	INH	18 46	CPU12X 00

RORY

Rotate Y Right through Carry (CPU12X)

RORY

Operation



Description

Shifts all bits of index register Y one place to the right. Bit 15 is loaded from the C status bit. The C bit is loaded from the least significant bit of Y. Rotate operations include the carry bit to allow extension of shift and rotate operations to multiple words.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $N \oplus C = [N \bullet \overline{C}] + [\overline{N} \bullet C]$ (for N and C after the shift)
Set if (N is set and C is cleared) or (N is cleared and C is set); cleared otherwise (for values of N and C after the shift)

C: Y0
Set if the LSB of Y was set before the shift; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
RORY	INH	18 56	CPU12X 00

RTC

Return from Call
(CPU12, CPU12X)

RTC

Operation

$(M_{(SP)}) \Rightarrow \text{PPAGE}$
 $(SP) + \$0001 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L$
 $(SP) + \$0002 \Rightarrow SP$

Description

Terminates subroutines in expanded memory invoked by the CALL instruction. Returns execution flow from the subroutine to the calling program. The program overlay page (PPAGE) register and the return address are restored from the stack; program execution continues at the restored address. For code compatibility purposes, CALL and RTC also execute correctly in devices that do not have expanded memory capability.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
RTC	INH	0A	<div>all</div> <div>uUnfPPP</div>

RTI

Return from Interrupt (CPU12, CPU12X)

RTI

Operation

$(M_{(SP)} : M_{(SP+1)}) \Rightarrow CCR_H : CCR_L; (SP) + \$0002 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A; (SP) + \$0002 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L; (SP) + \$0004 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) - \$0002 \Rightarrow SP$
 $(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L; (SP) + \$0004 \Rightarrow SP$

Description

Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

If another interrupt is pending when RTI has finished restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched. This operation is functionally identical to the same operation in the M68HC11, where registers actually are re-stacked, but is faster.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C	
↑	0	0	0	0	Δ	Δ	↓	Δ	Δ	Δ	Δ	Δ	Δ	supervisor state
–	0	0	0	0	–	–	–	Δ	–	Δ	Δ	Δ	Δ	user state

Condition codes take on the value pulled from the stack, except that the X mask bit cannot change from 0 to 1. Software can leave the X bit set, leave it cleared, or change it from 1 to 0, but it can be set only by a reset or by recognition of an \overline{XIRQ} interrupt.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
RTI (with interrupt pending)	INH	0B	UUUUUPPP UUUUUfVfPPP	UUUUuPPP UUUUuFVfPPP

RTS

Return from Subroutine (CPU12, CPU12X)

RTS

Operation

$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L; (SP) + \$0002 \Rightarrow SP$

Description

Restores context at the end of a subroutine. Loads the program counter with a 16-bit value pulled from the stack and increments the stack pointer by two. Program execution continues at the address restored from the stack.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
RTS	INH	3D	all UfPPP

SBA

Subtract Accumulators (CPU12, CPU12X)

SBA

Operation

$$(A) - (B) \Rightarrow A$$

Description

Subtracts the content of accumulator B from the content of accumulator A and places the result in A. The content of B is not affected. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{B7} \bullet \overline{R7} + \overline{A7} \bullet B7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet B7 + B7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the absolute value of B is larger than the absolute value of A; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
SBA	INH	18 16	all 00

SBCA

Subtract with Carry from A
(CPU12, CPU12X)

SBCA

Operation

$$(A) - (M) - C \Rightarrow A$$

Description

Subtracts the content of memory location M and the value of the C status bit from the content of accumulator A. The result is placed in A. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

$$V: A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$$

Set if a two's complement overflow resulted from the operation; cleared otherwise

$$C: \overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
SBCA #opr8i	IMM	82 ii	P
SBCA opr8a	DIR	92 dd	rPf
SBCA opr16a	EXT	B2 hh ll	rPO
SBCA oprx0_xysp	IDX	A2 xb	rPf
SBCA oprx9,xysp	IDX1	A2 xb ff	rPO
SBCA oprx16,xysp	IDX2	A2 xb ee ff	frPP
SBCA [D,xysp]	[D,IDX]	A2 xb	fIf rPf
SBCA [oprx16,xysp]	[IDX2]	A2 xb ee ff	fIPrPf

SBCB

Subtract with Carry from B
(CPU12, CPU12X)

SBCB

Operation

$$(B) - (M) - C \Rightarrow B$$

Description

Subtracts the content of memory location M and the value of the C status bit from the content of accumulator B. The result is placed in B. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
SBCB #opr8i	IMM	C2 ii	P
SBCB opr8a	DIR	D2 dd	rPf
SBCB opr16a	EXT	F2 hh ll	rPO
SBCB oprx0_xysp	IDX	E2 xb	rPf
SBCB oprx9,xysp	IDX1	E2 xb ff	rPO
SBCB oprx16,xysp	IDX2	E2 xb ee ff	frPP
SBCB [D,xysp]	[D,IDX]	E2 xb	fIf rPf
SBCB [oprx16,xysp]	[IDX2]	E2 xb ee ff	fIPrPf

SBED

Subtract with Borrow from D (CPU12X)

SBED

Operation

$$(D) - (M : M + 1) - C \Rightarrow D$$

Description

Subtracts the content of memory location $M : M + 1$ and the value of the C status bit from the content of accumulator D. The result is placed in D. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: The zero bit is set if the result is \$0000 AND the zero bit was set before the instruction

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
SBED #opr16i	IMM	18 83 jj kk	OPO
SBED opr8a	DIR	18 93 dd	ORP _f
SBED opr16a	EXT	18 B3 hh ll	ORPO
SBED oprx0_xysp	IDX	18 A3 xb	ORP _f
SBED oprx9_xysp	IDX1	18 A3 xb ff	ORPO
SBED oprx16_xysp	IDX2	18 A3 xb ee ff	OFRPP
SBED [D,xysp]	[D,IDX]	18 A3 xb	OfIFRP _f
SBED [opr16,xysp]	[IDX2]	18 A3 xb ee ff	OfIPRP _f

SBEX

Subtract with Borrow from X
(CPU12X)

SBEX

Operation

$$(X) - (M : M + 1) - C \Rightarrow X$$

Description

Subtracts the content of memory location M : M + 1 and the value of the C status bit from the content of index register X. The result is placed in X. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: The zero bit is set if the result is \$0000 AND the zero bit was set before the instruction

V: $X_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{X_{15}} \bullet M_{15} \bullet R_{15}$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} \bullet \overline{X_{15}}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
SBEX #opr16i	IMM	18 82 jj kk	OPO
SBEX opr8a	DIR	18 92 dd	ORPf
SBEX opr16a	EXT	18 B2 hh ll	ORPO
SBEX oprx0_xysp	IDX	18 A2 xb	ORPf
SBEX oprx9,xysp	IDX1	18 A2 xb ff	ORPO
SBEX oprx16,xysp	IDX2	18 A2 xb ee ff	OfRPP
SBEX [D,xysp]	[D,IDX]	18 A2 xb	OfIfRPf
SBEX [opr16,xysp]	[IDX2]	18 A2 xb ee ff	OfIPRPf

SBEY

Subtract with Borrow from Y (CPU12X)

SBEY

Operation

$$(Y) - (M : M + 1) - C \Rightarrow Y$$

Description

Subtracts the content of memory location M : M + 1 and the value of the C status bit from the content of index register Y. The result is placed in Y. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: The zero bit is set if the result is \$0000 AND the zero bit was set before the instruction

V: $Y15 \bullet \overline{M15} \bullet \overline{R15} + \overline{Y15} \bullet M15 \bullet R15$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{Y15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{Y15}$

Set if the absolute value of the content of memory plus previous carry is larger than the absolute value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
SBEY #opr16i	IMM	18 C2 jj kk	OPO
SBEY opr8a	DIR	18 D2 dd	ORP _f
SBEY opr16a	EXT	18 F2 hh ll	ORPO
SBEY oprx0_xysp	IDX	18 E2 xb	ORP _f
SBEY oprx9_xysp	IDX1	18 E2 xb ff	ORPO
SBEY oprx16_xysp	IDX2	18 E2 xb ee ff	O _f RP _{PP}
SBEY [D,xysp]	[D,IDX]	18 E2 xb	O _f I _f RP _f
SBEY [opr16,xysp]	[IDX2]	18 E2 xb ee ff	O _f I _f RP _f

SEC

Set Carry (CPU12, CPU12X)

SEC

Operation

1 \Rightarrow C bit

Description

Sets the C status bit. This instruction is assembled as ORCC #\$01. The ORCC instruction can be used to set any combination of bits in the CCR in one operation.

SEC can be used to set up the C bit prior to a shift or rotate instruction involving the C bit.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	1

C: 1; set

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
SEC <i>translates to...</i> ORCC #\$01	IMM	14 01	<div>all</div> <div>P</div>

SEI

Set Interrupt Mask (CPU12, CPU12X)

SEI

Operation

1 \Rightarrow I bit

Description

Sets the I mask bit. This instruction is assembled as ORCC #\$10. The ORCC instruction can be used to set any combination of bits in the CCR in one operation. When the I bit is set, all maskable interrupts are inhibited, and the CPU will recognize only non-maskable interrupt sources or an SWI.

CCR Details

S	X	H	I	N	Z	V	C	
–	–	–	1	–	–	–	–	supervisor state
–	–	–	–	–	–	–	–	user state

I: 1; set

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
SEI <i>translates to...</i> ORCC #\$10	IMM	14 10	<div>all</div> <div>P</div>

SEV

Set Two's Complement Overflow Bit (CPU12, CPU12X)

SEV

Operation

1 \Rightarrow V bit

Description

Sets the V status bit. This instruction is assembled as ORCC #\$02. The ORCC instruction can be used to set any combination of bits in the CCR in one operation.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	1	–

V: 1; set

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
SEV <i>translates to...</i> ORCC #\$02	IMM	14 02	all P

SEX

Sign Extend into 16-Bit Register (CPU12, CPU12X)

SEX

Operation

If r1 bit 7 = 0, then \$00 : (r1) \Rightarrow r3; If r1 bit 7 = 1, then \$FF : (r1) \Rightarrow r3

If r2 bit 15 = 0, then \$0000 \Rightarrow r3; If r2 bit 15 = 1, then \$FFFF \Rightarrow r3

Description

For the case r1,r3 this instruction is an alternate mnemonic for the TFR r1,r3 instruction, where r1 is an 8-bit register and r3 is a 16-bit register. The case r2,r3, where both r2 and r3 are 16-bit registers, is new on CPU12XV0. For both cases the result in r3 is the 16-bit sign extended representation of the original two's complement number in r1 or r2. The content of r1 or r2 is unchanged in all cases except that of SEX A,D (D is A : B).

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail	
			CPU12X	CPU12
SEX <i>abc,dxys</i>	INH	B7 eb	P	P
SEX <i>d,xy</i>	INH	B7 eb	P	NA

¹ Legal coding for eb is summarized in the following table. Columns represent the high-order source digit. Rows represent the low-order destination digit. Values are in hexadecimal.

	MS \Rightarrow	0	1	2	4
\Downarrow LS		A	B	CCR	D
3	TMP2	sex:A \Rightarrow TMP2 SEX A,TMP2	sex:B \Rightarrow TMP2 SEX B,TMP2	sex:CCR _L \Rightarrow TMP2 SEX CCR,TMP2 SEX CCRL,TMP2	NA
4	D	sex:A \Rightarrow D SEX A,D	sex:B \Rightarrow D SEX B,D	sex:CCR _L \Rightarrow D SEX CCR _L ,D SEX CCRL,D	NA
5	X	sex:A \Rightarrow X SEX A,X	sex:B \Rightarrow X SEX B,X	sex:CCR _L \Rightarrow X SEX CCR,X SEX CCRL,X	NA
6	Y	sex:A \Rightarrow Y SEX A,Y	sex:B \Rightarrow Y SEX B,Y	sex:CCR _L \Rightarrow Y SEX CCR,Y SEX CCRL,Y	NA
7	SP	sex:A \Rightarrow SP SEX A,SP	sex:B \Rightarrow SP SEX B,SP	sex:CCR _L \Rightarrow SP SEX CCR,SP SEX CCRL,SP	NA
C	D	sex:A \Rightarrow D SEX A,D	sex:B \Rightarrow D SEX B,D	NA	NA
D	X	NA	NA	NA	sex:D \Rightarrow X SEX D,X
E	Y	NA	NA	NA	sex:D \Rightarrow Y SEX D,Y

Note: Encodings in the shaded area (LS = C–E) are only available on the CPU12X.

STAA

Store Accumulator A
(CPU12, CPU12X)

STAA

Operation

(A) \Rightarrow M

Description

Stores the content of accumulator A in memory location M. The content of A is unchanged.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STAA <i>opr8a</i>	DIR	5A dd	Pw
STAA <i>opr16a</i>	EXT	7A hh ll	PwO
STAA <i>opr0_xysp</i>	IDX	6A xb	Pw
STAA <i>opr9_xysp</i>	IDX1	6A xb ff	PwO
STAA <i>opr16_xysp</i>	IDX2	6A xb ee ff	PwP
STAA [D, <i>xysp</i>]	[D,IDX]	6A xb	PIfw
STAA [<i>opr16_xysp</i>]	[IDX2]	6A xb ee ff	PIPw

STAB

Store Accumulator B
(CPU12, CPU12X)

STAB

Operation

(B) \Rightarrow M

Description

Stores the content of accumulator B in memory location M. The content of B is unchanged.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STAB <i>opr8a</i>	DIR	5B dd	Pw
STAB <i>opr16a</i>	EXT	7B hh ll	PwO
STAB <i>opr0_xysp</i>	IDX	6B xb	Pw
STAB <i>opr9_xysp</i>	IDX1	6B xb ff	PwO
STAB <i>opr16_xysp</i>	IDX2	6B xb ee ff	PwP
STAB [D, <i>xysp</i>]	[D,IDX]	6B xb	PIfw
STAB [<i>opr16_xysp</i>]	[IDX2]	6B xb ee ff	PIPw

STD

Store Double Accumulator (CPU12, CPU12X)

STD

Operation

$(A : B) \Rightarrow M : M + 1$

Description

Stores the content of double accumulator D in memory location $M : M + 1$. The content of D is unchanged.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STD <i>opr8a</i>	DIR	5C dd	PW
STD <i>opr16a</i>	EXT	7C hh ll	PWO
STD <i>opr0_xysp</i>	IDX	6C xb	PW
STD <i>opr9,xysp</i>	IDX1	6C xb ff	PWO
STD <i>opr16,xysp</i>	IDX2	6C xb ee ff	PWP
STD [D, <i>xysp</i>]	[D,IDX]	6C xb	PIfW
STD [<i>opr16,xysp</i>]	[IDX2]	6C xb ee ff	PIPW

STOP

Stop Processing (CPU12, CPU12X)

STOP

Operation

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$

In case of CPU12

$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$

In case of CPU12X

$(SP) - \$0002 \Rightarrow SP; CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

Stop All Clocks

Description

When the S control bit is set, STOP is disabled and operates like a 2-cycle NOP instruction. When the S bit is cleared, STOP stacks CPU12 context, stops all system clocks, and puts the device in standby mode.

Standby operation minimizes system power consumption. The contents of registers and the states of I/O pins remain unchanged.

Asserting the \overline{RESET} , \overline{XIRQ} , or \overline{IRQ} signals ends standby mode. Stacking on entry to STOP allows the CPU12 to recover quickly when an interrupt is used, provided a stable clock is applied to the device. If the system uses a clock reference crystal that also stops during low-power mode, crystal startup delay lengthens recovery time.

If \overline{XIRQ} is asserted while the X mask bit = 0 (\overline{XIRQ} interrupts enabled), execution resumes with a vector fetch for the \overline{XIRQ} interrupt. While the X mask bit = 1 (\overline{XIRQ} interrupts disabled), a 2-cycle recovery sequence is used to adjust the instruction queue and the stack pointer, and execution continues with the next instruction after STOP.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail		
			CPU12	CPU12XV0, CPU12XV1	CPU12XV2
STOP (entering STOP)	INH	18 3E	00SSSSsf	00SSSSsf	
(exiting STOP)			fVfPPP	fVfPPP	
(continue)			ff	ff	

Source Form	Address Mode	Object Code	Access Detail		
			CPU12	CPU12XV0, CPU12XV1	CPU12XV2
(if STOP disabled)			OO		OO
(CPU in user state)			NA	NA	OO

STS

Store Stack Pointer (CPU12, CPU12X)

STS

Operation

$(SP_H : SP_L) \Rightarrow M : M + 1$

Description

Stores the content of the stack pointer in memory. The most significant byte of the SP is stored at the specified address, and the least significant byte of the SP is stored at the next higher byte address (the specified address plus one).

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STS <i>opr8a</i>	DIR	5F dd	PW
STS <i>opr16a</i>	EXT	7F hh ll	PWO
STS <i>opr0_xysp</i>	IDX	6F xb	PW
STS <i>opr9,xysp</i>	IDX1	6F xb ff	PWO
STS <i>opr16,xysp</i>	IDX2	6F xb ee ff	PWP
STS [D, <i>xysp</i>]	[D,IDX]	6F xb	PIfW
STS [<i>opr16,xysp</i>]	[IDX2]	6F xb ee ff	PIPW

STX

Store Index Register X (CPU12, CPU12X)

STX

Operation

$$(X_H : X_L) \Rightarrow M : M + 1$$

Description

Stores the content of index register X in memory. The most significant byte of X is stored at the specified address, and the least significant byte of X is stored at the next higher byte address (the specified address plus one).

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STX <i>opr8a</i>	DIR	5E dd	PW
STX <i>opr16a</i>	EXT	7E hh ll	PWO
STX <i>opr0_xysp</i>	IDX	6E xb	PW
STX <i>opr9_xysp</i>	IDX1	6E xb ff	PWO
STX <i>opr16_xysp</i>	IDX2	6E xb ee ff	PWP
STX [D, <i>xysp</i>]	[D,IDX]	6E xb	PIfW
STX [<i>opr16_xysp</i>]	[IDX2]	6E xb ee ff	PIPW

STY

Store Index Register Y (CPU12, CPU12X)

STY

Operation

$(Y_H : Y_L) \Rightarrow M : M + 1$

Description

Stores the content of index register Y in memory. The most significant byte of Y is stored at the specified address, and the least significant byte of Y is stored at the next higher byte address (the specified address plus one).

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
STY <i>opr8a</i>	DIR	5D dd	PW
STY <i>opr16a</i>	EXT	7D hh ll	PWO
STY <i>opr0_xysp</i>	IDX	6D xb	PW
STY <i>opr9_xysp</i>	IDX1	6D xb ff	PWO
STY <i>opr16_xysp</i>	IDX2	6D xb ee ff	PWP
STY [D, <i>xysp</i>]	[D,IDX]	6D xb	PIfW
STY [<i>opr16_xysp</i>]	[IDX2]	6D xb ee ff	PIPW

SUBA

Subtract A
(CPU12, CPU12X)

SUBA

Operation

$$(A) - (M) \Rightarrow A$$

Description

Subtracts the content of memory location M from the content of accumulator A, and places the result in A. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $A7 \bullet \overline{M7} \bullet \overline{R7} + \overline{A7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{A7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{A7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
SUBA #opr8i	IMM	80 ii	P
SUBA opr8a	DIR	90 dd	rPf
SUBA opr16a	EXT	B0 hh ll	rPO
SUBA oprx0_xysp	IDX	A0 xb	rPf
SUBA oprx9_xysp	IDX1	A0 xb ff	rPO
SUBA oprx16_xysp	IDX2	A0 xb ee ff	frPP
SUBA [D,xysp]	[D,IDX]	A0 xb	fIfrPf
SUBA [opr16,xysp]	[IDX2]	A0 xb ee ff	fIPrPf

SUBB

Subtract B
(CPU12, CPU12X)

SUBB

Operation

$$(B) - (M) \Rightarrow B$$

Description

Subtracts the content of memory location M from the content of accumulator B and places the result in B. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: $B7 \bullet \overline{M7} \bullet \overline{R7} + \overline{B7} \bullet M7 \bullet R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{B7} \bullet M7 + M7 \bullet R7 + R7 \bullet \overline{B7}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
SUBB #opr8i	IMM	C0 ii	P
SUBB opr8a	DIR	D0 dd	rPf
SUBB opr16a	EXT	F0 hh ll	rPO
SUBB oprx0_xysp	IDX	E0 xb	rPf
SUBB oprx9_xysp	IDX1	E0 xb ff	rPO
SUBB oprx16_xysp	IDX2	E0 xb ee ff	frPP
SUBB [D,xysp]	[D,IDX]	E0 xb	fIfrPf
SUBB [opr16,xysp]	[IDX2]	E0 xb ee ff	fIPrPf

SUBD

Subtract Double Accumulator
(CPU12, CPU12X)

SUBD

Operation

$$(A : B) - (M : M + 1) \Rightarrow A : B$$

Description

Subtracts the content of memory location $M : M + 1$ from the content of double accumulator D and places the result in D . For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $D15 \bullet \overline{M15} \bullet \overline{R15} + \overline{D15} \bullet M15 \bullet R15$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{D15} \bullet M15 + M15 \bullet R15 + R15 \bullet \overline{D15}$

Set if the value of the content of memory is larger than the value of the accumulator; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
SUBD #opr16i	IMM	83 jj kk	PO
SUBD opr8a	DIR	93 dd	RPf
SUBD opr16a	EXT	B3 hh ll	RPO
SUBD oprx0_xysp	IDX	A3 xb	RPf
SUBD oprx9_xysp	IDX1	A3 xb ff	RPO
SUBD oprx16_xysp	IDX2	A3 xb ee ff	fRPP
SUBD [D,xysp]	[D,IDX]	A3 xb	fIfRPf
SUBD [opr16,xysp]	[IDX2]	A3 xb ee ff	fIPRPf

SUBX

Subtract Memory from X
(CPU12X)

SUBX

Operation

$$(X) - (M : M + 1) \Rightarrow X$$

Description

Subtracts the content of memory location M : M + 1 from the content of index register X and places the result in X. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $X_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{X_{15}} \bullet M_{15} \bullet R_{15}$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{X_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} \bullet \overline{X_{15}}$

Set if the value of the content of memory is larger than the value of the index register; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
SUBX #opr16i	IMM	18 80 jj kk	OPO
SUBX opr8a	DIR	18 90 dd	ORPf
SUBX opr16a	EXT	18 B0 hh ll	ORPO
SUBX oprx0_xysp	IDX	18 A0 xb	ORPf
SUBX oprx9_xysp	IDX1	18 A0 xb ff	ORPO
SUBX oprx16_xysp	IDX2	18 A0 xb ee ff	OFRPP
SUBX [D,xysp]	[D,IDX]	18 A0 xb	OfIFRPf
SUBX [oprx16,xysp]	[IDX2]	18 A0 xb ee ff	OfIPRPf

SUBY

Subtract Memory from Y (CPU12X)

SUBY

Operation

$$(Y) - (M : M + 1) \Rightarrow Y$$

Description

Subtracts the content of memory location $M : M + 1$ from the content of index register Y and places the result in Y. For subtraction instructions, the C status bit represents a borrow.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	Δ	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: $Y_{15} \bullet \overline{M_{15}} \bullet \overline{R_{15}} + \overline{Y_{15}} \bullet M_{15} \bullet R_{15}$

Set if a two's complement overflow resulted from the operation; cleared otherwise

C: $\overline{Y_{15}} \bullet M_{15} + M_{15} \bullet R_{15} + R_{15} \bullet \overline{Y_{15}}$

Set if the value of the content of memory is larger than the value of the index register; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
SUBY #opr16i	IMM	18 C0 jj kk	OPO
SUBY opr8a	DIR	18 D0 dd	ORPf
SUBY opr16a	EXT	18 F0 hh ll	ORPO
SUBY oprx0_xysp	IDX	18 E0 xb	ORPf
SUBY oprx9_xysp	IDX1	18 E0 xb ff	ORPO
SUBY oprx16_xysp	IDX2	18 E0 xb ee ff	OFRPP
SUBY [D,xysp]	[D,IDX]	18 E0 xb	OfIFRPf
SUBY [opr16,xysp]	[IDX2]	18 E0 xb ee ff	OfIFRPf

SWI

Software Interrupt (CPU12, CPU12X)

SWI

Operation

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$

In case of CPU12

$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$

In case of CPU12X

$(SP) - \$0002 \Rightarrow SP; CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$1 \Rightarrow I; 0 \Rightarrow U$

$(SWI\ Vector) \Rightarrow PC$

Description

Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SWI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the user state bit (U) is cleared, the PC is loaded with the SWI vector, and instruction execution resumes at that location. SWI is not affected by the I mask bit. Refer to [Chapter 7](#), “Exception Processing” for more information.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
0	–	–	–	–	–	–	–	–	1	–	–	–	–

I: 1; set

U: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
SWI	INH	3F	VSPSSPSSP	VSPSSPSsP ¹

¹ The CPU12 and CPU12X also use the SWI processing sequence for hardware interrupts and unimplemented opcode traps. A variation of the sequence (VfPPP) is used for resets.

Operation

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $(SP) - \$0002 \Rightarrow SP; CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$
 $1 \Rightarrow I; 0 \Rightarrow U$
 $(SYS \text{ Vector}) \Rightarrow PC$

Description

Causes an interrupt without an external interrupt service request. Uses the address of the next instruction after SYS as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked. The I mask bit is then set, the user state bit (U) is cleared, the PC is loaded with the SYS vector, and instruction execution resumes at that location. SYS is not affected by the I mask bit. Refer to [Chapter 7, “Exception Processing”](#) for more information.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
0	–	–	–	–	–	–	–	–	1	–	–	–	–

I: 1; set

U: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12XV1, CPU12XV2
SYS	INH	18 A7	OVSPSSPSSP ¹

¹ The CPU12X also uses the SYS processing sequence for unimplemented opcode traps.

TAB

Transfer from Accumulator A
to Accumulator B
(CPU12, CPU12X)

TAB

Operation

(A) \Rightarrow B

Description

Moves the content of accumulator A to accumulator B. The former content of B is lost; the content of A is not affected. Unlike the general transfer instruction TFR A,B which does not affect condition codes, the TAB instruction affects the N, Z, and V status bits for compatibility with M68HC11.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TAB	INH	18 0E	<div>all</div> <div>00</div>

TAP

Transfer from Accumulator A to Condition Code Register (CPU12, CPU12X)

TAP

Operation

$(A) \Rightarrow CCR_L$

Description

Transfers the logic states of bits [7:0] of accumulator A to the corresponding bit positions of the CCR. The content of A remains unchanged. The X mask bit can be cleared as a result of a TAP, but cannot be set if it was cleared prior to execution of the TAP. If the I bit is cleared, there is a 1-cycle delay before the system allows interrupt requests. This prevents interrupts from occurring between instructions in the sequences CLI, WAI and CLI, STOP.

This instruction is accomplished with the TFR A,CCR instruction. For compatibility with the M68HC11, the mnemonic TAP is translated by the assembler.

CCR Details

S	X	H	I	N	Z	V	C	
Δ	\Downarrow	Δ	Δ	Δ	Δ	Δ	Δ	supervisor state
—	—	Δ	—	Δ	Δ	Δ	Δ	user state

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{XIRQ} interrupt.

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TAP <i>translates to...</i> TFR A,CCR	INH	B7 02	<div>all</div> <div>P</div>

TBA

Transfer from Accumulator B
to Accumulator A
(CPU12, CPU12X)

TBA

Operation

(B) \Rightarrow A

Description

Moves the content of accumulator B to accumulator A. The former content of A is lost; the content of B is not affected. Unlike the general transfer instruction TFR B,A, which does not affect condition codes, the TBA instruction affects N, Z, and V for compatibility with M68HC11.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	–

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TBA	INH	18 0F	<div>all</div> 00

TBEQ

Test and Branch if Equal to Zero
(CPU12, CPU12X)

TBEQ

Operation

If (Counter) = 0, then (PC) + \$0003 + Rel \Rightarrow PC

Description

Tests the specified counter register A, B, D, X, Y, or SP. If the counter register is zero, branches to the specified relative destination. TBEQ is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBEQ and IBEQ instructions are similar to TBEQ, except that the counter is decremented or incremented rather than simply being tested. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
TBEQ <i>abdxys,rel9</i>	REL	04 1b rr	all PPP/PP0

¹ Encoding for 1b is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (TBEQ – 0) or not zero (TBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 0:1 for TBEQ.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	TBEQ A, <i>rel9</i>	04 40 rr	04 50 rr
B	001	TBEQ B, <i>rel9</i>	04 41 rr	04 51 rr
D	100	TBEQ D, <i>rel9</i>	04 44 rr	04 54 rr
X	101	TBEQ X, <i>rel9</i>	04 45 rr	04 55 rr
Y	110	TBEQ Y, <i>rel9</i>	04 46 rr	04 56 rr
SP	111	TBEQ SP, <i>rel9</i>	04 47 rr	04 57 rr

Operation

$$(M) + [(B) \times ((M+1) - (M))] \Rightarrow A$$

Description

Linearly interpolates one of 256 result values that fall between each pair of data entries in a lookup table stored in memory. Data entries in the table represent the Y values of endpoints of equally spaced line segments. Table entries and the interpolated result are 8-bit values. The result is stored in accumulator A. Before executing TBL, an index register points to the table entry corresponding to the X value (X1) that is closest to, but less than or equal to, the desired lookup point (XL, YL). This defines the left end of a line segment and the right end is defined by the next data entry in the table. Prior to execution, accumulator B holds a binary fraction (radix point to left of MSB), representing the ratio $(XL - X1) \div (X2 - X1)$.

The 8-bit unrounded result is calculated using the following expression:

$$A = Y1 + [(B) \times (Y2 - Y1)]$$

Where :

$$(B) = (XL - X1) \div (X2 - X1)$$

Y1 = 8-bit data entry pointed to by <effective address>

Y2 = 8-bit data entry pointed to by <effective address> + 1

The intermediate value $[(B) \times (Y2 - Y1)]$ produces a 16-bit result with the radix point between bits 7 and 8. Any indexed addressing mode referenced to X, Y, SP, or PC, except indirect modes or 9-bit and 16-bit offset modes, can be used to identify the first data point (X1,Y1). The second data point is the next table entry.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	Δ	Δ	-	Δ

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

C: Set if result can be rounded up; cleared otherwise

Detailed Syntax and Cycle-by-Cycle Operation

SourceForm	AddressMode	ObjectCode	Access Detail
TBL <i>oprX0_xysp</i>	IDX	18 3D xb	all ORffffP

TBNE

Test and Branch if Not Equal to Zero
(CPU12, CPU12X)

TBNE

Operation

If (Counter) $\neq 0$, then $(PC) + \$0003 + Rel \Rightarrow PC$

Description

Tests the specified counter register A, B, D, X, Y, or SP. If the counter register is not zero, branches to the specified relative destination. TBNE is encoded into three bytes of machine code including a 9-bit relative offset (–256 to +255 locations from the start of the next instruction).

DBNE and IBNE instructions are similar to TBNE, except that the counter is decremented or incremented rather than simply being tested. Bits 7 and 6 of the instruction postbyte are used to determine which operation is to be performed.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code ¹	Access Detail
TBNE <i>abdxys,rel9</i>	REL	04 1b rr	all PPP/PPO

¹ Encoding for 1b is summarized in the following table. Bit 3 is not used (don't care), bit 5 selects branch on zero (TBEQ – 0) or not zero (TBNE – 1) versions, and bit 4 is the sign bit of the 9-bit relative offset. Bits 7 and 6 should be 0:1 for TBNE.

Count Register	Bits 2:0	Source Form	Object Code (If Offset is Positive)	Object Code (If Offset is Negative)
A	000	TBNE A, <i>rel9</i>	04 60 rr	04 70 rr
B	001	TBNE B, <i>rel9</i>	04 61 rr	04 71 rr
D	100	TBNE D, <i>rel9</i>	04 64 rr	04 74 rr
X	101	TBNE X, <i>rel9</i>	04 65 rr	04 75 rr
Y	110	TBNE Y, <i>rel9</i>	04 66 rr	04 76 rr
SP	111	TBNE SP, <i>rel9</i>	04 67 rr	04 77 rr

TFR

Transfer Register Content to Another Register (Continued) (CPU12, CPU12X)

TFR

	MS⇒	0	1	2	3	4	5	6	7
↓ LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR _L ⇒ A TFR CCR,A TFR CCRL,A	TMP3 _L ⇒ A TFR TMP3,A TFR TMP3L,A	B ⇒ A TFR D,A	X _L ⇒ A TFR X, A TFR XL,A	Y _L ⇒ A TFR Y,A TFR YL,A	SP _L ⇒ A TFR SP,A TFR SPL,A
1	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR _L ⇒ B TFR CCR,B TFR CCRL,B	TMP3 _L ⇒ B TFR TMP3,B TFR TMP3L,B	B ⇒ B TFR D,B	X _L ⇒ B TFR X, B TFR XL,B	Y _L ⇒ B TFR Y,B TFR YL,B	SP _L ⇒ B TFR SP,B TFR SPL,B
2	CCR	A ⇒ CCR TFR A,CCR TFR A,CCRL	B ⇒ CCR TFR B,CCR TFR B,CCRL	CCR _L ⇒ CCR _L TFR CCR,CCR TFR CCRL,CCRL	TMP3 _L ⇒ CCR TFR TMP3,CCR TFR TMP3L,CCRL	B ⇒ CCR TFR D,CCR TFR D,CCRL	X _L ⇒ CCR TFR X,CCR TFR XL,CCRL	Y _L ⇒ CCR TFR Y,CCR TFR YL,CCRL	SP _L ⇒ CCR TFR SP,CCR TFR SPL,CCRL
3	TMP2	sex:A ⇒ TMP2 SEX A,TMP2	sex:B ⇒ TMP2 SEX B,TMP2	sex:CCR _L ⇒ TMP2 SEX CCR,TMP2 SEX CCRL,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP2 TFR D,TMP2	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
4	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR _L ⇒ D SEX CCR _L ,D SEX CCRL,D	TMP3 ⇒ D TFR TMP3,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
5	X	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR _L ⇒ X SEX CCR _L ,X SEX CCRL,X	TMP3 ⇒ X TFR TMP3,X	D ⇒ X TFR D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
6	Y	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR _L ⇒ Y SEX CCR _L ,Y SEX CCRL,Y	TMP3 ⇒ Y TFR TMP3,Y	D ⇒ Y TFR D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
7	SP	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR _L ⇒ SP SEX CCR _L ,SP SEX CCRL,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP
8	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR _H ⇒ A TFR CCRH,A	TMP3 _H ⇒ A TFR TMP3H,A	B ⇒ A TFR D,A	X _H ⇒ A TFR XH, A	Y _H ⇒ A TFR YH,A	SP _H ⇒ A TFR SPH,A
9	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR _L ⇒ B TFR CCRL,B	TMP3 _L ⇒ B TFR TMP3L,B	B ⇒ B TFR D,B	X _L ⇒ B TFR XL, B	Y _L ⇒ B TFR YL,B	SP _L ⇒ B TFR SPL,B
A	CCR	A ⇒ CCR _H TFR A,CCRH	B ⇒ CCR _L TFR B,CCRL	CCR _W ⇒ CCR _W TFR CCRW,CCRW	TMP3 ⇒ CCR _{H,L} TFR TMP3,CCRW	D ⇒ CCR _{H,L} TFR D,CCRW	X ⇒ CCR _{H,L} TFR X,CCRW	Y ⇒ CCR _{H,L} TFR Y,CCRW	SP ⇒ CCR _{H,L} TFR SP,CCRW
B	TMPx	A ⇒ TMP2 _H TFR A,TMP2H	B ⇒ TMP2 _L TFR B,TMP2L	CCR _{H,L} ⇒ TMP2 TFR CCRW,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP1 TFR D,TMP1	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
C	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	CCR _{H,L} ⇒ D TFR CCRW,D	TMP1 ⇒ D TFR TMP1,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
D	X	A ⇒ X _H TFR A,XH	B ⇒ X _L TFR B,XL	CCR _{H,L} ⇒ X TFR CCRW,X	TMP3 ⇒ X TFR TMP3,X	sex:D ⇒ X SEX D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
E	Y	A ⇒ Y _H TFR A,YH	B ⇒ Y _L TFR B,YL	CCR _{H,L} ⇒ Y TFR CCRW,Y	TMP3 ⇒ Y TFR TMP3,Y	sex:D ⇒ Y SEX D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
F	SP	A ⇒ SP _H TFR A,SPH	B ⇒ SP _L TFR B,SPL	CCR _{H,L} ⇒ SP TFR CCRW,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on CPU12X.

TPA

Transfer from Condition Code Register to Accumulator A (CPU12, CPU12X)

TPA

Operation

$(CCR_L) \Rightarrow A$

Description

Transfers the content of the condition code register to corresponding bit positions of accumulator A. The CCR remains unchanged.

This mnemonic is implemented by the TFR CCR,A instruction. For compatibility with the M68HC11, the mnemonic TPA is translated into the TFR CCR,A instruction by the assembler.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TPA <i>translates to...</i> TFR CCR,A	INH	B7 20	<div>allP</div>

TRAP

Unimplemented Opcode Trap (CPU12, CPU12X)

TRAP

Operation

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$

In case of CPU12

$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$

In case of CPU12X

$(SP) - \$0002 \Rightarrow SP; CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$1 \Rightarrow I; 0 \Rightarrow U$

(Trap Vector) $\Rightarrow PC$

Description

Traps unimplemented opcodes. There are opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU attempts to execute one of the unimplemented opcodes on page 2, an opcode trap interrupt occurs. Unimplemented opcode traps are essentially interrupts that share the \$FFF8:\$FFF9 interrupt vector.

TRAP uses the next address after the unimplemented opcode as a return address. It stacks the return address, index registers Y and X, accumulators B and A, and the CCR, automatically decrementing the SP before each item is stacked. The I mask bit is then set, the user state bit (U) is cleared, the PC is loaded with the trap vector, and instruction execution resumes at that location. This instruction is not maskable by the I bit. Refer to [Chapter 7, “Exception Processing”](#) for more information.

CCR Details

U	0	0	0	0	IPL	S	X	H	I	N	Z	V	C
0	–	–	–	–	–	–	–	–	1	–	–	–	–

I: 1; set

U: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail	
			CPU12X	CPU12
TRAP <i>trapnum</i>	INH	$\$18 \text{ } tn^1$	OVSPSSPSSP	OVSPSSPSsP

¹The value tn represents an unimplemented page 2 opcode (valid ranges depend on CPU version).

TST

Test Memory (CPU12, CPU12X)

TST

Operation

(M) – \$00

Description

Subtracts \$00 from the content of memory location M and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying M.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			all
TST <i>opr16a</i>	EXT	F7 hh ll	rPO
TST <i>opr0_xysp</i>	IDX	E7 xb	rPf
TST <i>opr9,xysp</i>	IDX1	E7 xb ff	rPO
TST <i>opr16,xysp</i>	IDX2	E7 xb ee ff	frPP
TST [D, <i>xysp</i>]	[D,IDX]	E7 xb	fIfrPf
TST [<i>opr16,xysp</i>]	[IDX2]	E7 xb ee ff	fIPrPf

TSTA

Test A
(CPU12, CPU12X)

TSTA

Operation

(A) – \$00

Description

Subtracts \$00 from the content of accumulator A and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying A.

The TSTA instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TSTA. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TSTA	INH	97	all 0

TSTB

Test B
(CPU12, CPU12X)

TSTB

Operation

(B) – \$00

Description

Subtracts \$00 from the content of accumulator B and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying B.

The TSTB instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TSTB. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$00; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TSTB	INH	D7	all O

TSTW

Test Memory for Zero or Minus (16 Bit)
(CPU12X)

TSTW

Operation

$$(M : M + 1) - 0$$

Description

Subtracts \$0000 from the content of memory location $M : M + 1$ and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying $M : M + 1$.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
			CPU12X
TSTW <i>opr16a</i>	EXT	18 F7 hh ll	ORPO
TSTW <i>opr0_xyyp</i>	IDX	18 E7 xb	ORPf
TSTW <i>opr9,xysp</i>	IDX1	18 E7 xb ff	ORPO
TSTW <i>opr16,xysp</i>	IDX2	18 E7 xb ee ff	OfRPP
TSTW [<i>D,xysp</i>]	[D,IDX]	18 E7 xb	OfIRPf
TSTW [<i>opr16,xysp</i>]	[IDX2]	18 E7 xb ee ff	OfIRPf

TSTX

Test X for Zero or Minus
(CPU12X)

TSTX

Operation

$(X) - 0$

Description

Subtracts \$0000 from the content of index register X and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying X.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail CPU12X
TSTX	INH	18 97	00

TSTY

Test Y for Zero or Minus (CPU12X)

TSTY

Operation

$(Y) - 0$

Description

Subtracts \$0000 from the content of index register Y and sets the condition codes accordingly.

The subtraction is accomplished internally without modifying Y.

The TST instruction provides limited information when testing unsigned values. Since no unsigned value is less than zero, BLO and BLS have no utility following TST. While BHI can be used after TST, it performs the same function as BNE, which is preferred. After testing signed values, all signed branches are available.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	Δ	Δ	0	0

N: Set if MSB of result is set; cleared otherwise

Z: Set if result is \$0000; cleared otherwise

V: 0; cleared

C: 0; cleared

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TSTY	INH	18 D7	CPU12X 00

TSX

Transfer from Stack Pointer
to Index Register X
(CPU12, CPU12X)

TSX

Operation

(SP) \Rightarrow X

Description

This is an alternate mnemonic to transfer the stack pointer value to index register X. The content of the SP remains unchanged. After a TSX instruction, X points at the last value that was stored on the stack.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TSX <i>translates to...</i> TFR SP,X	INH	B7 75	all P

TSY

Transfer from Stack Pointer
to Index Register Y
(CPU12, CPU12X)

TSY

Operation

(SP) \Rightarrow Y

Description

This is an alternate mnemonic to transfer the stack pointer value to index register Y. The content of the SP remains unchanged. After a TSY instruction, Y points at the last value that was stored on the stack.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TSY <i>translates to...</i> TFR SP,Y	INH	B7 76	all P

TXS

Transfer from Index Register X to Stack Pointer (CPU12, CPU12X)

TXS

Operation

$(X) \Rightarrow SP$

Description

This is an alternate mnemonic to transfer index register X value to the stack pointer. The content of X is unchanged.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TXS <i>translates to...</i> TFR X,SP	INH	B7 57	all P

TYS

Transfer from Index Register Y to Stack Pointer (CPU12, CPU12X)

TYS

Operation

 $(Y) \Rightarrow SP$

Description

This is an alternate mnemonic to transfer index register Y value to the stack pointer. The content of Y is unchanged.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
TYS <i>translates to...</i> TFR Y,SP	INH	B7 67	<div>all</div> <div>P</div>

WAI

Wait for Interrupt (CPU12, CPU12X)

WAI

Operation

$(SP) - \$0002 \Rightarrow SP; RTN_H : RTN_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; Y_H : Y_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; X_H : X_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

$(SP) - \$0002 \Rightarrow SP; B : A \Rightarrow (M_{(SP)} : M_{(SP+1)})$

In case of CPU12

$(SP) - \$0001 \Rightarrow SP; CCR \Rightarrow (M_{(SP)})$

In case of CPU12X

$(SP) - \$0002 \Rightarrow SP; CCR_H : CCR_L \Rightarrow (M_{(SP)} : M_{(SP+1)})$

Stop CPU Clocks

Description

Puts the CPU into a wait state. Uses the address of the instruction following WAI as a return address. Stacks the return address, index registers Y and X, accumulators B and A, and the CCR, decrementing the SP before each item is stacked.

The CPU then enters a wait state for an integer number of bus clock cycles. During the wait state, CPU clocks are stopped, but other MCU clocks can continue to run. The CPU leaves the wait state when it senses an interrupt that has not been masked.

If XIRQ is asserted while the X mask bit = 0 (XIRQ interrupts enabled), execution resumes with a vector fetch for the XIRQ interrupt. While the X mask bit = 1 (XIRQ interrupts disabled), a 2-cycle recovery sequence is used to adjust the instruction queue and the stack pointer, and execution continues with the next instruction after WAI.

CCR Details

S	X	H	I	N	Z	V	C
-	-	-	-	-	-	-	-

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail		
			CPU12	CPU12XV0 CPU12XV1	CPU12XV2
WAI (before interrupt)	INH	3E	OSSSSsF	OSSSSsF	
WAI (when interrupt comes)			fVfPPP	fVfPPP	
(continue)			ff	ff	
(CPU in user state)			NA	NA	0

WAV

Weighted Average (CPU12V0, CPU12XV0)

WAV

Operation

Do until B = 0, leave SOP in Y : D, SOW in X

Partial Product = (M pointed to by X) × (M pointed to by Y)

Sum-of-Products (24-bit SOP) = Previous SOP + Partial Product

Sum-of-Weights (16-bit SOW) = Previous SOW + (M pointed to by Y)

(X) + \$0001 ⇒ X; (Y) + \$0001 ⇒ Y

(B) – \$01 ⇒ B

Description

Performs weighted average calculations on values stored in memory. Uses indexed (X) addressing mode to reference one source operand list, and indexed (Y) addressing mode to reference a second source operand list. Accumulator B is used as a counter to control the number of elements to be included in the weighted average.

For each pair of data points, a 24-bit sum of products (SOP) and a 16-bit sum of weights (SOW) is accumulated in temporary registers. When B reaches zero (no more data pairs), the SOP is placed in Y : D. The SOW is placed in X. To arrive at the final weighted average, divide the content of Y : D by X by executing an EDIV after the WAV.

This instruction can be interrupted. If an interrupt occurs during WAV execution, the intermediate results (six bytes) are stacked in the order SOW_[15:0], SOP_[15:0], \$00:SOP_[23:16] before the interrupt is processed. The wavr pseudo-instruction is used to resume execution after an interrupt. The mechanism is re-entrant. New WAV instructions can be started and interrupted while a previous WAV instruction is interrupted.

This instruction is often used in fuzzy logic rule evaluation. Refer to [Section Chapter 9, “Fuzzy Logic Support”](#) for more information.

CCR Details

S	X	H	I	N	Z	V	C
–	–	?	–	?	1	?	?

Z: 1; set

H, N, V and C may be altered by this instruction

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail ¹ CPU12V0, CPU12XV0
WAV	Special	18 3C	Of (frr, ffff) O (replace comma if interrupted) SSS + UUrrr

¹ The replace comma sequence in parentheses represents the loop for one iteration of SOP and SOW accumulation.

XGDX

Exchange Double Accumulator
and Index Register X
(CPU12, CPU12X)

XGDX

Operation

(D) \Leftrightarrow (X)

Description

Exchanges the content of double accumulator D and the content of index register X. For compatibility with the M68HC11, the XGDX instruction is translated into an EXG D,X instruction by the assembler.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
XGDX <i>translates to...</i> EXG D,X	INH	B7 C5	<div>all</div> <div>P</div>

XGDY

Exchange Double Accumulator
and Index Register Y
(CPU12, CPU12X)

XGDY

Operation

(D) \Leftrightarrow (Y)

Description

Exchanges the content of double accumulator D and the content of index register Y. For compatibility with the M68HC11, the XGDY instruction is translated into an EXG D,Y instruction by the assembler.

CCR Details

S	X	H	I	N	Z	V	C
–	–	–	–	–	–	–	–

Detailed Syntax and Cycle-by-Cycle Operation

Source Form	Address Mode	Object Code	Access Detail
XGDY <i>translates to...</i> EXG D,Y	INH	B7 C6	<div>all</div> <div>P</div>



Chapter 7

Exception Processing

7.1 Introduction

Exceptions are events that require processing outside the normal flow of instruction execution. This chapter describes exceptions and the way each is handled.

7.2 Types of Exceptions

Central Processor Unit (CPU) exceptions on the CPU12 Family include:

- Resets
- An unimplemented opcode trap
- A software interrupt instruction (SWI) or BDM vector request
- A system call interrupt instruction (SYS) (CPU12XV1 and CPU12XV2 only)
- Non-maskable access error interrupts
- Non-maskable (X bit) interrupts
- Maskable (I bit) interrupts

Each exception has an associated 16-bit vector, which points to the memory location where the routine that handles the exception is located. The 16-bit exception vectors are taken from a vector table. For more details about the content and the location of the exception vector table please refer to the relevant chapters in the MCU reference manual of the device, specifically the chapters describing the Reset- and Interrupt-vectors in the device top-level and the interrupt module.

A CPU from the CPU12 Family can handle up to 128 exception vectors, but the number actually used varies from device to device, and some vectors are reserved for Freescale use.

Exceptions can be classified into different categories, depending on the effect of the different ways to mask interrupts.

- Resets. These exceptions are not maskable.
- Software exceptions. These include the unimplemented op-code trap, the SWI instruction, the SYS instruction and the BGND instruction. Software exceptions are not maskable.
- Non-maskable hardware interrupts. These include the access error interrupt from the Memory Protection Unit (MPU) and the software error interrupt from the XGATE coprocessor module. Please refer to the MCU reference manual if an MPU and/or an XGATE module is implemented.
- Interrupt service requests from the $\overline{\text{XIRQ}}$ pin. This exception can be masked with the X bit (X=1). The I bit and the IPL bits (on CPU12X) have no effect.

- All remaining interrupt service requests can be masked with the I bit (I=1) and are subject to priority filtering using the IPL bits (on CPU12X).

7.3 Exception Priority

A hardware priority hierarchy determines which reset or interrupt is serviced first when simultaneous requests are made. Refer to the Interrupt Module chapter in the MCU reference manual for more details concerning interrupt priority and servicing.

Resets share the highest exception-processing priority.

The unimplemented page 2 opcode trap (TRAP), the SYS and the SWI instruction are special cases. In one sense, these three exceptions have very low priority, because any enabled interrupt source that is pending prior to the time exception processing begins will take precedence. However, once the CPU begins processing a TRAP, SYS or SWI, neither can be interrupted. Also, since these are mutually exclusive instructions, they have no relative priority. An exception from this is an SWI (or BGND) execution forced by the Debug module (DBG) (please refer to the DBG chapter in the MCU reference manual for more information). In this case the SWI (or BGND) instruction takes precedence, even if a TRAP or SYS instruction is about to be executed.

The access error interrupts are non-maskable. They are generated by either the Memory Protection Unit (MPU) module, or by the XGATE coprocessor module to flag illegal memory accesses (Refer to the respective chapters in the MCU reference manual for details).

The $\overline{\text{XIRQ}}$ interrupt is pseudo-non-maskable. After reset, the X bit in the CCR is set, which inhibits all interrupt service requests from the $\overline{\text{XIRQ}}$ pin until the X bit is cleared. The X bit can be cleared by a program instruction, but program instructions cannot change X from 0 to 1. Once the X bit is cleared, interrupt service requests made via the $\overline{\text{XIRQ}}$ pin become non-maskable.

All remaining interrupts are subject to masking via the I bit in the CCR. Most microcontroller units (MCU) have an external $\overline{\text{IRQ}}$ pin, which is assigned the highest I bit interrupt priority and an internal periodic real-time interrupt generator, which has the next highest priority. The other maskable sources have default priorities that follow the address order of the interrupt vectors — the higher the address, the higher the priority of the interrupt. Other maskable interrupts are associated with on-chip peripherals such as timers or serial ports. On the CPU12 logic in the device integration module can give one I-masked source priority over other I-masked sources. On the CPU12X an interrupt controller will determine the priority of the interrupt based on the IPL bits and associated vector address. Refer to the MCU reference manual for a specific CPU12 and CPU12X derivative for more information.

7.4 Resets

CPU12 Family devices perform resets with a combination of hardware and software. Integration module circuitry determines the type of reset that has occurred, performs basic system configuration, then passes control to the CPU. The CPU fetches a vector determined by the type of reset that has occurred, jumps to the address pointed to by the vector, and begins to execute code at that address. For more information on possible causes of a reset and the associated reset vectors please refer to the MCU reference manual of the device.

7.5 Interrupts

Each CPU12 Family device can recognize a number of interrupt sources. Each source is associated with a vector in the vector table. The unimplemented opcode trap, the SYS instruction, the SWI instruction and the access violation interrupts are non-maskable, and have a fixed priority. The $\overline{\text{XIRQ}}$ signal is X bit-maskable. The remaining interrupt sources can be masked by the I bit. The I bit maskable interrupt sources have default priorities that follow the address order of the interrupt vectors. The higher the vector address, the higher the priority of the interrupt. On the CPU12 a device integration module incorporates logic that can give any one maskable source priority over other maskable sources. On the CPU12X the priority of each I bit maskable interrupt can be configured to one out of 7 possible priority levels, controlled by the IPL bits. Please refer to the interrupt module chapter in the MCU reference manual for details.

7.5.1 X-bit-Maskable Interrupt Request ($\overline{\text{XIRQ}}$)

The $\overline{\text{XIRQ}}$ input is an updated version of the non-maskable interrupt ($\overline{\text{NMI}}$) input of earlier MCUs. The $\overline{\text{XIRQ}}$ function is disabled during system reset and upon entering the interrupt service routine for an $\overline{\text{XIRQ}}$ interrupt.

During reset, both the I bit and the X bit in the CCR are set. This disables maskable interrupts and interrupt service requests made by asserting the $\overline{\text{XIRQ}}$ signal. After minimum system initialization, software can clear the X bit using an instruction such as `ANDCC #$BF`. Software cannot set the X bit from 0 to 1 once it has been cleared, and interrupt requests made via the $\overline{\text{XIRQ}}$ pin become non-maskable. When a non-maskable interrupt is recognized, both the X and I bits are set after context is saved. The X bit is not affected by maskable interrupts. Execution of an return-from-interrupt (RTI) instruction at the end of the interrupt service routine normally restores the X and I bits to the pre-interrupt request state.

7.5.2 I-bit-Maskable Interrupt Requests

Maskable interrupt sources include on-chip peripheral systems and external interrupt service requests. Interrupts from these sources are recognized when the global interrupt mask bit (I) in the CCR is cleared. The default state of the I bit out of reset is 1, but it can be written at any time if the CPU is not in user state.

The interrupt module manages maskable interrupt priorities. Typically, an on-chip interrupt source is subject to masking by associated bits in control registers in addition to global masking by the I bit in the CCR. Sources generally must be enabled by writing one or more bits in associated control registers. There may be other interrupt-related control bits and flags, and there may be specific register read-write sequences associated with interrupt service. Refer to individual on-chip peripheral descriptions for details.

7.5.3 Interrupt Recognition

Once enabled, an interrupt request can be recognized at any time after the I bit was cleared. When an interrupt service request is recognized, the CPU responds at the completion of the instruction being executed. Interrupt latency varies according to the number of cycles required to complete the current instruction. Because the fuzzy logic rule evaluation (REV), fuzzy logic rule evaluation weighted (REWV), and weighted average (WAV) instructions can take many cycles to complete, they are designed so that they can be interrupted. Instruction execution resumes when interrupt execution is complete. When the CPU begins to service an interrupt, the instruction queue is refilled, a return address is calculated, and then the

return address and the contents registers are stacked as shown in Table 7-1 for CPU12 and Table 7-2 for CPU12X.

Table 7-1. CPU12 Stacking Order on Entry to Interrupts

Memory Location	CPU12 Registers
SP + 7	RTN _H : RTN _L
SP + 5	Y _H : Y _L
SP + 3	X _H : X _L
SP + 1	B : A
SP	CCR

Table 7-2. CPU12X Stacking Order on Entry to Interrupts

Memory Location	CPU12 Registers
SP + 8	RTN _H : RTN _L
SP + 6	Y _H : Y _L
SP + 4	X _H : X _L
SP + 2	B : A
SP	CCR _H :CCR _L

After the CCR is stacked, the I bit (and the X bit, if an $\overline{\text{XIRQ}}$ interrupt service request caused the interrupt) is set to prevent other interrupts from disrupting the interrupt service routine. On CPU12XV2 the U bit is cleared to make sure the interrupt service routine is executed in supervisor state. Execution continues at the address pointed to by the vector for the highest-priority interrupt that was pending at the beginning of the interrupt sequence. At the end of the interrupt service routine, an RTI instruction restores context from the stacked registers, and normal program execution resumes.

7.5.4 Return-from-Interrupt Instruction (RTI)

RTI is used to terminate interrupt service routines. RTI is an 8-cycle instruction when no other interrupt is pending and 11 cycles, when another interrupt is pending. In either case, the first five cycles are used to restore (pull) the CCR, B:A, X, Y, and the return address from the stack. If no other interrupt is pending at this point, three program words are fetched to refill the instruction queue from the area of the return address and processing proceeds from there.

If another interrupt is pending after registers are restored, a new vector is fetched, and the stack pointer is adjusted to point at the CCR value that was just recovered ($\text{SP} = \text{SP} - 9$ for CPU12 and $\text{SP} = \text{SP} - 10$ for CPU12X). This makes it appear that the registers have been stacked again. After the SP is adjusted, three program words are fetched to refill the instruction queue, starting at the address the vector points to. Processing then continues with execution of the instruction that is now at the head of the queue.

7.6 Unimplemented Opcode Trap

The CPU12 has opcodes in all 256 positions in the page 1 opcode map, but only 54 of the 256 positions on page 2 of the opcode map are used. If the CPU12 attempts to execute one of the 202 unused opcodes

on page 2, an unimplemented opcode trap occurs. The 202 unimplemented opcodes are essentially interrupts that share a common interrupt vector, the un-implemented op-code trap.

The CPU12X has opcodes in all 256 positions in the page 1 opcode map, and 227 of the 256 positions on page 2 of the opcode map are used. If the CPU12X attempts to execute one of the 29 unused opcodes on page 2, an unimplemented opcode trap occurs. The 29 unimplemented opcodes are essentially interrupts that share a common interrupt vector, the un-implemented op-code trap.

The CPU uses the next address after an unimplemented page 2 opcode as a return address. The stacked return address can be used to calculate the address of the unimplemented opcode for software-controlled traps.

7.7 Software Interrupt Instruction (SWI)

Execution of the SWI instruction causes an interrupt without an interrupt service request. SWI cannot be masked by the global mask bits in the CCR, and execution of SWI sets the I mask bit. Once an SWI interrupt begins, maskable interrupts are inhibited until the I bit in the CCR is cleared. This typically occurs when an RTI instruction at the end of the SWI service routine restores context.

7.8 System Call Instruction (SYS) (CPU12XV1 & CPU12XV2)

Execution of the SYS instruction causes an interrupt without an interrupt service request. SYS is not inhibited by the global mask bits in the CCR, and execution of SYS sets the I bit. Once an SYS interrupt begins, maskable interrupts are inhibited until the I bit in the CCR is cleared. This typically occurs when an RTI instruction at the end of the SYS service routine restores context.

7.9 Exception Processing Flow

The first cycle in the exception processing flow for all CPU12 Family exceptions is the same, regardless of the source of the exception. Between the first and second cycles of execution, the CPU chooses one of three alternative paths. The first path is for resets, the second path is for pending X or I interrupts, and the third path is used for software interrupts (SWI) and trapping unimplemented opcodes. The last two paths are virtually identical, differing only in the details of calculating the return address. Refer to [Figure 7-1](#) for the following description of events.

7.9.1 Vector Fetch

The first cycle of all exception processing, regardless of the cause, is a vector fetch. The vector points to the address where exception processing will continue. Exception vectors are stored in a table located at the top of the memory map (\$FFxx) if not placed else where using the Interrupt Vector Base Register (CPU12X only). The CPU12 cannot use the fetched vector until the third cycle of the exception processing sequence.

On the CPU12XV2, supervisor state is forced before the vector fetch cycle to ensure the entire exception processing sequence takes place in supervisor state. This is independent from the actual clearing of the U bit which during an interrupt sequence does not happen until the CCRW register was stacked.

During the vector fetch cycle, the CPU issues a signal that tells the interrupt module to drive the vector address of the highest priority, pending exception onto the system address bus (the CPU12 does not provide this address).

After the vector fetch, the CPU selects one of the three alternate execution paths, depending upon the cause of the exception.

7.9.2 Reset Exception Processing

If reset caused the exception, processing continues to cycle 2.0. This cycle sets the S, X, and I bits and clears the U and IPL[2:0] bits (CPU12X) in the CCRH. Cycles 3.0 through 5.0 are program word fetches that refill the instruction queue. Fetches start at the address pointed to by the reset vector. When the fetches are completed, exception processing ends, and the CPU starts executing the instruction at the head of the instruction queue.

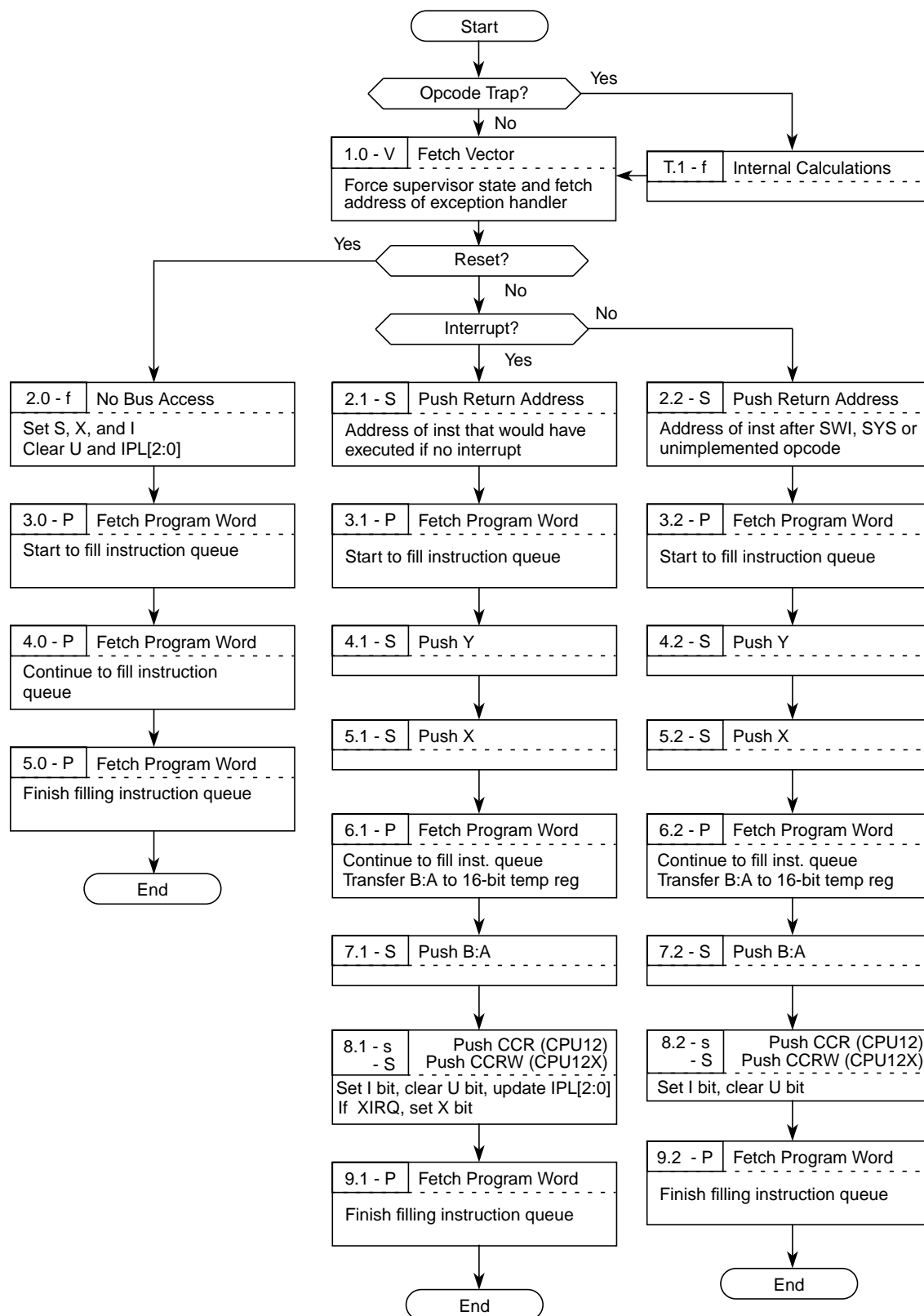


Figure 7-1. Exception Processing Flow Diagram

7.9.3 Interrupt and Unimplemented Opcode Trap Exception Processing

If an exception was not caused by a reset, a return address is calculated.

- Cycles 2.1 and 2.2 are both S cycles (stack a 16-bit word), but the CPU performs different return address calculations for each type of exception.
 - When an X- or I-related interrupt causes the exception, the return address points to the next instruction that would have been executed had processing not been interrupted.
 - When an exception is caused by an SWI opcode or by an unimplemented opcode (see [Section 7.6, “Unimplemented Opcode Trap”](#)), the return address points to the next address after the opcode.
- Once calculated, the return address is pushed onto the stack.
- Cycles 3.1 through 9.1 are identical to cycles 3.2 through 9.2 for the rest of the sequence, except for optional setting of the X mask bit performed in cycle 8.1 (see below).
- Cycle 3.1/3.2 is the first of three program word fetches that refill the instruction queue.
- Cycle 4.1/4.2 pushes Y onto the stack.
- Cycle 5.1/5.2 pushes X onto the stack.
- Cycle 6.1/6.2 is the second of three program word fetches that refill the instruction queue. During this cycle, the contents of the A and B accumulators are concatenated into a 16-bit word in the order B:A.
- Cycle 7.1/7.2 pushes the 16-bit word containing B:A onto the stack.
- Cycle 8.1/8.2 pushes the 8-bit CCR (CPU12) respectively a 16-bit CCRW (CPU12X) onto the stack, then updates the mask bits.
 - When an $\overline{\text{XIRQ}}$ interrupt causes an exception, both X and I are set, which inhibits further interrupts during exception processing.
 - When any other interrupt causes an exception, the I bit is set, but the X bit is not changed.
 - The IPL[2:0] bits are updated (CPU12X) and the U bit is cleared (CPU12XV2 only)
- Cycle 9.1/9.2 is the third of three program word fetches that refill the instruction queue. It is the last cycle of exception processing. After this cycle the CPU starts executing the first cycle of the instruction at the head of the instruction queue.

Chapter 8 Debugging Support

8.1 Introduction

This section describes development and debug support features built into a Central Processor Unit from the CPU12 Family. Topics include:

- Instruction queue operation and reconstruction
- Instruction tagging

8.2 External Reconstruction of the Queue

The CPU12 Family uses an instruction queue to buffer program information and increase instruction throughput. The CPU queue consists of three 16-bit stages. Program information is always fetched in aligned 16-bit words. At least three bytes of program information are available to the CPU when instruction execution begins.

Because of the queue, program information is fetched a few cycles before it is used by the CPU. Internally, the microcontroller unit (MCU) only needs to buffer the fetched data. But, in order to monitor cycle-by-cycle CPU activity externally, it is necessary to capture data and address to discern what is happening in the instruction queue.

External pins, (IPIPE[1:0] for CPU12), (IQSTAT[3:0] for CPU12X), provide information about data movement in the queue and instruction execution. The instruction queue and cycle-by-cycle activity can be reconstructed in real time or from trace history captured by a logic analyzer. However, neither scheme can be used to stop the CPU at a specific instruction. By the time an operation is visible outside the MCU, the instruction has already begun execution. A separate instruction tagging mechanism is provided for this purpose. A tag follows the information in the queue as the queue is advanced. During debugging, the CPU enters active background debug mode when a tagged instruction reaches the head of the queue, rather than executing the tagged instruction. For more information about tagging, refer to [Section 8.5, “Instruction Tagging \(CPU12\)”](#).

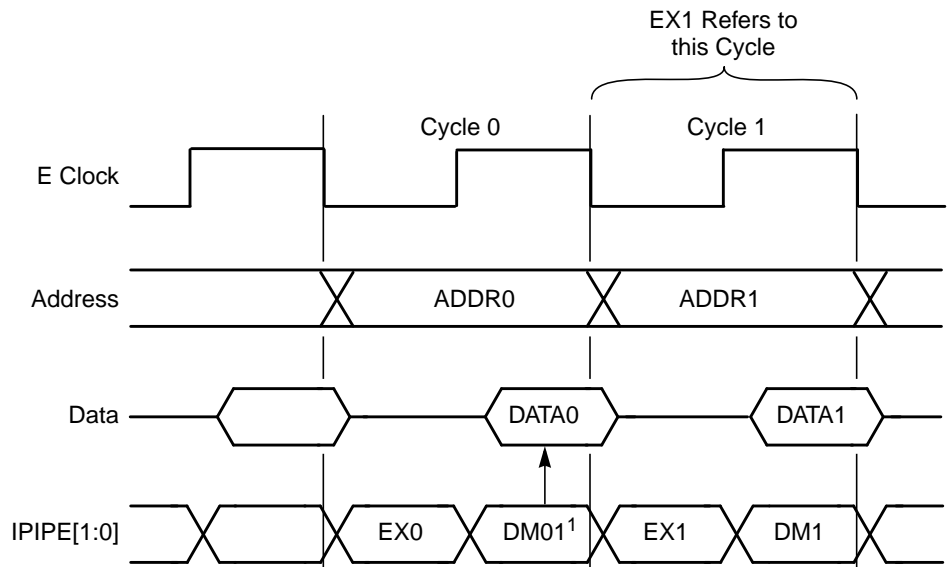
8.3 Instruction Queue Status Signals

The (IPIPE[1:0] for CPU12), (IQSTAT[3:0] for CPU12X) signals carry information about data movement and instruction execution during normal CPU operation.

To reconstruct the queue, the information carried by the status signals must be captured externally. The definition of the this signals is different from CPU12 and CPU12X, refer to [Section 8.3.1, “CPU12 Timing Detail”](#) and [Section 8.3.2, “CPU12X Timing Detail”](#).

8.3.1 CPU12 Timing Detail

In the CPU12, data-movement information is available when E clock is high or on falling edges of the E clock; execution-start information is available when E clock is low or on rising edges of the E clock, as shown in Figure 8-1. Data-movement information refers to data on the bus. Execution-start information refers to the bus cycle that starts with that E-low time and continues through the following E-high time. Table 8-1 summarizes the information encoded on the IPIPE1 and IPIPE0 pins.



1. DM0 refers to data captured at the end of current E-high period.

Figure 8-1. Queue Status Signal Timing (CPU12)

Table 8-1. IPIPE1 and IPIPE0 Decoding CPU12

	Mnemonic	Meaning
Data Movement	Capture at E Fall in CPU12	
0:0	—	No movement
0:1	—	Unused?
1:0	ALD	Advance queue and load from bus
1:1		
Execution Start	Capture at E Rise in CPU12	
0:0	—	No start
0:1	INT	Start interrupt sequence
1:0	SEV	Start even instruction
1:1	SOD	Start odd instruction

8.3.2 CPU12X Timing Detail

In the CPU12X, data-movement information and execution-start information as shown in Figure 8-2 are demultiplexed and available on the signal IQSTAT[3:0] when the next E clock is low, as shown in Figure 8-3. Data-movement information refers to data on the previous two bus cycles. Execution-start information refers to the previous bus cycle. Table 8-2 summarizes the information on the IQSTAT[3:0].

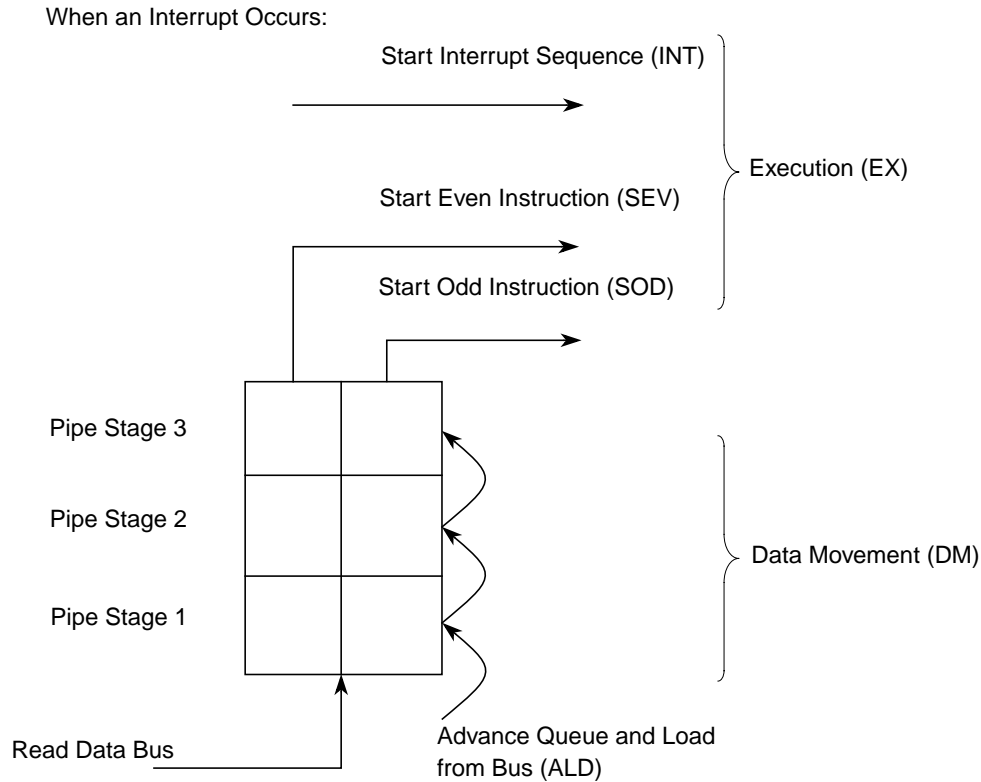
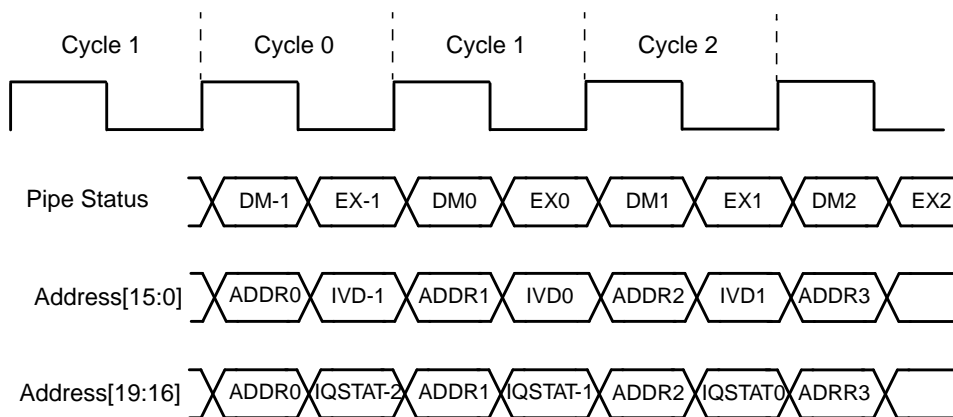


Figure 8-2. Pipe Status Signal



Note: IQSTAT contains data movement of the pipe in time T-2 (ALD) and/or the corresponding execution information in T-1 (INT, SEV, or SOD)

Figure 8-3. IQSTAT Timing

8.3.3 Null

The (00 for CPU12) (0000 for CPU12X) data movement state indicates that there was no data movement in the instruction queue; the (00 for CPU12) (0000 for CPU12X) execution start state indicates continuation of an instruction or interrupt sequence (no new instruction or interrupt start).

8.3.4 ALD — Advance and Load from Data Bus

The instruction queue is advanced by one word and stage one is refilled with a word of program information from the data bus. The CPU requested the information two bus cycles earlier but, due to access delays, the information was not available until the E cycle referred to by the ALD code.

8.3.5 INT — Interrupt Sequence Start

The E cycle associated with this code is the first cycle after an interrupt sequence. Normally, this cycle is one cycle after a read of the interrupt vector. However, in systems that have interrupt vectors in external memory and an 8-bit data bus, the cycle before this code reads the upper byte of the 16-bit interrupt vector.

Table 8-2. IQSTAT[3:0] on CPU12XV0

	Mnemonic	Meaning
Execution Start	Capture at E Fall in CPU12X	
0001	INT	Start interrupt sequence
0010	SOD	Start instruction at odd address
0100	SEV	Start instruction at even address
Data Movement	Capture at E Fall in CPU12X	
1000	ALD	Advance the instruction queue and load first stage
Data Movement and Execution Start	Capture at E Fall in CPU12X	
0000	Null	No movement, No start
1100	ALD&SEV	Advance the instruction queue and load first stage then Start instruction at even address
1010	ALD&SOD	Advance the instruction queue and load first stage then Start instruction at odd address
1001	ALD&INT	Advance the instruction queue and load first stage then Start interrupt sequence
Others	Not Implemented	—

8.3.6 SEV — Start Instruction on Even Address

The E cycle associated with this code is the first cycle after the instruction in the even (high order) half of the word at the head of the instruction queue. The queue treats the \$18 prebyte for instructions on page 2 of the opcode map as a special 1-byte, 1-cycle instruction, except that interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

8.3.7 SOD — Start Instruction on Odd Address

The E cycle associated with this code is the first cycle after the instruction in the odd (low order) half of the word at the head of the instruction queue. The queue treats the \$18 prebyte for instructions on page 2 of the opcode map as a special 1-byte, 1-cycle instruction, except that interrupts are not recognized at the boundary between the prebyte and the rest of the instruction.

8.4 Queue Reconstruction (for CPU12)

The raw signals required for queue reconstruction are the address bus (ADDR), the data bus (DATA), the system clock (E), and the queue status signals (IPIPE1 and IPIPE2). An ALD data movement implies a read; therefore, it is not necessary to capture the R/\overline{W} signal. An E clock cycle begins at a falling edge of E. Addresses and execution status must be captured at the rising E edge in the middle of the cycle. Data and data-movement status must be captured at the falling edge of E at the end of the cycle. These captures can then be organized into records with one record per E clock cycle.

Implementation details depend on the type of MCU and the mode of operation. For instance, the data bus can be eight bits or 16 bits wide, and nonmultiplexed or multiplexed. In all cases, the externally reconstructed queue must use 16-bit words. Demultiplexing and assembly of 8-bit data into 16-bit words is done before program information enters the real queue, so it must also be done for the external reconstruction.

An example:

Systems with an 8-bit data bus and a program stored in external memory require two cycles for each program word fetch. MCU bus-control logic freezes the CPU12 clocks long enough to do two 8-bit accesses rather than a single 16-bit access, so the CPU12 sees only 16-bit words of program information. To recover the 16-bit program words externally, latch the data bus state at the falling edge of E when ADDR0 = 0, and gate the outputs of the latch onto DATA[15:8] when an ALD cycle occurs. Since the 8-bit data bus is connected to DATA[7:0], the 16-bit word on the data lines corresponds to the ALD during the last half of the second 8-bit fetch, which is always to an odd address. IPIPE[1:0] status signals indicate 0:0 for the second half of the E cycle corresponding to the first 8-bit fetch.

Some MCUs have address lines to support memory expansion beyond the standard 64KB address space. When memory expansion is used, expanded addresses must also be captured and maintained.

8.4.1 Queue Reconstruction Registers (for CPU12)

Queue reconstruction requires the following registers, which can be implemented as software variables when previously captured trace data is used, or as hardware latches in real time.

8.4.1.1 `fetch_add` Register

This register buffers the fetch address.

8.4.1.2 `st1_add`, `st1_dat` Registers

These registers contain address and data for the first stage of the reconstructed instruction queue.

8.4.1.3 `st2_add`, `st2_dat` Registers

These registers contain address and data for the middle stage of the reconstructed instruction queue.

8.4.1.4 `st3_add`, `st3_dat` Registers

These registers contain address and data for the final stage of the reconstructed instruction queue. When the IPIPE[1:0] signals indicate the execution status, the address and opcode can be found in these registers.

8.5 Instruction Tagging (CPU12)

The instruction queue and cycle-by-cycle CPU12 activity can be reconstructed in real time or from trace history that was captured by a logic analyzer. However, the reconstructed queue cannot be used to stop the CPU12 at a specific instruction, because execution has already begun by the time an operation is visible outside the MCU. A separate instruction tagging mechanism is provided for this purpose.

Executing the BDM TAGGO command configures two MCU pins for tagging. The $\overline{\text{TAGLO}}$ signal shares a pin with the $\overline{\text{LSTRB}}$ signal, and the $\overline{\text{TAGHI}}$ signal shares the BKGD pin. Tagging information is latched on the falling edge of ECLK, as shown in Figure 8-4.

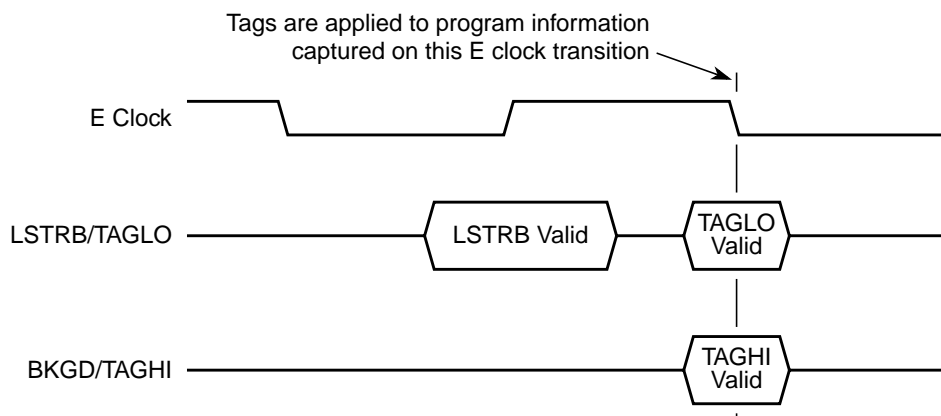


Figure 8-4. Tag Input Timing (CPU12)

Table 8-3 shows the functions of the two independent tagging pins. The presence of logic level 0 on either pin at the fall of ECLK tags (marks) the associated byte of program information as it is read into the instruction queue. Tagging is allowed in all modes. Tagging is disabled when BDM becomes active.

Table 8-3. Tag Pin Function (CPU12)

TAGHI	TAGLO	Tag
1	1	No tag
1	0	Low byte
0	1	High byte
0	0	Both bytes

In CPU12 derivatives that have hardware breakpoint capability, the breakpoint control logic and BDM control logic use the same internal signals for instruction tagging. The CPU12 does not differentiate between the two kinds of tags.

The tag follows program information as it advances through the queue. When a tagged instruction reaches the head of the queue, the CPU12 enters active background debug mode rather than executing the instruction.

8.6 Instruction Tagging (CPU12X)

The instruction queue and cycle-by-cycle CPU12 activity can be reconstructed in real time or from trace history that was captured by a logic analyzer. However, the reconstructed queue cannot be used to stop the CPU12X at a specific instruction, because execution has already begun by the time an operation is visible outside the MCU. A separate instruction tagging mechanism is provided for this purpose.

The TAGLO signal shares a pin with the RE and MODA signals, and the TAGHI signal shares the MODB pin. Tagging information is latched on the rising edge of ECLK, as shown in Figure 8-5.

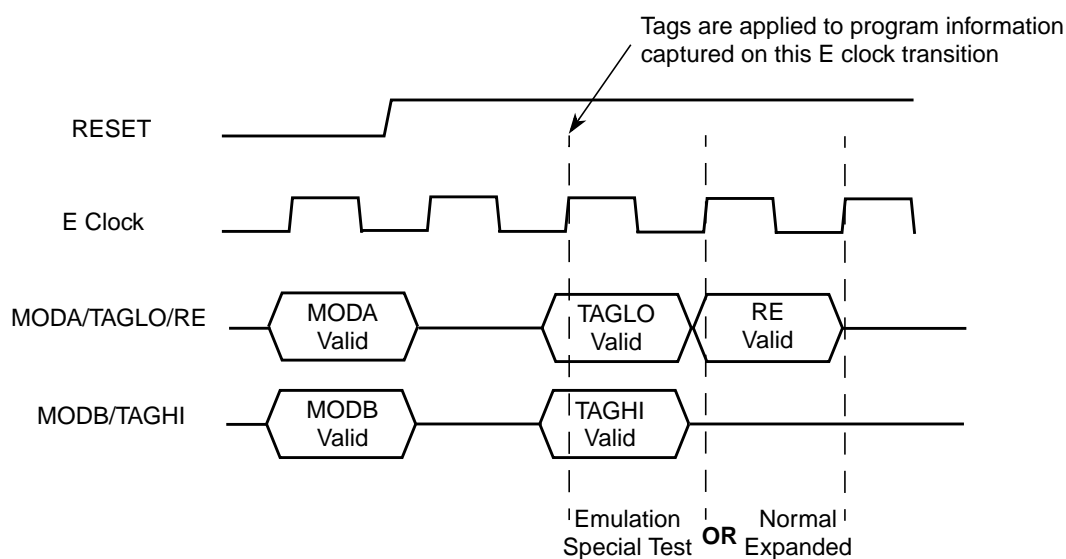


Figure 8-5. Tag Input Timing (CPU12X)

Table 8-4 shows the functions of the two independent tagging pins. The presence of logic level 0 on either pin at the rise of ECLK tags (marks) the associated byte of program information as it is read into the instruction queue. Tagging is allowed only in emulation modes. Tagging is disabled when BDM becomes active.

Table 8-4. Tag Pin Function (CPU12X)

TAGHI	TAGLO	Tag
1	1	No tag
1	0	Low byte
0	1	High byte
0	0	Both bytes

On the CPU12X internal breakpoints can also be generated by the DBG module. Breakpoints generated by the TAGLO or TAGHI have a higher priority than the internally generated breakpoints.

The tag follows program information as it advances through the queue. When a tagged instruction reaches the head of the queue, a tag hit occurs generating a hardware breakpoint to BDM or SWI.

Chapter 9

Fuzzy Logic Support

NOTE

The four Fuzzy instructions (MEM, REV, REVW, WAV/WAVR) are removed on CPU12V1, CPU12XV1 and CPU12XV2.

9.1 Introduction

The instruction set of the CPU12 Family is the first instruction set to specifically address the needs of fuzzy logic. This section describes the use of fuzzy logic in control systems, discusses the CPU12 Family fuzzy logic instructions, and provides examples of fuzzy logic programs.

The CPU12 Family includes four instructions that perform specific fuzzy logic tasks. In addition, several other instructions are especially useful in fuzzy logic programs. The overall C-friendliness of the instruction set also aids development of efficient fuzzy logic programs.

This section explains the basic fuzzy logic algorithm for which the four fuzzy logic instructions are intended. Each of the fuzzy logic instructions are then explained in detail. Finally, other custom fuzzy logic algorithms are discussed, with emphasis on use of other CPU instructions.

The four fuzzy logic instructions are:

- MEM (determine grade of membership), which evaluates trapezoidal membership functions
- REV (fuzzy logic rule evaluation) and REVW (fuzzy logic rule evaluation weighted), which perform unweighted or weighted MIN-MAX rule evaluation
- WAV (weighted average), which performs weighted average defuzzification on singleton output membership functions.

Other instructions that are useful for custom fuzzy logic programs include:

- MINA (place smaller of two unsigned 8-bit values in accumulator A)
- EMIND (place smaller of two unsigned 16-bit values in accumulator D)
- MAXM (place larger of two unsigned 8-bit values in memory)
- EMAXM (place larger of two unsigned 16-bit values in memory)
- TBL (table lookup and interpolate)
- ETBL (extended table lookup and interpolate)
- EMACS (extended multiply and accumulate signed 16-bit by 16-bit to 32-bit)

For higher resolution fuzzy programs, the fast extended precision math instructions in the CPU12 Family instruction set are also beneficial. Flexible indexed addressing modes help simplify access to fuzzy logic data structures stored as lists or tabular data structures in memory.

The actual logic additions required to implement fuzzy logic support in the CPU12 Family are quite small, so there is no appreciable increase in cost for the typical user. A fuzzy inference kernel for the CPU12 Family requires one-fifth as much code space and executes almost 50 times faster than a comparable kernel implemented on a typical midrange microcontroller.

9.2 Fuzzy Logic Basics

This is an overview of basic fuzzy logic concepts. It can serve as a general introduction to the subject, but that is not the main purpose. There are a number of fuzzy logic programming strategies. This discussion concentrates on the methods implemented in the CPU12 Family fuzzy logic instructions. The primary goal is to provide a background for a detailed explanation of the CPU12 Family fuzzy logic instructions.

In general, fuzzy logic provides for set definitions that have fuzzy boundaries rather than the crisp boundaries of Aristotelian logic. These sets can overlap so that, for a specific input value, one or more sets associated with linguistic labels may be true to a degree at the same time. As the input varies from the range of one set into the range of an adjacent set, the first set becomes progressively less true while the second set becomes progressively more true.

Fuzzy logic has membership functions which emulate human concepts like “temperature is warm”; that is, conditions are perceived to have gradual boundaries. This concept seems to be a key element of the human ability to solve certain types of complex problems that have eluded traditional control methods.

Fuzzy sets provide a means of using linguistic expressions like “temperature is warm” in rules which can then be evaluated with a high degree of numerical precision and repeatability. This directly contradicts the common misperception that fuzzy logic produces approximate results — a specific set of input conditions always produces the same result, just as a conventional control system does.

A microcontroller-based fuzzy logic control system has two parts:

- A fuzzy inference kernel which is executed periodically to determine system outputs based on current system inputs
- A knowledge base which contains membership functions and rules

Figure 9-1 is a block diagram of this kind of fuzzy logic system.

The knowledge base can be developed by an application expert without any microcontroller programming experience. Membership functions are simply expressions of the expert’s understanding of the linguistic terms that describe the system to be controlled. Rules are ordinary language statements that describe the actions a human expert would take to solve the application problem.

Rules and membership functions can be reduced to relatively simple data structures (the knowledge base) stored in non-volatile memory. A fuzzy inference kernel can be written by a programmer who does not know how the application system works. The only thing the programmer needs to do with knowledge base information is store it in the memory locations used by the kernel.

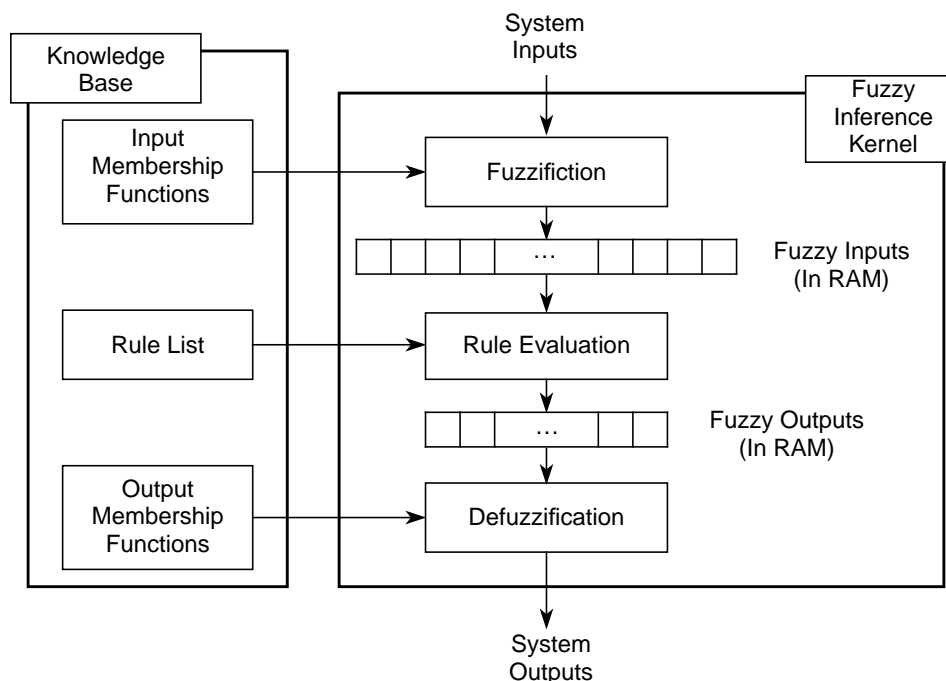


Figure 9-1. Block Diagram of a Fuzzy Logic System

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; delaying too long between passes can cause the system to get too far out of control. Choosing a periodic rate for a fuzzy control system is the same as it would be for a conventional control system.

9.2.1 Fuzzification (MEM)

During the fuzzification step, the current system input values are compared against stored input membership functions to determine the degree to which each label of each system input is true. This is accomplished by finding the y-value for the current input value on a trapezoidal membership function for each label of each system input. The MEM instruction in the CPU performs this calculation for one label of one system input. To perform the complete fuzzification task for a system, several MEM instructions must be executed, usually in a program loop structure.

Figure 9-2 shows a system of three input membership functions, one for each label of the system input. The x-axis of all three membership functions represents the range of possible values of the system input. The vertical line through all three membership functions represents a specific system input value. The y-axis represents degree of truth and varies from completely false (\$00 or 0 percent) to completely true (\$FF or 100 percent). The y-value where the vertical line intersects each of the membership functions, is the degree to which the current input value matches the associated label for this system input. For example, the expression “temperature is warm” is 25 percent true (\$40). The value \$40 is stored to a random-access memory (RAM) location and is called a fuzzy input (in this case, the fuzzy input for “temperature is warm”). There is a RAM location for each fuzzy input (for each label of each system input).

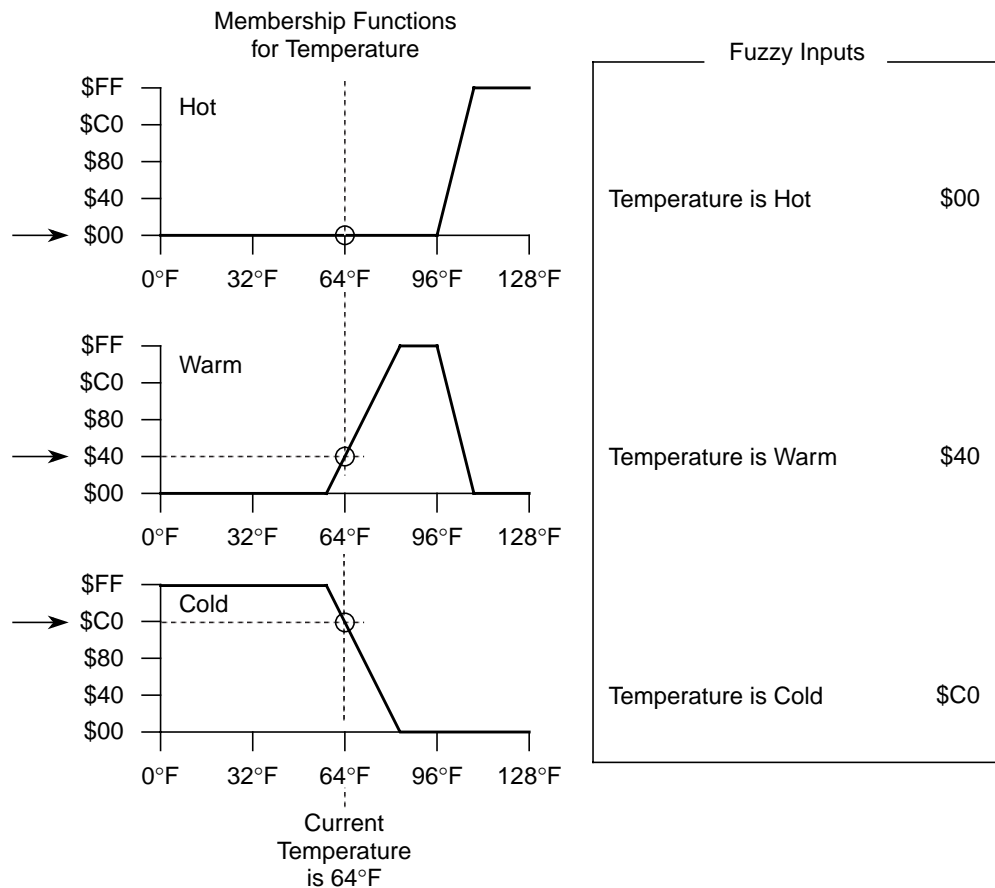


Figure 9-2. Fuzzification Using Membership Functions

When the fuzzification step begins, the current value of the system input is in an accumulator of the CPU, one index register points to the first membership function definition in the knowledge base, and a second index register points to the first fuzzy input in RAM. As each fuzzy input is calculated by executing a MEM instruction, the result is stored to the fuzzy input and both pointers are updated automatically to point to the locations associated with the next fuzzy input. The MEM instruction takes care of everything except counting the number of labels per system input and loading the current value of any subsequent system inputs.

The end result of the fuzzification step is a table of fuzzy inputs representing current system conditions.

9.2.2 Rule Evaluation (REV and REVW)

Rule evaluation is the central element of a fuzzy logic inference program. This step processes a list of rules from the knowledge base using current fuzzy input values from RAM to produce a list of fuzzy outputs in RAM. These fuzzy outputs can be thought of as raw suggestions for what the system output should be in response to the current input conditions. Before the results can be applied, the fuzzy outputs must be further processed, or defuzzified, to produce a single output value that represents the combined effect of all of the fuzzy outputs.

The CPU12 Family offers two variations of rule evaluation instructions. The REV instruction provides for unweighted rules (all rules are considered to be equally important). The REVW instruction is similar but allows each rule to have a separate weighting factor which is stored in a separate parallel data structure in the knowledge base. In addition to the weights, the two rule evaluation instructions also differ in the way rules are encoded into the knowledge base.

An understanding of the structure and syntax of rules is needed to understand how a microcontroller performs the rule evaluation task. An example of a typical rule is:

If temperature is warm and pressure is high, then heat is (should be) off.

At first glance, it seems that encoding this rule in a compact form understandable to the microcontroller would be difficult, but it is actually simple to reduce the rule to a small list of memory pointers. The antecedent portion of the rule is a statement of input conditions and the consequent portion of the rule is a statement of output actions.

The antecedent portion of a rule is made up of one or more (in this case two) antecedents connected by a fuzzy *and* operator. Each antecedent expression consists of the name of a system input, followed by *is*, followed by a label name. The label must be defined by a membership function in the knowledge base. Each antecedent expression corresponds to one of the fuzzy inputs in RAM. Since *and* is the only operator allowed to connect antecedent expressions, there is no need to include these in the encoded rule. The antecedents can be encoded as a simple list of pointers to (or addresses of) the fuzzy inputs to which they refer.

The consequent portion of a rule is made up of one or more (in this case one) consequents. Each consequent expression consists of the name of a system output, followed by *is*, followed by a label name. Each consequent expression corresponds to a specific fuzzy output in RAM. Consequents for a rule can be encoded as a simple list of pointers to (or addresses of) the fuzzy outputs to which they refer.

The complete rules are stored in the knowledge base as a list of pointers or addresses of fuzzy inputs and fuzzy outputs. For the rule evaluation logic to work, there must be some means of knowing which pointers refer to fuzzy inputs and which refer to fuzzy outputs. There also must be a way to know when the last rule in the system has been reached.

- One method of organization is to have a fixed number of rules with a specific number of antecedents and consequents.
- A second method, employed in Freescale Freeware M68HC11 kernels, is to mark the end of the rule list with a reserved value, and use a bit in the pointers to distinguish antecedents from consequents.
- A third method of organization, used in the CPU12 Family, is to mark the end of the rule list with a reserved value, and separate antecedents and consequents with another reserved value. This permits any number of rules, and allows each rule to have any number of antecedents and consequents, subject to the limits imposed by availability of system memory.

Each rule is evaluated sequentially, but the rules as a group are treated as if they were all evaluated simultaneously. Two mathematical operations take place during rule evaluation. The fuzzy *and* operator corresponds to the mathematical minimum operation and the fuzzy *or* operation corresponds to the mathematical maximum operation. The fuzzy *and* is used to connect antecedents within a rule. The fuzzy *or* is implied between successive rules. Before evaluating any rules, all fuzzy outputs are set to zero

(meaning not true at all). As each rule is evaluated, the smallest (minimum) antecedent is taken to be the overall truth of the rule. This rule truth value is applied to each consequent of the rule (by storing this value to the corresponding fuzzy output) unless the fuzzy output is already larger (maximum). If two rules affect the same fuzzy output, the rule that is most true governs the value in the fuzzy output because the rules are connected by an implied fuzzy *or*.

In the case of rule weighting, the truth value for a rule is determined as usual by finding the smallest rule antecedent. Before applying this truth value to the consequents for the rule, the value is multiplied by a fraction from zero (rule disabled) to one (rule fully enabled). The resulting modified truth value is then applied to the fuzzy outputs.

The end result of the rule evaluation step is a table of suggested or “raw” fuzzy outputs in RAM. These values were obtained by plugging current conditions (fuzzy input values) into the system rules in the knowledge base. The raw results cannot be supplied directly to the system outputs because they may be ambiguous. For instance, one raw output can indicate that the system output should be medium with a degree of truth of 50 percent while, at the same time, another indicates that the system output should be low with a degree of truth of 25 percent. The defuzzification step resolves these ambiguities.

9.2.3 Defuzzification (WAV)

The final step in the fuzzy logic program combines the raw fuzzy outputs into a composite system output. Unlike the trapezoidal shapes used for inputs, the CPU12 Family typically uses singletons for output membership functions. As with the inputs, the x-axis represents the range of possible values for a system output. Singleton membership functions consist of the x-axis position for a label of the system output. Fuzzy outputs correspond to the y-axis height of the corresponding output membership function.

The WAV instruction calculates the numerator and denominator sums for weighted average of the fuzzy outputs according to the formula:

$$\text{System Output} = \frac{\sum_{i=1}^n S_i F_i}{\sum_{i=1}^n F_i}$$

Where n is the number of labels of a system output, S_i are the singleton positions from the knowledge base, and F_i are fuzzy outputs from RAM. For a common fuzzy logic program on the CPU12 Family, n is eight or less (though this instruction can handle any value to 255) and S_i and F_i are 8-bit values. The final divide is performed with a separate EDIV instruction placed immediately after the WAV instruction.

Before executing WAV, an accumulator must be loaded with the number of iterations (n), one index register must be pointed at the list of singleton positions in the knowledge base, and a second index register must be pointed at the list of fuzzy outputs in RAM. If the system has more than one system output, the WAV instruction is executed once for each system output.

9.3 Example Inference Kernel

Figure 9-3 is a complete fuzzy inference kernel written in CPU12 Family assembly language. Numbers in square brackets are cycle counts for an CPU12 Family device. The kernel uses two system inputs with seven labels each and one system output with seven labels. The program assembles to 57 bytes. It executes in about 20 μ s at an 25-MHz bus rate. The basic structure can easily be extended to a general-purpose system with a larger number of inputs and outputs.

```

*
01 [2]    FUZZIFY    LDX    #INPUT_MFS    ;Point at MF definitions
02 [2]    LDY    #FUZ_INS    ;Point at fuzzy input table
03 [3]    LDAA    CURRENT_INS    ;Get first input value
04 [1]    LDAB    #7    ;7 labels per input
05 [5]    GRAD_LOOP    MEM    ;Evaluate one MF
06 [3]    DBNE    B,GRAD_LOOP    ;For 7 labels of 1 input
07 [3]    LDAA    CURRENT_INS+1    ;Get second input value
08 [1]    LDAB    #7    ;7 labels per input
09 [5]    GRAD_LOOP1    MEM    ;Evaluate one MF
10 [3]    DBNE    B,GRAD_LOOP1    ;For 7 labels of 1 input

11 [1]    LDAB    #7    ;Loop count
12 [2]    RULE_EVAL    CLR    1,Y+    ;Clr a fuzzy out & inc ptr
13 [3]    DBNE    b,RULE_EVAL    ;Loop to clr all fuzzy outs
14 [2]    LDX    #RULE_START    ;Point at first rule element
15 [2]    LDY    #FUZ_INS    ;Point at fuzzy ins and outs
16 [1]    LDAA    #$FF    ;Init A (and clears V-bit)
17 [3n+4]    REV    ;Process rule list

18 [2]    DEFUZ    LDY    #FUZ_OUT    ;Point at fuzzy outputs
19 [2]    LDX    #SGLTN_POS    ;Point at singleton positions
20 [1]    LDAB    #7    ;7 fuzzy outs per COG output
21 [7b+4]    WAV    ;Calculate sums for wtd av
22 [11]    EDIV    ;Final divide for wtd av
23 [1]    TFR    Y,D    ;Move result to A:B
24 [3]    STAB    COG_OUT    ;Store system output
*
***** End

```

Figure 9-3. Fuzzy Inference Engine

Lines 1 to 3 set up pointers and load the system input value into the A accumulator.

Line 4 sets the loop count for the loop in lines 5 and 6.

Lines 5 and 6 make up the fuzzification loop for seven labels of one system input. The MEM instruction finds the y-value on a trapezoidal membership function for the current input value, for one label of the current input, and then stores the result to the corresponding fuzzy input. Pointers in X and Y are automatically updated by four and one so they point at the next membership function and fuzzy input respectively.

Line 7 loads the current value of the next system input. Pointers in X and Y already point to the right places as a result of the automatic update function of the MEM instruction in line 5.

Line 8 reloads a loop count.

Lines 9 and 10 form a loop to fuzzify the seven labels of the second system input. When the program drops to line 11, the Y index register is pointing at the next location after the last fuzzy input, which is the first fuzzy output in this system.

Line 11 sets the loop count to clear seven fuzzy outputs.

Lines 12 and 13 form a loop to clear all fuzzy outputs before rule evaluation starts.

Line 14 initializes the X index register to point at the first element in the rule list for the REV instruction.

Line 15 initializes the Y index register to point at the fuzzy inputs and outputs in the system. The rule list (for REV) consists of 8-bit offsets from this base address to particular fuzzy inputs or fuzzy outputs. The special value \$FE is interpreted by REV as a marker between rule antecedents and consequents.

Line 16 initializes the A accumulator to the highest 8-bit value in preparation for finding the smallest fuzzy input referenced by a rule antecedent. The LDAA #\$FF instruction also clears the V-bit in the CPU12 Family's condition code register so the REV instruction knows it is processing antecedents. During rule list processing, the V bit is toggled each time an \$FE is detected in the list. The V bit indicates whether REV is processing antecedents or consequents.

Line 17 is the REV instruction, a self-contained loop to process successive elements in the rule list until an \$FF character is found. For a system of 17 rules with two antecedents and one consequent each, the REV instruction takes 259 cycles, but it is interruptible so it does not cause a long interrupt latency.

Lines 18 through 20 set up pointers and an iteration count for the WAV instruction.

Line 21 is the beginning of defuzzification. The WAV instruction calculates a sum-of-products and a sum-of-weights.

Line 22 completes defuzzification. The EDIV instruction performs a 32-bit by 16-bit divide on the intermediate results from WAV to get the weighted average.

Line 23 moves the EDIV result into the double accumulator.

Line 24 stores the low 8-bits of the defuzzification result.

This example inference program shows how easy it is to incorporate fuzzy logic into general applications using the CPU12 Family. Code space and execution time are no longer serious factors in the decision to use fuzzy logic. The next section begins a much more detailed look at the fuzzy logic instructions of the CPU12 Family.

9.4 MEM Instruction Details

This section provides a more detailed explanation of the membership function evaluation instruction (MEM), including details about abnormal special cases for improperly defined membership functions.

9.4.1 Membership Function Definitions

Figure 9-4 shows how a normal membership function is specified in the CPU12 Family. Typically, a software tool is used to input membership functions graphically, and the tool generates data structures for the target processor and software kernel. Alternatively, points and slopes for the membership functions can be determined and stored in memory with define-constant assembler directives.

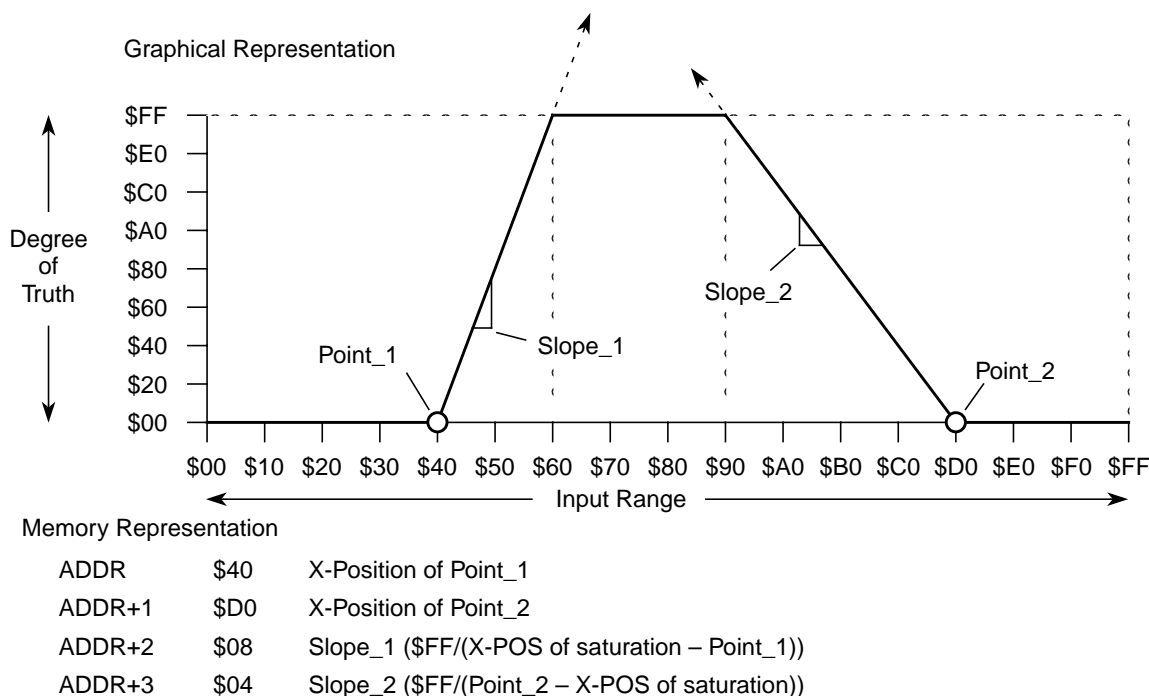


Figure 9-4. Defining a Normal Membership Function

An internal CPU algorithm calculates the y-value where the current input intersects a membership function. This algorithm assumes the membership function obeys some common-sense rules. If the membership function definition is improper, the results may be unusual. See [Section 9.4.2, “Abnormal Membership Function Definitions”](#) for a discussion of these cases.

These rules apply to normal membership functions.

- $\$00 \leq \text{Point_1} < \FF
- $\$00 < \text{Point_2} \leq \FF
- $\text{Point_1} < \text{Point_2}$
- The sloping sides of the trapezoid meet at or above \$FF.

Each system input such as temperature has several labels such as cold, cool, normal, warm, and hot. Each label of each system input must have a membership function to describe its meaning in an unambiguous numerical way. Typically, there are three to seven labels per system input, but there is no practical restriction on this number as far as the fuzzification step is concerned.

9.4.2 Abnormal Membership Function Definitions

In the CPU12 Family, it is possible (and proper) to define “crisp” membership functions. A crisp membership function has one or both sides vertical (infinite slope). Since the slope value \$00 is not used otherwise, it is assigned to mean infinite slope to the MEM instruction in the CPU12 Family.

Although a good fuzzy development tool will not allow the user to specify an improper membership function, it is possible to have program errors or memory errors which result in erroneous abnormal

membership functions. Although these abnormal shapes do not correspond to any working systems, understanding how these cases are treated in the CPU12 Family can be helpful for debugging.

A close examination of the MEM instruction algorithm will show how such membership functions are evaluated. **Figure 9-5** is a complete flow diagram for the execution of a MEM instruction. Each rectangular box represents one CPU bus cycle. The number in the upper left corner corresponds to the cycle number and the letter corresponds to the cycle type (refer to <st-blue>Chapter 6 Instruction Glossary for details). The upper portion of the box includes information about bus activity during this cycle (if any). The lower portion of the box, which is separated by a dashed line, includes information about internal CPU processes. It is common for several internal functions to take place during a single CPU cycle (for example, in cycle 2, two 8-bit subtractions take place and a flag is set based on the results).

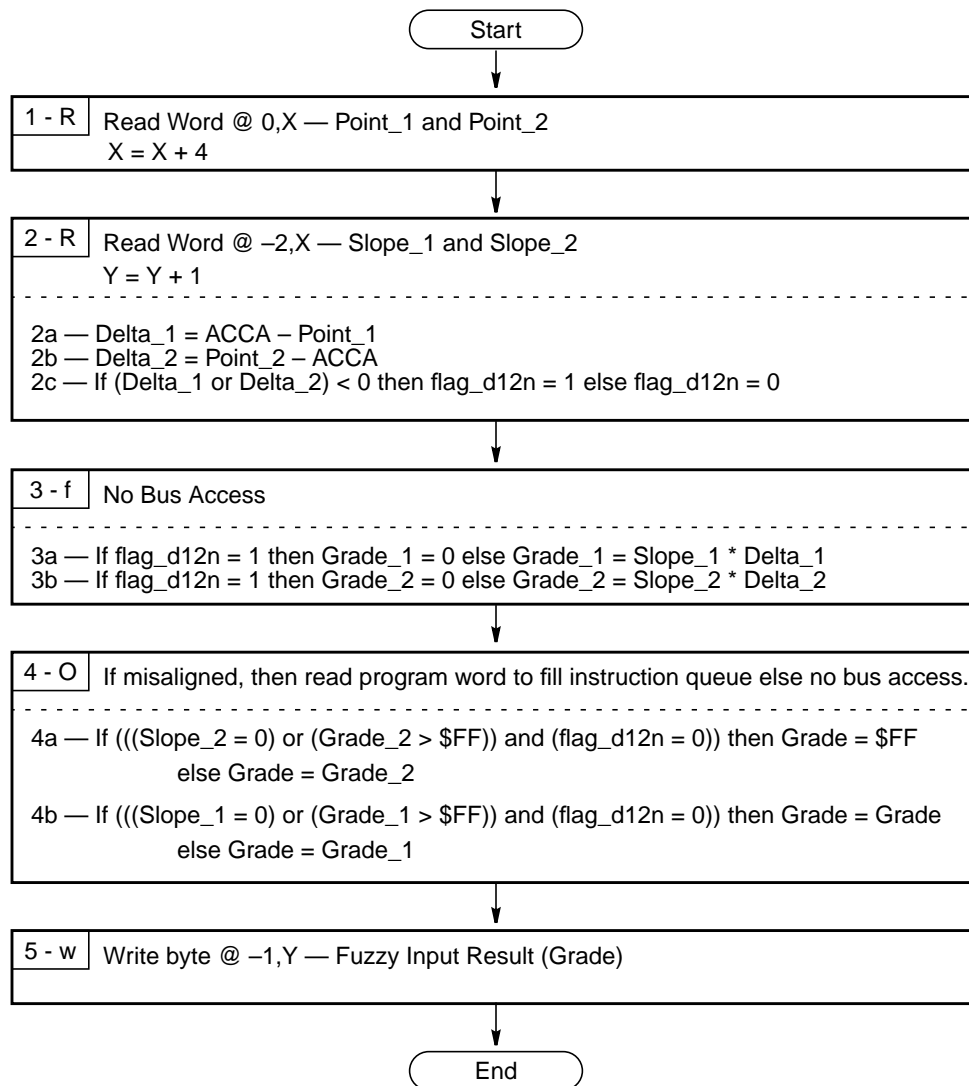


Figure 9-5. MEM Instruction Flow Diagram

Consider 4a: If $((\text{Slope_2} = 0) \text{ or } (\text{Grade_2} > \$FF)) \text{ and } (\text{flag_d12n} = 0)$.

The flag_d12n is zero as long as the input value (in accumulator A) is within the trapezoid. Everywhere outside the trapezoid, one or the other delta term will be negative, and the flag will equal one. Slope_2 equals zero indicates the right side of the trapezoid has infinite slope, so the resulting grade should be \$FF everywhere in the trapezoid, including at Point_2, as far as this side is concerned. The term Grade_2 greater than \$FF means the value is far enough into the trapezoid that the right sloping side of the trapezoid has crossed above the \$FF cutoff level and the resulting grade should be \$FF as far as the right sloping side is concerned. 4a decides if the value is left of the right sloping side (Grade = \$FF), or on the sloping portion of the right side of the trapezoid (Grade = Grade_2). 4b could still override this tentative value in grade.

In 4b, Slope_1 is zero if the left side of the trapezoid has infinite slope (vertical). If so, the result (grade) should be \$FF at and to the right of Point_1 everywhere within the trapezoid as far as the left side is concerned. The Grade_1 greater than \$FF term corresponds to the input being to the right of where the left sloping side passes the \$FF cutoff level. If either of these conditions is true, the result (grade) is left at the value it got from 4a. The “else” condition in 4b corresponds to the input falling on the sloping portion of the left side of the trapezoid (or possibly outside the trapezoid), so the result is grade equal Grade_1. If the input was outside the trapezoid, flag_d12n would be one and Grade_1 and Grade_2 would have been forced to \$00 in cycle 3. The else condition of 4b would set the result to \$00.

The special cases shown here represent abnormal membership function definitions. The explanations describe how the specific algorithm in the 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.

9.4.2.1 Abnormal Membership Function Case 1

This membership function is abnormal because the sloping sides cross below the \$FF cutoff level. The flag_d12n signal forces the membership function to evaluate to \$00 everywhere except from Point_1 to Point_2. Within this interval, the tentative values for Grade_1 and Grade_2 calculated in cycle 3 fall on the crossed sloping sides. In step 4a, grade gets set to the Grade_2 value, but in 4b this is overridden by the Grade_1 value, which ends up as the result of the MEM instruction. One way to say this is that the result follows the left sloping side until the input passes Point_2, where the result goes to \$00.

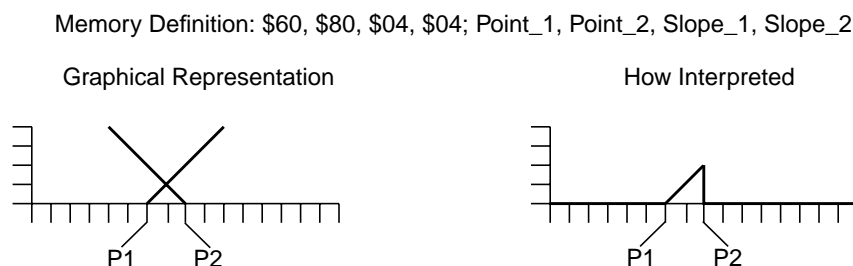


Figure 9-6. Abnormal Membership Function Case 1

If Point_1 was to the right of Point_2, flag_d12n would force the result to be \$00 for all input values. In fact, flag_d12n always limits the region of interest to the space greater than or equal to Point_1 and less than or equal to Point_2.

9.4.2.2 Abnormal Membership Function Case 2

Like the previous example, the membership function in case 2 is abnormal because the sloping sides cross below the \$FF cutoff level, but the left sloping side reaches the \$FF cutoff level before the input gets to Point_2. In this case, the result follows the left sloping side until it reaches the \$FF cutoff level. At this point, the $(\text{Grade}_1 > \$FF)$ term of 4b kicks in, making the expression true so grade equals grade (no overwrite). The result from here to Point_2 becomes controlled by the “else” part of 4a ($\text{grade} = \text{Grade}_2$), and the result follows the right sloping side.

Memory Definition: \$60, \$C0, \$04, \$04; Point_1, Point_2, Slope_1, Slope_2

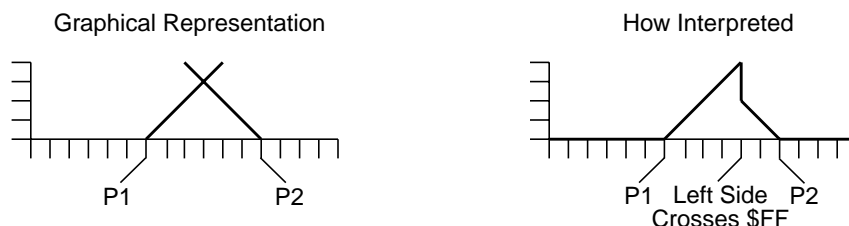


Figure 9-7. Abnormal Membership Function Case 2

9.4.2.3 Abnormal Membership Function Case 3

The membership function in case 3 is abnormal because the sloping sides cross below the \$FF cutoff level, and the left sloping side has infinite slope. In this case, 4a is not true, so grade equals Grade_2. 4b is true because Slope_1 is zero, so 4b does not overwrite grade.

Memory Definition: \$60, \$80, \$00, \$04; Point_1, Point_2, Slope_1, Slope_2

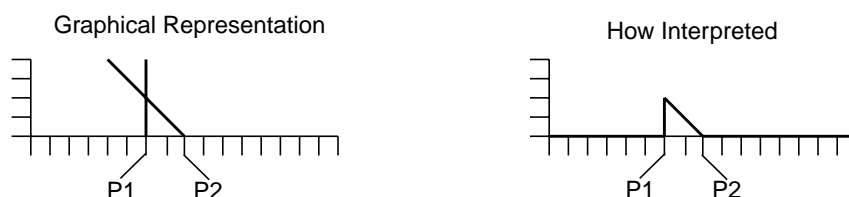


Figure 9-8. Abnormal Membership Function Case 3

9.5 REV and REVW Instruction Details

This section provides a more detailed explanation of the rule evaluation instructions (REV and REVW). The data structures used to specify rules are somewhat different for the weighted versus unweighted versions of the instruction. One uses 8-bit offsets in the encoded rules, while the other uses full 16-bit addresses. This affects the size of the rule data structure and execution time.

9.5.1 Unweighted Rule Evaluation (REV)

This instruction implements basic min-max rule evaluation. CPU registers are used for pointers and intermediate calculation results.

Since the REV instruction is essentially a list-processing instruction, execution time is dependent on the number of elements in the rule list. The REV instruction is interruptible (typically within three bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and REVW instructions.

9.5.1.1 Set Up Prior to Executing REV

Some CPU registers and memory locations need to be set up prior to executing the REV instruction. X and Y index registers are used as index pointers to the rule list and the fuzzy inputs and outputs. The A accumulator is used for intermediate calculation results and needs to be set to \$FF initially. The V condition code bit is used as an instruction status indicator to show whether antecedents or consequents are being processed. Initially, the V bit is cleared to zero to indicate antecedents are being processed. The fuzzy outputs (working RAM locations) need to be cleared to \$00. If these values are not initialized before executing the REV instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The REV instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the REV instruction finishes, X will point at the next address past the \$FF separator character that marks the end of the rule list.

The Y index register is set to the base address for the fuzzy inputs and outputs (in working RAM). Each rule antecedent is an unsigned 8-bit offset from this base address to the referenced fuzzy input. Each rule consequent is an unsigned 8-bit offset from this base address to the referenced fuzzy output. The Y index register remains constant throughout execution of the REV instruction.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REV instruction. During antecedent processing, A starts out at \$FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent (MIN). During consequent processing, A holds the truth value for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REV, A must be set to \$FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to \$FF when the instruction detects the \$FE marker character between the last consequent of the previous rule and the first antecedent of a new rule.

The instruction LDAA #\$FF clears the V bit at the same time it initializes A to \$FF. This satisfies the REV setup requirement to clear the V bit as well as the requirement to initialize A to \$FF. Once the REV instruction starts, the value in the V bit is automatically maintained as \$FE separator characters are detected.

The final requirement to clear all fuzzy outputs to \$00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value for the current rule.

If the current truth value is larger, it is written over the previous value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REV finishes, A will hold the truth value for the last rule in the rule list. The V condition code bit should be one because the last element before the \$FF end marker should have been a rule consequent. If V is zero after executing REV, it indicates the rule list was structured incorrectly.

9.5.1.2 Interrupt Details

The REV instruction includes a 3-cycle processing loop for each byte in the rule list (including antecedents, consequents, and special separator characters). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REV instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REV instruction, points to the REV instruction rather than the instruction that follows. This causes the CPU to try to execute a new REV instruction upon return from the interrupt. Since the CPU registers (including the V bit in the condition codes register) indicate the current status of the interrupted REV instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.5.1.3 Cycle-by-Cycle Details for REV

The central element of the REV instruction is a 3-cycle loop that is executed once for each byte in the rule list. There is a small amount of housekeeping activity to get this loop started as REV begins and a small sequence to end the instruction. If an interrupt comes, there is a special small sequence to save CPU status on the stack before honoring the requested interrupt.

Figure 9-9 is a REV instruction flow diagram. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to execution cycle codes (refer to [Chapter 6 Instruction Glossary](#) for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit or no data is transferred.

When a value is read from memory, it cannot be used by the CPU until the second cycle after the read takes place. This is due to access and propagation delays.

Since there is more than one flow path through the REV instruction, cycle numbers have a decimal place. This decimal place indicates which of several possible paths is being used. The CPU normally moves forward by one digit at a time within the same flow (flow number is indicated after the decimal point in the cycle number). There are two exceptions possible to this orderly sequence through an instruction. The first is a branch back to an earlier cycle number to form a loop as in 6.0 to 4.0. The second type of sequence change is from one flow to a parallel flow within the same instruction such as 4.0 to 5.2, which occurs if the REV instruction senses an interrupt. In this second type of sequence branch, the whole number advances by one and the flow number changes to a new value (the digit after the decimal point).

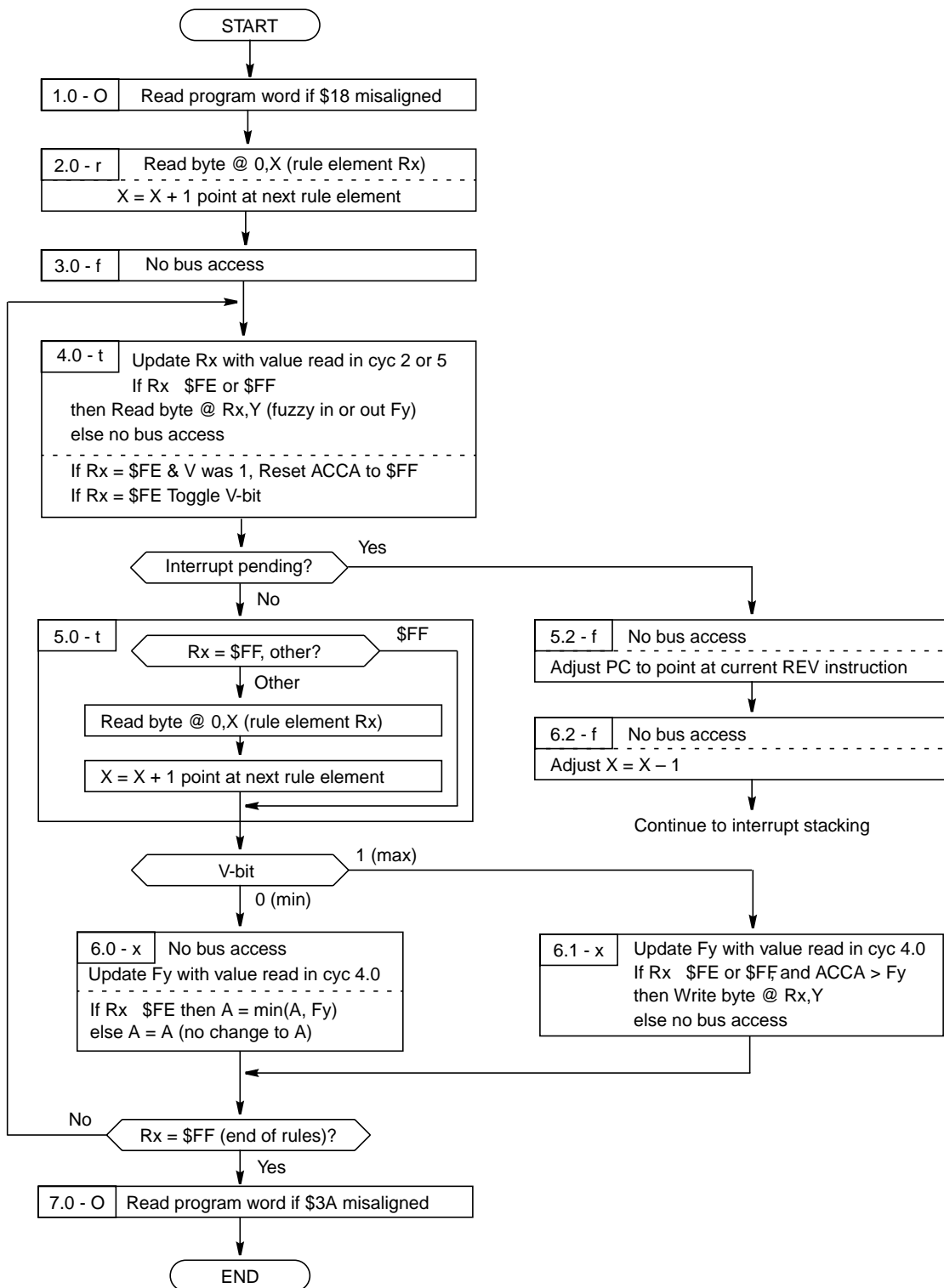


Figure 9-9. REV Instruction Flow Diagram

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 type cycle. One or the other of these will be a program word fetch, while the other will be a free cycle where the CPU does not access the bus. Although the \$18 page prebyte is a required part of the REV instruction, it is treated by the CPU as a somewhat separate single cycle instruction.

Rule evaluation begins at cycle 2.0 with a byte read of the first element in the rule list. Usually this would be the first antecedent of the first rule, but the REV instruction can be interrupted, so this could be a read of any byte in the rule list. The X index register is incremented so it points to the next element in the rule list. Cycle 3.0 is needed to satisfy the required delay between a read and when data is valid to the CPU. Some internal CPU housekeeping activity takes place during this cycle, but there is no bus activity. By cycle 4.0, the rule element that was read in cycle 2.0 is available to the CPU.

Cycle 4.0 is the first cycle of the main three cycle rule evaluation loop. Depending upon whether rule antecedents or consequents are being processed, the loop will consist of cycles 4.0, 5.0, 6.0, or the sequence 4.0, 5.0, 6.1. This loop is executed once for every byte in the rule list, including the \$FE separators and the \$FF end-of-rules marker.

At each cycle 4.0, a fuzzy input or fuzzy output is read, except during the loop passes associated with the \$FE and \$FF marker bytes, where no bus access takes place during cycle 4.0. The read access uses the Y index register as the base address and the previously read rule byte (R_x) as an unsigned offset from Y. The fuzzy input or output value read here will be used during the next cycle 6.0 or 6.1. Besides being used as the offset from Y for this read, the previously read R_x is checked to see if it is a separator character (\$FE). If R_x was \$FE and the V bit was one, this indicates a switch from processing consequents of one rule to starting to process antecedents of the next rule. At this transition, the A accumulator is initialized to \$FF to prepare for the min operation to find the smallest fuzzy input. Also, if R_x is \$FE, the V bit is toggled to indicate the change from antecedents to consequents, or consequents to antecedents.

During cycle 5.0, a new rule byte is read unless this is the last loop pass, and R_x is \$FF (marking the end of the rule list). This new rule byte will not be used until cycle 4.0 of the next pass through the loop.

Between cycle 5.0 and 6.x, the V-bit is used to decide which of two paths to take. If V is zero, antecedents are being processed and the CPU progresses to cycle 6.0. If V is one, consequents are being processed and the CPU goes to cycle 6.1.

During cycle 6.0, the current value in the A accumulator is compared to the fuzzy input that was read in the previous cycle 4.0, and the lower value is placed in the A accumulator (min operation). If R_x is \$FE, this is the transition between rule antecedents and rule consequents, and this min operation is skipped (although the cycle is still used). No bus access takes place during cycle 6.0 but cycle 6.x is considered an x type cycle because it could be a byte write (cycle 6.1) or a free cycle (cycle 6.0 or 6.1 with $R_x = \$FE$ or \$FF).

If an interrupt arrives while the REV instruction is executing, REV can break between cycles 4.0 and 5.0 in an orderly fashion so that the rule evaluation operation can resume after the interrupt has been serviced. Cycles 5.2 and 6.2 are needed to adjust the PC and X index register so the REV operation can recover after the interrupt. PC is adjusted backward in cycle 5.2 so it points to the currently running REV instruction. After the interrupt, rule evaluation will resume, but the values that were stored on the stack for index registers, accumulator A, and CCR will cause the operation to pick up where it left off. In cycle 6.2, the X

index register is adjusted backward by one because the last rule byte needs to be re-fetched when the REV instruction resumes.

After cycle 6.2, the REV instruction is finished, and execution would continue to the normal interrupt processing flow.

9.5.2 Weighted Rule Evaluation (RE VW)

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 RE VW instruction is essentially a list-processing instruction, execution time is dependent on the number of rules and the number of elements in the rule list. The RE VW instruction is interruptible (typically within three to five bus cycles), so it does not adversely affect worst case interrupt latency. Since all intermediate results and instruction status are held in stacked CPU registers, the interrupt service code can even include independent REV and RE VW instructions.

The rule structure is different for RE VW than for REV. For RE VW, the rule list is made up of 16-bit elements rather than 8-bit elements. Each antecedent is represented by the full 16-bit address of the corresponding fuzzy input. Each rule consequent is represented by the full address of the corresponding fuzzy output.

The markers separating antecedents from consequents are the reserved 16-bit value \$FFFE, and the end of the last rule is marked by the reserved 16-bit value \$FFFF. Since \$FFFE and \$FFFF correspond to the addresses of the reset vector, there would never be a fuzzy input or output at either of these locations.

9.5.2.1 Set Up Prior to Executing RE VW

Some CPU registers and memory locations need to be set up prior to executing the RE VW instruction. X and Y index registers are used as index pointers to the rule list and the list of rule weights. The A accumulator is used for intermediate calculation results and needs to be set to \$FF initially. The V condition code bit is used as an instruction status indicator that shows whether antecedents or consequents are being processed. Initially the V bit is cleared to zero to indicate antecedents are being processed. The C condition code bit is used to indicate whether rule weights are to be used (1) or not (0). The fuzzy outputs (working RAM locations) need to be cleared to \$00. If these values are not initialized before executing the RE VW instruction, results will be erroneous.

The X index register is set to the address of the first element in the rule list (in the knowledge base). The RE VW instruction automatically updates this pointer so that the instruction can resume correctly if it is interrupted. After the RE VW instruction finishes, X will point at the next address past the \$FFFF separator word that marks the end of the rule list.

The Y index register is set to the starting address of the list of rule weights. Each rule weight is an 8-bit value. The weighted result is the truncated upper eight bits of the 16-bit result, which is derived by multiplying the minimum rule antecedent value (\$00–\$FF) by the weight plus one (\$001–\$100). This

method of weighting rules allows an 8-bit weighting factor to represent a value between zero and one inclusive.

The 8-bit A accumulator is used to hold intermediate calculation results during execution of the REVW instruction. During antecedent processing, A starts out at \$FF and is replaced by any smaller fuzzy input that is referenced by a rule antecedent. If rule weights are enabled by the C condition code bit equal one, the rule truth value is multiplied by the rule weight just before consequent processing starts. During consequent processing, A holds the truth value (possibly weighted) for the rule. This truth value is stored to any fuzzy output that is referenced by a rule consequent, unless that fuzzy output is already larger (MAX).

Before starting to execute REVW, A must be set to \$FF (the largest 8-bit value) because rule evaluation always starts with processing of the antecedents of the first rule. For subsequent rules in the list, A is automatically set to \$FF when the instruction detects the \$FFFE marker word between the last consequent of the previous rule, and the first antecedent of a new rule.

Both the C and V condition code bits must be set up prior to starting a REVW instruction. Once the REVW instruction starts, the C bit remains constant and the value in the V bit is automatically maintained as \$FFFE separator words are detected.

The final requirement to clear all fuzzy outputs to \$00 is part of the MAX algorithm. Each time a rule consequent references a fuzzy output, that fuzzy output is compared to the truth value (weighted) for the current rule. If the current truth value is larger, it is written over the previous value in the fuzzy output. After all rules have been evaluated, the fuzzy output contains the truth value for the most-true rule that referenced that fuzzy output.

After REVW finishes, A will hold the truth value (weighted) for the last rule in the rule list. The V condition code bit should be one because the last element before the \$FFFF end marker should have been a rule consequent. If V is zero after executing REVW, it indicates the rule list was structured incorrectly.

9.5.2.2 Interrupt Details

The REVW instruction includes a 3-cycle processing loop for each word in the rule list (this loop expands to five cycles between antecedents and consequents to allow time for the multiplication with the rule weight). Within this loop, a check is performed to see if any qualified interrupt request is pending. If an interrupt is detected, the current CPU registers are stacked and the interrupt is honored. When the interrupt service routine finishes, an RTI instruction causes the CPU to recover its previous context from the stack, and the REVW instruction is resumed as if it had not been interrupted.

The stacked value of the program counter (PC), in case of an interrupted REVW instruction, points to the REVW instruction rather than the instruction that follows. This causes the CPU to try to execute a new REVW instruction upon return from the interrupt. Since the CPU registers (including the C bit and V bit in the condition codes register) indicate the current status of the interrupted REVW instruction, this effectively causes the rule evaluation operation to resume from where it left off.

9.5.2.3 Cycle-by-Cycle Details for REVW

The central element of the REVW instruction is a 3-cycle loop that is executed once for each word in the rule list. For the special case pass (where the \$FFFE separator word is read between the rule antecedents and the rule consequents, and weights are enabled by the C bit equal one), this loop takes five cycles. There is a small amount of housekeeping activity to get this loop started as REVW begins and a small sequence to end the instruction. If an interrupt comes, there is a special small sequence to save CPU status on the stack before the interrupt is serviced.

Figure 9-10 is a detailed flow diagram for the REVW instruction. Each rectangular box represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of each bold box correspond to the execution cycle codes (refer to Chapter 6 Instruction Glossary for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

In cycle 2.0, the first element of the rule list (a 16-bit address) is read from memory. Due to propagation delays, this value cannot be used for calculations until two cycles later (cycle 4.0). The X index register, which is used to access information from the rule list, is incremented by two to point at the next element of the rule list.

The operations performed in cycle 4.0 depend on the value of the word read from the rule list. \$FFFE is a special token that indicates a transition from antecedents to consequents or from consequents to antecedents of a new rule. The V bit can be used to decide which transition is taking place, and V is toggled each time the \$FFFE token is detected. If V was zero, a change from antecedents to consequents is taking place, and it is time to apply weighting (provided it is enabled by the C bit equal one). The address in TMP2 (derived from Y) is used to read the weight byte from memory. In this case, there is no bus access in cycle 5.0, but the index into the rule list is updated to point to the next rule element.

The old value of X (X_0) is temporarily held on internal nodes, so it can be used to access a rule word in cycle 7.2. The read of the rule word is timed to start two cycles before it will be used in cycle 4.0 of the next loop pass. The actual multiply takes place in cycles 6.2 through 8.2. The 8-bit weight from memory is incremented (possibly overflowing to \$100) before the multiply, and the upper eight bits of the 16-bit internal result is used as the weighted result. By using weight+1, the result can range from 0.0 times A to 1.0 times A. After 8.2, flow continues to the next loop pass at cycle 4.0.

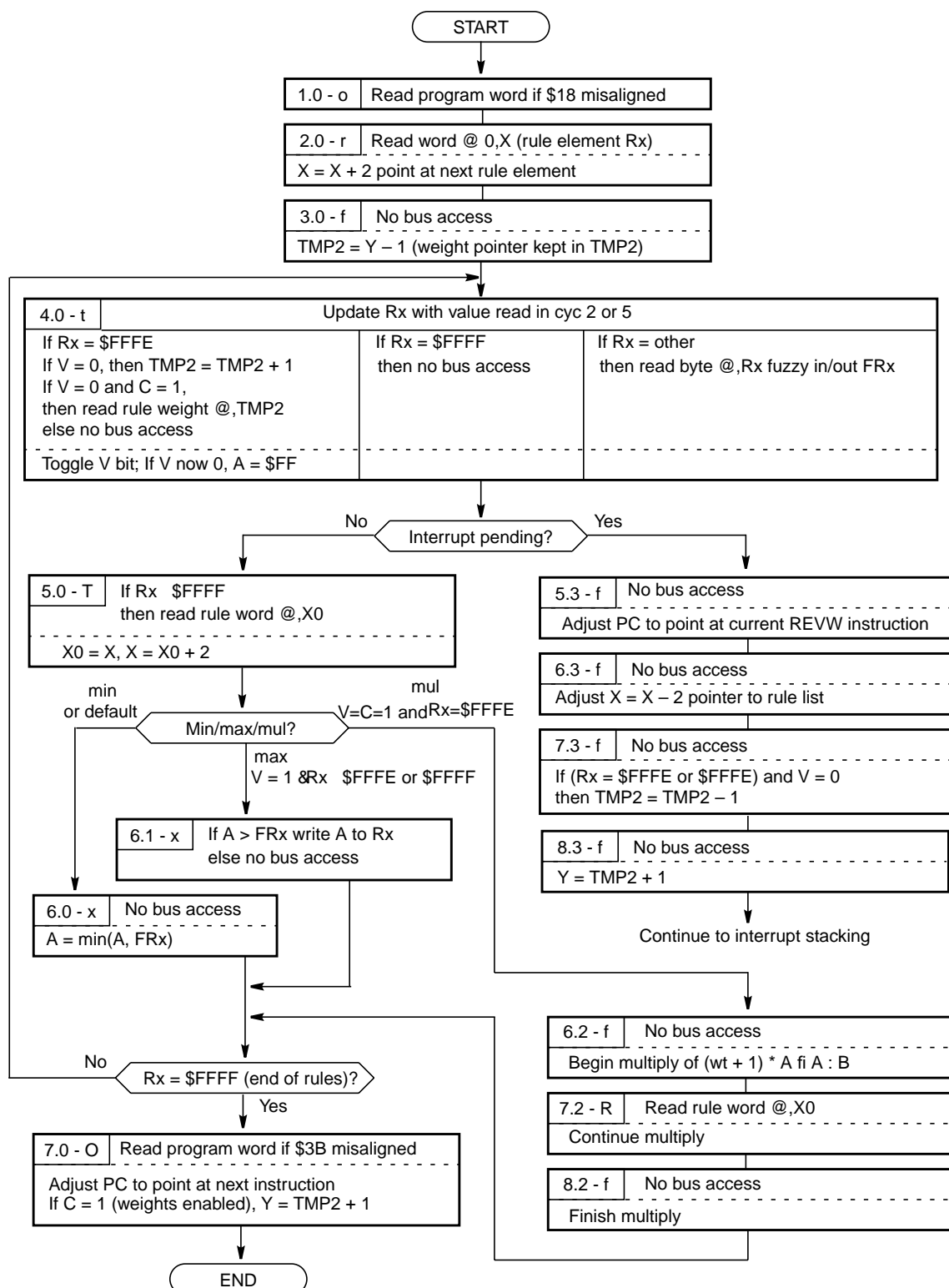


Figure 9-10. REVW Instruction Flow Diagram

At cycle 4.0, if R_x is \$FFFE and V was one, a change from consequents to antecedents of a new rule is taking place, so accumulator A must be reinitialized to \$FF. During processing of rule antecedents, A is updated with the smaller of A, or the current fuzzy input (cycle 6.0). Cycle 5.0 is usually used to read the next rule word and update the pointer in X. This read is skipped if the current R_x is \$FFFF (end of rules mark). If this is a weight multiply pass, the read is delayed until cycle 7.2. During processing of consequents, cycle 6.1 is used to optionally update a fuzzy output if the value in accumulator A is larger.

After all rules have been processed, cycle 7.0 is used to update the PC to point at the next instruction. If weights were enabled, Y is updated to point at the location that immediately follows the last rule weight.

9.6 WAV Instruction Details

The WAV instruction performs weighted average calculations used in defuzzification. The pseudo-instruction `wavr` is used to resume an interrupted weighted average operation. WAV calculates the numerator and denominator sums using:

$$\text{System Output} = \frac{\sum_{i=1}^n S_i F_i}{\sum_{i=1}^n F_i}$$

Where n is the number of labels of a system output, S_i are the singleton positions from the knowledge base, and F_i are fuzzy outputs from RAM. S_i and F_i are 8-bit values. The 8-bit B accumulator holds the iteration count n . Internal temporary registers hold intermediate sums, 24 bits for the numerator and 16 bits for the denominator. This makes this instruction suitable for n values up to 255 although eight is a more typical value. The final long division is performed with a separate EDIV instruction immediately after the WAV instruction. The WAV instruction returns the numerator and denominator sums in the correct registers for the EDIV. (EDIV performs the unsigned division $Y = Y : D / X$; remainder in D.)

Execution time for this instruction depends on the number of iterations (labels for the system output). WAV is interruptible so that worst case interrupt latency is not affected by the execution time for the complete weighted average operation. WAV includes initialization for the 24-bit and 16-bit partial sums so the first entry into WAV looks different than a resume from interrupt operation. The CPU handles this difficulty with a pseudo-instruction (`wavr`), which is specifically intended to resume an interrupted weighted average calculation. Refer to [Section 9.6.3, “Cycle-by-Cycle Details for WAV and wavr”](#) for more detail.

9.6.1 Set Up Prior to Executing WAV

Before executing the WAV instruction, index registers X and Y and accumulator B must be set up. Index register X is a pointer to the S_i singleton list. X must have the address of the first singleton value in the knowledge base. Index register Y is a pointer to the fuzzy outputs F_i . Y must have the address of the first fuzzy output for this system output. B is the iteration count n . The B accumulator must be set to the number of labels for this system output.

9.6.2 WAV Interrupt Details

The WAV instruction includes a 7-cycle processing loop for each label of the system output (8 cycles in M68HC12). Within this loop, the CPU checks whether a qualified interrupt request is pending. If an interrupt is detected, the current values of the internal temporary registers for the 24-bit and 16-bit sums are stacked, the CPU registers are stacked, and the interrupt is serviced.

A special processing sequence is executed when an interrupt is detected during a weighted average calculation. This exit sequence adjusts the PC so that it points to the second byte of the WAV object code (\$3C), before the PC is stacked. Upon return from the interrupt, the \$3C value is interpreted as a wavr pseudo-instruction. The wavr pseudo-instruction causes the CPU to execute a special WAV resumption sequence. The wavr recovery sequence adjusts the PC so that it looks like it did during execution of the original WAV instruction, then jumps back into the WAV processing loop. If another interrupt occurs before the weighted average calculation finishes, the PC is adjusted again as it was for the first interrupt. WAV can be interrupted any number of times, and additional WAV instructions can be executed while a WAV instruction is interrupted.

9.6.3 Cycle-by-Cycle Details for WAV and wavr

The WAV instruction is unusual in that the logic flow has two separate entry points. The first entry point is the normal start of a WAV instruction. The second entry point is used to resume the weighted average operation after a WAV instruction has been interrupted. This recovery operation is called the wavr pseudo-instruction.

Figure 9-11 is a flow diagram of the WAV instruction in the CPU, including the wavr pseudo-instruction. Each rectangular box in these figures represents one CPU clock cycle. Decision blocks and connecting arrows are considered to take no time at all. The letters in the small rectangles in the upper left corner of the boxes correspond to execution cycle codes (refer to Chapter 6, “Instruction Glossary” for details). Lower case letters indicate a cycle where 8-bit or no data is transferred. Upper case letters indicate cycles where 16-bit data could be transferred.

The cycle-by-cycle description provided here refers to the CPU flow in Figure 9-11. In terms of cycle-by-cycle bus activity, the \$18 page select prebyte is treated as a special 1-byte instruction. In cycle 1.0 of the WAV instruction, one word of program information will be fetched into the instruction queue if the \$18 is located at an odd address. If the \$18 is at an even address, the instruction queue cannot advance so there is no bus access in this cycle.

In cycle 2.0, three internal 16-bit temporary registers are cleared in preparation for summation operations, but there is no bus access. The WAV instruction maintains a 32-bit sum-of-products in TMP1 : TMP2 and a 16-bit sum-of-weights in TMP3. By keeping these sums inside the CPU, bus accesses are reduced and the WAV operation is optimized for high speed.

Cycles 3.0 through 9.0 form the 7-cycle main loop for WAV. The value in the 8-bit B accumulator is used to count the number of loop iterations. B is decremented at the top of the loop in cycle 3.0, and the test for zero is located at the bottom of the loop after cycle 9.0. Cycle 4.0 and 5.0 are used to fetch the 8-bit operands for one iteration of the loop. X and Y index registers are used to access these operands. The index registers are incremented as the operands are fetched. Cycle 6.0 is used to accumulate the current fuzzy output into TMP3. Cycles 7.0 through 9.0 are used to perform the eight by eight multiply of F_i times S_i ,

and accumulate this result into TMP1 : TMP2. Even though the sum-of-products will not exceed 24 bits, the sum is maintained in the 32-bit combined TMP1 : TMP2 register because it is easier to use existing 16-bit operations than it would be to create a new smaller operation to handle the high order bits of this sum.

Since the weighted average operation could be quite long, it is made to be interruptible. The usual longest latency path is from very early in cycle 6.0, through cycle 9.0, to the top of the loop to cycle 3.0, through cycle 5.0 to the interrupt check.

If the WAV instruction is interrupted, the internal temporary registers TMP3, TMP2, and TMP1 need to be stored on the stack so the operation can be resumed. Since the WAV instruction included initialization in cycle 2.0, the recovery path after an interrupt needs to be different. The wavr pseudo-instruction has the same opcode as WAV, but it is on the first page of the opcode map so there is no page prebyte (\$18) like there is for WAV. When WAV is interrupted, the PC is adjusted to point at the second byte of the WAV object code, so that it will be interpreted as the wavr pseudo-instruction on return from the interrupt, rather than the WAV instruction. During the recovery sequence, the PC is readjusted in case another interrupt comes before the weighted average operation finishes.

The resume sequence includes recovery of the temporary registers from the stack (1.1 through 3.1), and reads to get the operands for the current iteration. The normal WAV flow is then rejoined at cycle 6.0.

Upon normal completion of the instruction (cycle 10.0), the PC is adjusted so it points to the next instruction. The results are transferred from the TMP registers into CPU registers in such a way that the EDIV instruction can be used to divide the sum-of-products by the sum-of-weights. TMP1 : TMP2 is transferred into Y : D and TMP3 is transferred into X.

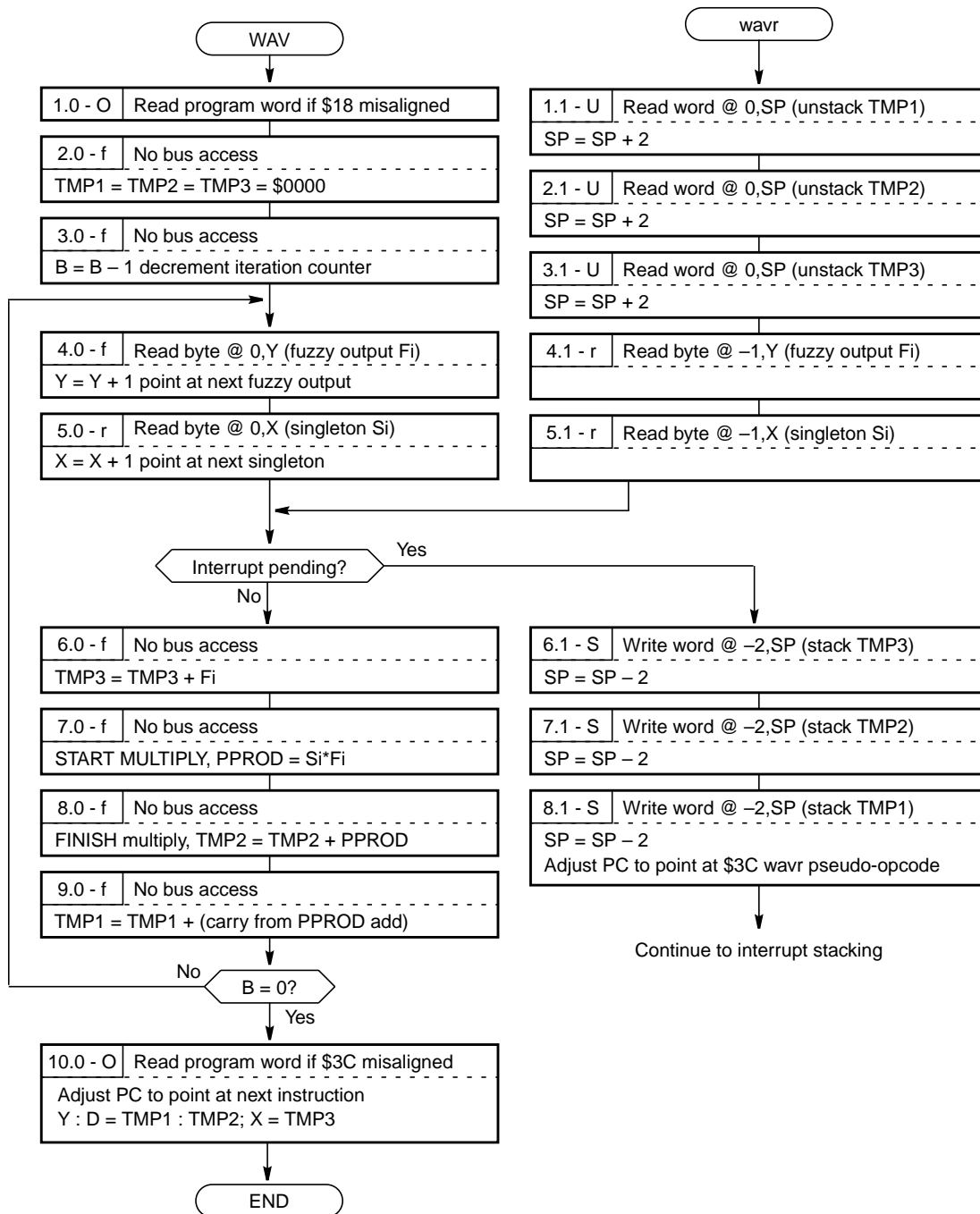


Figure 9-11. WAV and wavr Instruction Flow Diagram

9.7 Custom Fuzzy Logic Programming

The basic fuzzy logic inference techniques described earlier are suitable for a broad range of applications, but some systems may require customization. The built-in fuzzy instructions use 8-bit resolution and some systems may require finer resolution. The rule evaluation instructions only support variations of

MIN-MAX rule evaluation and other methods have been discussed in fuzzy logic literature. The weighted average of singletons is not the only defuzzification technique. The CPU12 Family has several instructions and addressing modes that can be helpful when developing custom fuzzy logic systems.

9.7.1 Fuzzification Variations

The MEM instruction supports trapezoidal membership functions and several other varieties, including membership functions with vertical sides (infinite slope sides). Triangular membership functions are a subset of trapezoidal functions. Some practitioners refer to s-, z-, and π -shaped membership functions. These refer to a trapezoid butted against the right end of the x-axis, a trapezoid butted against the left end of the x-axis, and a trapezoidal membership function that isn't butted against either end of the x-axis, respectively. Many other membership function shapes are possible, if memory space and processing bandwidth are sufficient.

Tabular membership functions offer complete flexibility in shape and very fast evaluation time. However, tables take a very large amount of memory space (as many as 256 bytes per label of one system input). The excessive size to specify tabular membership functions makes them impractical for most microcontroller-based fuzzy systems. The CPU12 Family instruction set includes two instructions (TBL and ETBL) for lookup and interpolation of compressed tables.

The TBL instruction uses 8-bit table entries (y-values) and returns an 8-bit result. The ETBL instruction uses 16-bit table entries (y-values) and returns a 16-bit result. A flexible indexed addressing mode is used to identify the effective address of the data point at the beginning of the line segment, and the data value for the end point of the line segment is the next consecutive memory location (byte for TBL and word for ETBL). In both cases, the B accumulator represents the ratio of (the x-distance from the beginning of the line segment to the lookup point) to (the x-distance from the beginning of the line segment to the end of the line segment). B is treated as an 8-bit binary fraction with radix point left of the MSB, so each line segment can effectively be divided into 256 pieces. During execution of the TBL or ETBL instruction, the difference between the end point y-value and the beginning point y-value (a signed byte-TBL or word-ETBL) is multiplied by the B accumulator to get an intermediate delta-y term. The result is the y-value of the beginning point, plus this signed intermediate delta-y value.

Because indexed addressing mode is used to identify the starting point of the line segment of interest, there is a great deal of flexibility in constructing tables. A common method is to break the x-axis range into 256 equal width segments and store the y value for each of the resulting 257 endpoints. The 16-bit D accumulator is then used as the x input to the table. The upper eight bits (A) is used as a coarse lookup to find the line segment of interest, and the lower eight bits (B) is used to interpolate within this line segment.

In the program sequence

```
LDX      #TBL_START
LDD      DATA_IN
TBL      A, X
```

The notation A,X causes the TBL instruction to use the Ath line segment in the table. The low-order half of D (B) is used by TBL to calculate the exact data value from this line segment. This type of table uses only 257 entries to approximate a table with 16 bits of resolution. This type of table has the disadvantage of equal width line segments, which means just as many points are needed to describe a flat portion of the desired function as are needed for the most active portions.

Another type of table stores x:y coordinate pairs for the endpoints of each linear segment. This type of table may reduce the table storage space compared to the previous fixed-width segments because flat areas of the functions can be specified with a single pair of endpoints. This type of table is a little harder to use with the CPU12 Family TBL and ETBL instructions because the table instructions expect y-values for segment endpoints to be in consecutive memory locations.

Consider a table made up of an arbitrary number of x:y coordinate pairs, where all values are eight bits. The table is entered with the x-coordinate of the desired point to lookup in the A accumulator. When the table is exited, the corresponding y-value is in the A accumulator. Figure 9-12 shows one way to work with this type of table.

BEGIN	LDY	#TABLE_START-2	;setup initial table pointer
FIND_LOOP	CMPA	2,+Y	;find first Xn > XL
			; (auto pre-inc Y by 2)
	BLS	FIND_LOOP	;loop if XL .le. Xn
* on fall thru,	XB@-2,Y	YB@-1,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:X
	SUBB	-2,Y	;D = 0:(X-XB)
	EXG	D,X	;X = (X-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)/(X-XB)
	EXG	D,X	;move fractional result to A:B
	EXG	A,B	;byte swap - need result in B
	TSTA		;check for rounding
	BPL	NO_ROUND	
	INCB		;round B up by 1
NO_ROUND	LDAA	1,Y	;YE
	PSHA		;put on stack for TBL later
	LDAA	-1,Y	;YB
	PSHA		;now YB@0,SP and YE@1,SP
	TBL	2,SP+	;interpolate and deallocate
			;stack temps

Figure 9-12. Endpoint Table Handling

The basic idea is to find the segment of interest, temporarily build a 1-segment table of the correct format on the stack, then use TBL with stack relative indexed addressing to interpolate. The most difficult part of the routine is calculating the proportional distance from the beginning of the segment to the lookup point versus the width of the segment $((XL-XB)/(XE-XB))$. With this type of table, this calculation must be done at run time. In the previous type of table, this proportional term is an inherent part (the lowest order bits) of the data input to the table.

Some fuzzy theorists have suggested membership functions should be shaped like normal distribution curves or other mathematical functions. This may be correct, but the processing requirements to solve for an intercept on such a function would be unacceptable for most microcontroller-based fuzzy systems. Such a function could be encoded into a table of one of the previously described types.

For many common systems, the thing that is most important about membership function shape is that there is a gradual transition from non-membership to membership as the system input value approaches the central range of the membership function.

Examine the human problem of stopping a car at an intersection. Rules such as “If intersection is close and speed is fast, apply brakes” might be used. The meaning (reflected in membership function shape and position) of the labels “close” and “fast” will be different for a teenager than they are for a grandmother, but both can accomplish the goal of stopping. It makes intuitive sense that the exact shape of a membership function is much less important than the fact that it has gradual boundaries.

9.7.2 Rule Evaluation Variations

The REV and REVW instructions expect fuzzy input and fuzzy output values to be 8-bit values. In a custom fuzzy inference program, higher resolution may be desirable (although this is not a common requirement). The CPU12 Family includes variations of minimum and maximum operations that work with the fuzzy MIN-MAX inference algorithm. The problem with the fuzzy inference algorithm is that the min and max operations need to store their results differently, so the min and max instructions must work differently or more than one variation of these instructions is needed.

The CPU12 Family has MIN and MAX instructions for 8- or 16-bit operands, where one operand is in an accumulator and the other is a referenced memory location. There are separate variations that replace the accumulator or the memory location with the result. While processing rule antecedents in a fuzzy inference program, a reference value must be compared to each of the referenced fuzzy inputs, and the smallest input must end up in an accumulator. The instruction

```
EMIND          2,X+          ;process one rule antecedent
```

automates the central operations needed to process rule antecedents. The E stands for extended, so this instruction compares 16-bit operands. The D at the end of the mnemonic stands for the D accumulator, which is both the first operand for the comparison and the destination of the result. The 2,X+ is an indexed addressing specification that says X points to the second operand for the comparison and it will be post-incremented by 2 to point at the next rule antecedent.

When processing rule consequents, the operand in the accumulator must remain constant (in case there is more than one consequent in the rule), and the result of the comparison must replace the referenced fuzzy output in RAM. To do this, use the instruction

```
EMAXM          2,X+          ;process one rule consequent
```

The M at the end of the mnemonic indicates that the result will replace the referenced memory operand. Again, indexed addressing is used. These two instructions would form the working part of a 16-bit resolution fuzzy inference routine.

There are many other methods of performing inference, but none of these are as widely used as the min-max method. Since the CPU12 Family is a general-purpose microcontroller family, the programmer has complete freedom to program any algorithm desired. A custom programmed algorithm would typically take more code space and execution time than a routine that used the built-in REV or REVW instructions.

9.7.3 Defuzzification Variations

Other CPU12 Family instructions can help with custom defuzzification routines in two main areas:

- The first case is working with operands that are more than eight bits.
- The second case involves using an entirely different approach than weighted average of singletons.

The primary part of the WAV instruction is a multiply and accumulate operation to get the numerator for the weighted average calculation. When working with operands as large as 16 bits, the EMACS instruction could at least be used to automate the multiply and accumulate function. The CPU12 Family has extended math capabilities, including the EMACS instruction which uses 16-bit input operands and accumulates the sum to a 32-bit memory location and 32-bit by 16-bit divide instructions.

One benefit of the WAV instruction is that both a sum of products and a sum of weights are maintained, while the fuzzy output operand is only accessed from memory once. Since memory access time is such a significant part of execution time, this provides a speed advantage compared to conventional instructions.

The weighted average of singletons is the most commonly used technique in microcontrollers because it is computationally less difficult than most other methods. The simplest method is called max defuzzification, which simply uses the largest fuzzy output as the system result. However, this approach does not take into account any other fuzzy outputs, even when they are almost as true as the chosen max output. Max defuzzification is not a good general choice because it only works for a subset of fuzzy logic applications.

The CPU12 Family is well suited for more computationally challenging algorithms than weighted average. A 32-bit by 16-bit divide instruction takes 11 or 12 25-MHz cycles for unsigned or signed variations. A 16-bit by 16-bit multiply with a 32-bit result takes only three 25-MHz cycles. The EMACS instruction uses 16-bit operands and accumulates the result in a 32-bit memory location, taking only 12 25-MHz cycles per iteration, including accessing all operands from memory and storing the result to memory.

Appendix A

Instruction Reference

A.1 Introduction

This appendix provides quick references for the instruction set, opcode map, and encoding.

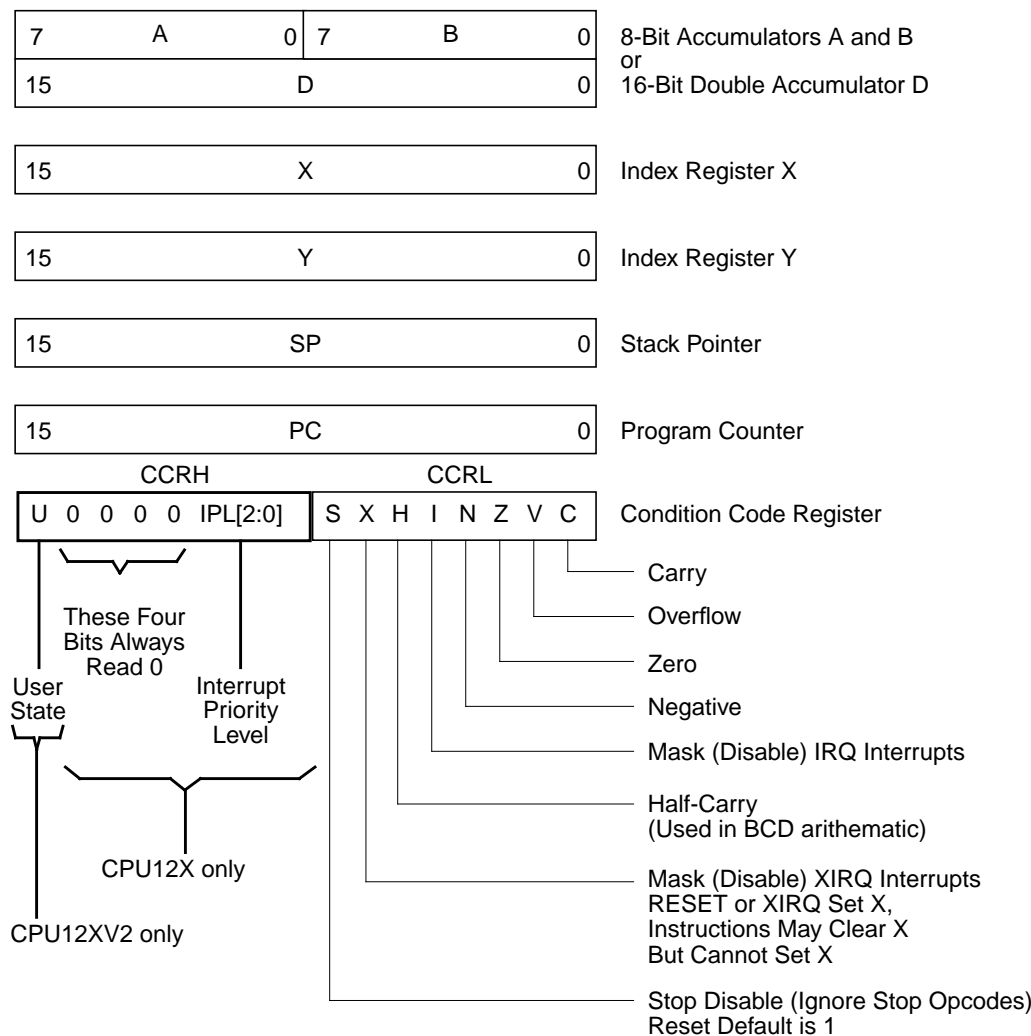
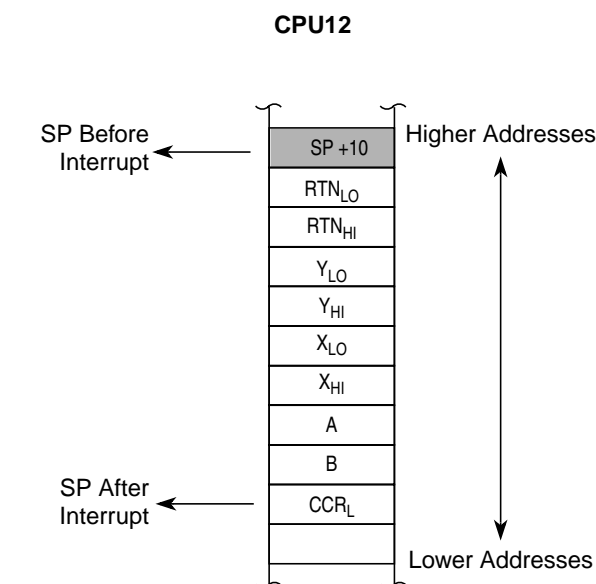


Figure A-1. Programming Model

A.2 Stack and Memory Layout

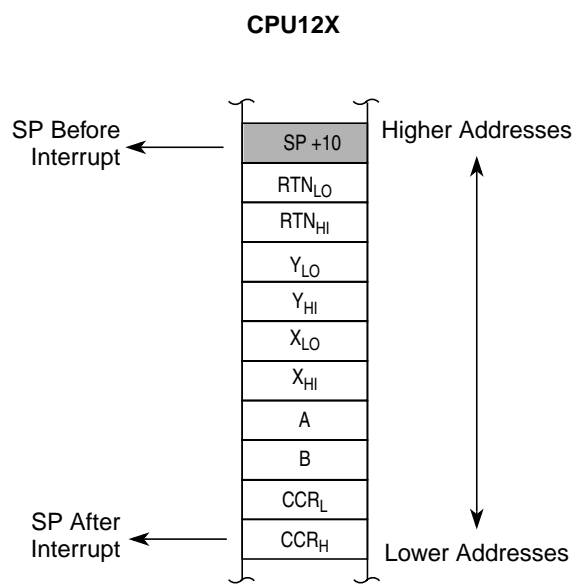


STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS ODD BEFORE INTERRUPT

SP +8	RTN _{LO}		SP +9
SP +6	Y _{LO}	RTN _{HI}	SP +7
SP +4	X _{LO}	Y _{HI}	SP +5
SP +2	A	X _{HI}	SP +3
SP	CCR	B	SP +1
SP -2			SP -1

STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS EVEN BEFORE INTERRUPT

SP +9			SP +10
SP +7	RTN _{HI}	RTN _{LO}	SP +8
SP +5	Y _{HI}	Y _{LO}	SP +6
SP +4	X _{HI}	X _{LO}	SP +4
SP +1	B	A	SP +2
SP -1		CCR	SP



STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS ODD BEFORE INTERRUPT

SP +9	RTN _{LO}		SP +10
SP +7	Y _{LO}	RTN _{HI}	SP +8
SP +5	X _{LO}	Y _{HI}	SP +6
SP +3	A	X _{HI}	SP +4
SP +1	CCR _L	B	SP +2
SP -1		CCR _H	SP

STACK UPON ENTRY TO SERVICE ROUTINE
IF SP WAS EVEN BEFORE INTERRUPT

SP +10			SP +11
SP +8	RTN _{HI}	RTN _{LO}	SP +9
SP +6	Y _{HI}	Y _{LO}	SP +7
SP +4	X _{HI}	X _{LO}	SP +5
SP +2	B	A	SP +3
SP	CCR _H	CCR _L	SP +1

A.3 Interrupt Vector Locations

\$FFFE, \$FFFF	Power-On (POR) or External Reset
\$FFFC, \$FFFD	Clock Monitor Reset
\$FFFA, \$FFFB	Computer Operating Properly (COP Watchdog Reset)
\$FFF8, \$FFF9	Unimplemented Opcode Trap
\$FFF6, \$FFF7	Software Interrupt Instruction (SWI)
\$FFF4, \$FFF5	XIRQ
\$FFF2, \$FFF3	IRQ
\$FF00–\$FFF1	Device-Specific Interrupt Sources

A.4 Notation Used in Instruction Set Summary

CPU12 Family Register Notation

Accumulator A — A or a	Index Register Y — Y or y
Accumulator B — B or b	Stack Pointer — SP, sp, or s
Accumulator D — D or d	Program Counter — PC, pc, or p
Index Register X — X or x	Condition Code Register — CCR or c

Explanation of Italic Expressions in Source Form Column

<i>abc</i>	— A or B or CCR												
<i>abcdxys</i>	— A or B or CCR or D or X or Y or SP. Some assemblers also allow T2 or T3.												
<i>abd</i>	— A or B or D												
<i>abdxys</i>	— A or B or D or X or Y or SP												
<i>dxys</i>	— D or X or Y or SP												
<i>msk8</i>	— 8-bit mask, some assemblers require # symbol before value												
<i>opr8i</i>	— 8-bit immediate value												
<i>opr16i</i>	— 16-bit immediate value												
<i>opr8a</i>	— 8-bit address used with direct address mode												
<i>opr16a</i>	— 16-bit address value												
<i>opr0_xys</i>	— Indexed addressing postbyte code: <table> <tr> <td><i>opr3,−xys</i></td><td>Predecrement X or Y or SP by 1 . . . 8</td></tr> <tr> <td><i>opr3,+xys</i></td><td>Preincrement X or Y or SP by 1 . . . 8</td></tr> <tr> <td><i>opr3,xys−</i></td><td>Postdecrement X or Y or SP by 1 . . . 8</td></tr> <tr> <td><i>opr3,xys+</i></td><td>Postincrement X or Y or SP by 1 . . . 8</td></tr> <tr> <td><i>opr5,xysp</i></td><td>5-bit constant offset from X or Y or SP or PC</td></tr> <tr> <td><i>abd,xysp</i></td><td>Accumulator A or B or D offset from X or Y or SP or PC</td></tr> </table>	<i>opr3,−xys</i>	Predecrement X or Y or SP by 1 . . . 8	<i>opr3,+xys</i>	Preincrement X or Y or SP by 1 . . . 8	<i>opr3,xys−</i>	Postdecrement X or Y or SP by 1 . . . 8	<i>opr3,xys+</i>	Postincrement X or Y or SP by 1 . . . 8	<i>opr5,xysp</i>	5-bit constant offset from X or Y or SP or PC	<i>abd,xysp</i>	Accumulator A or B or D offset from X or Y or SP or PC
<i>opr3,−xys</i>	Predecrement X or Y or SP by 1 . . . 8												
<i>opr3,+xys</i>	Preincrement X or Y or SP by 1 . . . 8												
<i>opr3,xys−</i>	Postdecrement X or Y or SP by 1 . . . 8												
<i>opr3,xys+</i>	Postincrement X or Y or SP by 1 . . . 8												
<i>opr5,xysp</i>	5-bit constant offset from X or Y or SP or PC												
<i>abd,xysp</i>	Accumulator A or B or D offset from X or Y or SP or PC												
<i>opr3</i>	— Any positive integer 1 . . . 8 for pre/post increment/decrement												
<i>opr5</i>	— Any integer in the range −16 . . . +15												
<i>opr9</i>	— Any integer in the range −256 . . . +255												
<i>opr16</i>	— Any integer in the range −32,768 . . . 65,535												
<i>page</i>	— 8-bit value for PPAGE, some assemblers require # symbol before this value												

- rel8 — Label of branch destination within –128 to +127 locations
- rel9 — Label of branch destination within –256 to +255 locations
- rel16 — Any label within 64K memory space
- trapnum — Any 8-bit integer in the range \$30–\$39 or \$40–\$FF
- xys — X or Y or SP
- xysp — X or Y or SP or PC

Operators

- + — Addition
- — Subtraction
- — Logical AND
- | — Logical OR (inclusive)
- ⊕ — Logical exclusive OR
- × — Multiplication
- ÷ — Division
- \overline{M} — Negation. One's complement (invert each bit of M)
- :
- Example: A : B means the 16-bit value formed by concatenating 8-bit accumulator A with 8-bit accumulator B.
A is in the high-order position.
- ⇒ — Transfer
- Example: (A) ⇒ M means the content of accumulator A is transferred to memory location M.
- ↔ — Exchange
- Example: D ↔ X means exchange the contents of D with those of X.

Address Mode Notation

- INH — Inherent; no operands in object code
- IMM — Immediate; operand in object code
- DIR — Direct; operand is the lower byte of an address from \$0000 to \$00FF
- EXT — Operand is a 16-bit address
- REL — Two's complement relative offset; for branch instructions
- IDX — Indexed (no extension bytes); includes:
 - 5-bit constant offset from X, Y, SP, or PC
 - Pre/post increment/decrement by 1 . . . 8
 - Accumulator A, B, or D offset
- IDX1 — 9-bit signed offset from X, Y, SP, or PC; 1 extension byte
- IDX2 — 16-bit signed offset from X, Y, SP, or PC; 2 extension bytes
- [IDX2] — Indexed-indirect; 16-bit offset from X, Y, SP, or PC
- [D, IDX] — Indexed-indirect; accumulator D offset from X, Y, SP, or PC

Machine Coding

- dd — 8-bit direct address \$0000 to \$00FF. (High byte assumed to be \$00).
- ee — High-order byte of a 16-bit constant offset for indexed addressing.
- eb — Exchange/Transfer post-byte. See [Table A-5](#).
- ff — Low-order eight bits of a 9-bit signed constant offset for indexed addressing, or low-order byte of a 16-bit constant offset for indexed addressing.
- hh — High-order byte of a 16-bit extended address.
- ii — 8-bit immediate data value.
- jj — High-order byte of a 16-bit immediate data value.
- kk — Low-order byte of a 16-bit immediate data value.
- lb — Loop primitive (DBNE) post-byte. See [Table A-6](#).
- ll — Low-order byte of a 16-bit extended address.
- mm — 8-bit immediate mask value for bit manipulation instructions.
Set bits indicate bits to be affected.
- pg — Program page (bank) number used in CALL instruction.
- qq — High-order byte of a 16-bit relative offset for long branches.
- tn — Trap number \$30–\$39 or \$40–\$FF.
- rr — Signed relative offset \$80 (–128) to \$7F (+127).
Offset relative to the byte following the relative offset byte, or low-order byte of a 16-bit relative offset for long branches.
- xb — Indexed addressing post-byte. See [Table A-3](#) and [Table A-4](#).

Access Detail

Each code letter except (,), and comma equals one CPU cycle. Uppercase = 16-bit operation and lowercase = 8-bit operation. For complex sequences refer to [Chapter 5, “Instruction Set Overview”](#).

- f — Free cycle, CPU doesn’t use bus
- g — Read PPAGE internally
- I — Read indirect pointer (indexed indirect)
- i — Read indirect PPAGE value (CALL indirect only)
- n — Write PPAGE internally
- NA — Not available
- O — Optional program word fetch (P) if instruction is misaligned and has an odd number of bytes of object code — otherwise, appears as a free cycle (f); Page 2 prebyte treated as a separate 1-byte instruction
- P — Program word fetch (always an aligned-word read)
- r — 8-bit data read
- R — 16-bit data read

- s — 8-bit stack write
- S — 16-bit stack write
- w — 8-bit data write
- W — 16-bit data write
- u — 8-bit stack read
- U — 16-bit stack read
- V — 16-bit vector fetch (always an aligned-word read)
- t — 8-bit conditional read (or free cycle)
- T — 16-bit conditional read (or free cycle)
- x — 8-bit conditional write (or free cycle)
- () — Indicate a microcode loop
- , — Indicates where an interrupt could be honored

Special Cases

- PPP/P — Short branch, PPP if branch taken, P if not
- OPPP/OPO — Long branch, OPPP if branch taken, OPO if not

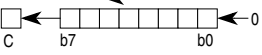
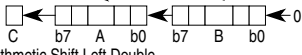
Condition Codes Columns

- — Status bit not affected by operation.
- 0 — Status bit cleared by operation.
- 1 — Status bit set by operation.
- Δ — Status bit affected by operation.
- ⇓ — Status bit may be cleared or remain set, but is not set by operation.
- ⇑ — Status bit may be set or remain cleared, but is not cleared by operation.
- ? — Status bit may be changed by operation but the final state is not defined.
- ! — Status bit used for a special purpose.

Table A-1. Instruction Set Summary (Sheet 1 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
ABA	(A) + (B) ⇒ A Add Accumulators A and B	INH	18 06	OO	--Δ-	Δ Δ Δ Δ
ABX*	(B) + (X) ⇒ X <i>Translates to LEAX B,X</i>	IDX	1A E5	P Pf	----	----
ABY*	(B) + (Y) ⇒ Y <i>Translates to LEAY B,Y</i>	IDX	19 ED	P Pf	----	----
* The cycle timing of CPU12V1 has been improved by removing one free cycle compared to CPU12V0 : P						
ADCA #opr8i ADCA opr8a ADCA opr16a ADCA oprx0_xysp ADCA oprx9_xysp ADCA oprx16_xysp ADCA [D,xysp] ADCA [oprx16,xysp]	(A) + (M) + C ⇒ A Add with Carry to A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	89 ii 99 dd B9 hh ll A9 xb A9 xb ff A9 xb ee ff A9 xb A9 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	--Δ-	Δ Δ Δ Δ
ADCB #opr8i ADCB opr8a ADCB opr16a ADCB oprx0_xysp ADCB oprx9_xysp ADCB oprx16_xysp ADCB [D,xysp] ADCB [oprx16,xysp]	(B) + (M) + C ⇒ B Add with Carry to B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C9 ii D9 dd F9 hh ll E9 xb E9 xb ff E9 xb ee ff E9 xb E9 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	--Δ-	Δ Δ Δ Δ
ADDA #opr8i ADDA opr8a ADDA opr16a ADDA oprx0_xysp ADDA oprx9_xysp ADDA oprx16_xysp ADDA [D,xysp] ADDA [oprx16,xysp]	(A) + (M) ⇒ A Add without Carry to A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8B ii 9B dd BB hh ll AB xb AB xb ff AB xb ee ff AB xb AB xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	--Δ-	Δ Δ Δ Δ
ADDB #opr8i ADDB opr8a ADDB opr16a ADDB oprx0_xysp ADDB oprx9_xysp ADDB oprx16_xysp ADDB [D,xysp] ADDB [oprx16,xysp]	(B) + (M) ⇒ B Add without Carry to B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CB ii DB dd FB hh ll EB xb EB xb ff EB xb ee ff EB xb EB xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	--Δ-	Δ Δ Δ Δ
ADDD #opr16i ADDD opr8a ADDD opr16a ADDD oprx0_xysp ADDD oprx9_xysp ADDD oprx16_xysp ADDD [D,xysp] ADDD [oprx16,xysp]	(A:B) + (M:M+1) ⇒ A:B Add 16-Bit to D (A:B)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C3 jj kk D3 dd F3 hh ll E3 xb E3 xb ff E3 xb ee ff E3 xb E3 xb ee ff	PO RPf RPO RPf RPO frPP fIfRRPf fIPRRPf	----	Δ Δ Δ Δ
ADDX #opr16i ADDX opr8a ADDX opr16a ADDX oprx0_xysp ADDX oprx9_xysp ADDX oprx16_xysp ADDX [D,xysp] ADDX [oprx16,xysp]	(X) + (M:M+1) ⇒ X Add without Carry to X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8B jj kk 18 9B dd 18 BB hh ll 18 AB xb 18 AB xb ff 18 AB xb ee ff 18 AB xb 18 AB xb ee ff	OPO ORPf ORPO ORPf ORPO OfRRPf OfIfRRPf OfIPRRPf	NA NA NA NA NA NA NA NA	Δ Δ Δ Δ
ADDY #opr16i ADDY opr8a ADDY opr16a ADDY oprx0_xysp ADDY oprx9_xysp ADDY oprx16_xysp ADDY [D,xysp] ADDY [oprx16,xysp]	(Y) + (M:M+1) ⇒ Y Add without Carry to Y	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 CB jj kk 18 DB dd 18 FB hh ll 18 EB xb 18 EB xb ff 18 EB xb ee ff 18 EB xb 18 EB xb ee ff	OPO ORPf ORPO ORPf ORPO OfRRPf OfIfRRPf OfIPRRPf	NA NA NA NA NA NA NA NA	Δ Δ Δ Δ

Table A-1. Instruction Set Summary (Sheet 2 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
ADED #opr16i ADED opr8a ADED opr16a ADED oprx0_xysp ADED oprx9_xysp ADED oprx16_xysp ADED [D,xysp] ADED [opr16,xysp]	$(A:B) + (M:M+1) + C \Rightarrow A:B$ Add with Carry to D (A:B)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C3 jj kk 18 D3 dd 18 F3 hh ll 18 E3 xb 18 E3 xb ff 18 E3 xb ee ff 18 E3 xb 18 E3 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
ADEX #opr16i ADEX opr8a ADEX opr16a ADEX oprx0_xysp ADEX oprx9_xysp ADEX oprx16_xysp ADEX [D,xysp] ADEX [opr16,xysp]	$(X) + (M:M+1) + C \Rightarrow X$ Add with Carry to X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 89 jj kk 18 99 dd 18 B9 hh ll 18 A9 xb 18 A9 xb ff 18 A9 xb ee ff 18 A9 xb 18 A9 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
ADEY #opr16i ADEY opr8a ADEY opr16a ADEY oprx0_xysp ADEY oprx9_xysp ADEY oprx16_xysp ADEY [D,xysp] ADEY [opr16,xysp]	$(Y) + (M:M+1) + C \Rightarrow Y$ Add with Carry to Y	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C9 jj kk 18 D9 dd 18 F9 hh ll 18 E9 xb 18 E9 xb ff 18 E9 xb ee ff 18 E9 xb 18 E9 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
ANDA #opr8i ANDA opr8a ANDA opr16a ANDA oprx0_xysp ANDA oprx9_xysp ANDA oprx16_xysp ANDA [D,xysp] ANDA [opr16,xysp]	$(A) \bullet (M) \Rightarrow A$ Logical AND A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	84 ii 94 dd B4 hh ll A4 xb A4 xb ff A4 xb ee ff A4 xb A4 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	---- Δ Δ 0 -	
ANDB #opr8i ANDB opr8a ANDB opr16a ANDB oprx0_xysp ANDB oprx9_xysp ANDB oprx16_xysp ANDB [D,xysp] ANDB [opr16,xysp]	$(B) \bullet (M) \Rightarrow B$ Logical AND B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C4 ii D4 dd F4 hh ll E4 xb E4 xb ff E4 xb ee ff E4 xb E4 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	---- Δ Δ 0 -	
ANDCC #opr8i	$(CCR) \bullet (M) \Rightarrow CCR$ Logical AND CCR with Memory	IMM	10 ii	P	↓↓↓↓	↓↓↓↓
ANDX #opr16i ANDX opr8a ANDX opr16a ANDX oprx0_xysp ANDX oprx9_xysp ANDX oprx16_xysp ANDX [D,xysp] ANDX [opr16,xysp]	$(X) \bullet (M:M+1) \Rightarrow X$ Logical AND X with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 84 jj kk 18 94 dd 18 B4 hh ll 18 A4 xb 18 A4 xb ff 18 A4 xb ee ff 18 A4 xb 18 A4 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ 0 -
ANDY #opr16i ANDY opr8a ANDY opr16a ANDY oprx0_xysp ANDY oprx9_xysp ANDY oprx16_xysp ANDY [D,xysp] ANDY [opr16,xysp]	$(Y) \bullet (M:M+1) \Rightarrow Y$ Logical AND Y with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C4 jj kk 18 D4 dd 18 F4 hh ll 18 E4 xb 18 E4 xb ff 18 E4 xb ee ff 18 E4 xb 18 E4 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ 0 -
ASL opr16a ASL oprx0_xysp ASL oprx9_xysp ASL oprx16_xysp ASL [D,xysp] ASL [opr16,xysp] ASLA ASLB	 Arithmetic Shift Left	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff	rPwO rPw rPwO frPwP fIfrrPw fIPrrPw	NA NA NA NA NA NA	---- Δ Δ Δ Δ
ASLD	 Arithmetic Shift Left Double	INH INH	48 58	O O	NA NA	---- Δ Δ Δ Δ

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
ASLW opr16a ASLW oprx0_xysp ASLW oprx9.xysp ASLW oprx16,xysp ASLW [D,xysp] ASLW [oprx16,xysp] ASLX ASLY	 Arithmetic Shift Left Arithmetic Shift Left Index Register X Arithmetic Shift Left Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 78 hh ll 18 68 xb 18 68 xb ff 18 68 xb ee ff 18 68 xb 18 68 xb ee ff 18 48 18 58	ORPWO NA ORPW NA ORPWO NA OFRFPW NA OFIFRPW NA OFIPRPW NA OO NA OO NA	----	Δ Δ Δ Δ
ASR opr16a ASR oprx0_xysp ASR oprx9.xysp ASR oprx16,xysp ASR [D,xysp] ASR [oprx16,xysp] ASRA ASRB	 Arithmetic Shift Right Arithmetic Shift Right Accumulator A Arithmetic Shift Right Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	77 hh ll 67 xb 67 xb ff 67 xb ee ff 67 xb 67 xb ee ff 47 57	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	----	Δ Δ Δ Δ
ASRW opr16a ASRW oprx0_xysp ASRW oprx9.xysp ASRW oprx16,xysp ASRW [D,xysp] ASRW [oprx16,xysp] ASRX ASRY	 Arithmetic Shift Right Arithmetic Shift Right Index Register X Arithmetic Shift Right Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 77 hh ll 18 67 xb 18 67 xb ff 18 67 xb ee ff 18 67 xb 18 67 xb ee ff 18 47 18 57	ORPWO NA ORPW NA ORPWO NA OFRFPW NA OFIFRPW NA OFIPRPW NA OO NA OO NA	----	Δ Δ Δ Δ
BCC rel8	Branch if Carry Clear (if C = 0)	REL	24 rr	PPP/P ¹	----	----
BCLR opr8a, msk8 BCLR opr16a, msk8 BCLR oprx0_xysp, msk8 BCLR oprx9.xysp, msk8 BCLR oprx16,xysp, msk8	(M) • (mm) ⇒ M Clear Bit(s) in Memory	DIR EXT IDX IDX1 IDX2	4D dd mm 1D hh ll mm 0D xb mm 0D xb ff mm 0D xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	----	Δ Δ 0 -
BCS rel8	Branch if Carry Set (if C = 1)	REL	25 rr	PPP/P ¹	----	----
BEQ rel8	Branch if Equal (if Z = 1)	REL	27 rr	PPP/P ¹	----	----
BGE rel8	Branch if Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	2C rr	PPP/P ¹	----	----
BGND	Place CPU in Background Mode	INH	00	VfPPP	----	----
BGT rel8	Branch if Greater Than (if Z + (N ⊕ V) = 0) (signed)	REL	2E rr	PPP/P ¹	----	----
BHI rel8	Branch if Higher (if C + Z = 0) (unsigned)	REL	22 rr	PPP/P ¹	----	----
BHS rel8	Branch if Higher or Same (if C = 0) (unsigned) same function as BCC	REL	24 rr	PPP/P ¹	----	----
BITA #opr8i BITA opr8a BITA opr16a BITA oprx0_xysp BITA oprx9.xysp BITA oprx16,xysp BITA [D,xysp] BITA [oprx16,xysp]	(A) • (M) Logical AND A with Memory Does not change Accumulator or Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	85 ii 95 dd B5 hh ll A5 xb A5 xb ff A5 xb ee ff A5 xb A5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	Δ Δ 0 -
BITB #opr8i BITB opr8a BITB opr16a BITB oprx0_xysp BITB oprx9.xysp BITB oprx16,xysp BITB [D,xysp] BITB [oprx16,xysp]	(B) • (M) Logical AND B with Memory Does not change Accumulator or Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C5 ii D5 dd F5 hh ll E5 xb E5 xb ff E5 xb ee ff E5 xb E5 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	Δ Δ 0 -
BITX #opr16i BITX opr8a BITX opr16a BITX oprx0_xysp BITX oprx9.xysp BITX oprx16,xysp BITX [D,xysp] BITX [oprx16,xysp]	(X) • (M:M+1) Logical AND X with Memory Does not change Index Register or Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 85 jj kk 18 95 dd 18 B5 hh ll 18 A5 xb 18 A5 xb ff 18 A5 xb ee ff 18 A5 xb 18 A5 xb ee ff	OPO NA ORPf NA ORPO NA ORPf NA ORPO NA OFRPF NA OFIFRPf NA OFIPRPf NA	----	Δ Δ 0 -

Table A-1. Instruction Set Summary (Sheet 4 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
BITY #opr16i BITY opr8a BITY opr16a BITY oprx0_xysp BITY oprx9_xysp BITY oprx16_xysp BITY [D,xysp] BITY [opr16,xysp]	(Y) • (M:M+1) Logical AND Ywith Memory Does not change Index Register or Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C5 jj kk 18 D5 dd 18 F5 hh ll 18 E5 xb 18 E5 xb ff 18 E5 xb ee ff 18 E5 xb 18 E5 xb ee ff	OPO ORP ORPO ORP ORPO OfRPP OfIRP OfIRP	NA NA NA NA NA NA NA NA	---- Δ Δ 0 -
Note 1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.						
BLE rel8	Branch if Less Than or Equal (if $Z + (N \oplus V) = 1$) (signed)	REL	2F rr	PPP/P ¹	----	----
BLO rel8	Branch if Lower (if $C = 1$) (unsigned) same function as BCS	REL	25 rr	PPP/P ¹	----	----
BLS rel8	Branch if Lower or Same (if $C + Z = 1$) (unsigned)	REL	23 rr	PPP/P ¹	----	----
BLT rel8	Branch if Less Than (if $N \oplus V = 1$) (signed)	REL	2D rr	PPP/P ¹	----	----
BMI rel8	Branch if Minus (if $N = 1$)	REL	2B rr	PPP/P ¹	----	----
BNE rel8	Branch if Not Equal (if $Z = 0$)	REL	26 rr	PPP/P ¹	----	----
BPL rel8	Branch if Plus (if $N = 0$)	REL	2A rr	PPP/P ¹	----	----
BRA rel8	Branch Always (if $1 = 1$)	REL	20 rr	PPP	----	----
BRCLR opr8a, msk8, rel8 BRCLR opr16a, msk8, rel8 BRCLR oprx0_xysp, msk8, rel8 BRCLR oprx9_xysp, msk8, rel8 BRCLR oprx16_xysp, msk8, rel8	Branch if (M) • (mm) = 0 (if All Selected Bit(s) Clear)	DIR EXT IDX IDX1 IDX2	4F dd mm rr 1F hh ll mm rr 0F xb mm rr 0F xb ff mm rr 0F xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	----	----
BRN rel8	Branch Never (if $1 = 0$)	REL	21 rr	P	----	----
BRSET opr8, msk8, rel8 BRSET opr16a, msk8, rel8 BRSET oprx0_xysp, msk8, rel8 BRSET oprx9_xysp, msk8, rel8 BRSET oprx16_xysp, msk8, rel8	Branch if (M) • (mm) = 0 (if All Selected Bit(s) Set)	DIR EXT IDX IDX1 IDX2	4E dd mm rr 1E hh ll mm rr 0E xb mm rr 0E xb ff mm rr 0E xb ee ff mm rr	rPPP rfPPP rPPP rfPPP PrfPPP	----	----
BSET opr8, msk8 BSET opr16a, msk8 BSET oprx0_xysp, msk8 BSET oprx9_xysp, msk8 BSET oprx16_xysp, msk8	(M) l (mm) ⇒ M Set Bit(s) in Memory Set CCR flags with respect to the result	DIR EXT IDX IDX1 IDX2	4C dd mm 1C hh ll mm 0C xb mm 0C xb ff mm 0C xb ee ff mm	rPwO rPwP rPwO rPwP frPwPO	----	Δ Δ 0 -
BSR rel8	(SP) - 2 ⇒ SP; RTN _H :RTN _L ⇒ M _(SP) :M _(SP+1) Subroutine address fi PC Branch to Subroutine	REL	07 rr	SPPP	----	----
BTAS opr8, msk8 BTAS opr16a, msk8 BTAS oprx0_xysp, msk8 BTAS oprx9_xysp, msk8 BTAS oprx16_xysp, msk8	(M) l (Mask) ⇒ M Set Bit(s) in Memory Set CCR flags with respect to operand (M) read	DIR EXT IDX IDX1 IDX2	18 35 dd mm 18 36 hh ll mm 18 37 xb mm 18 37 xb ff mm 18 37 xb ee ff mm	OrPwO OrPwP OrPwO OrPwP OfPwPO	NA NA NA NA NA	---- Δ Δ 0 -
BVC rel8	Branch if Overflow Bit Clear (if $V = 0$)	REL	28 rr	PPP/P ¹	----	----
BVS rel8	Branch if Overflow Bit Set (if $V = 1$)	REL	29 rr	PPP/P ¹	----	----
CALL opr16a, page CALL oprx0_xysp, page CALL oprx9_xysp, page CALL oprx16_xysp, page CALL [D,xysp] CALL [opr16, xysp]	(SP) - 2 ⇒ SP; RTN _H :RTN _L ⇒ M _(SP) :M _(SP+1) (SP) - 1 ⇒ SP; (PPG) ⇒ M _(SP) pg ⇒ PPAGE register; Program address ⇒ PC Call subroutine in extended memory (Program may be located on another expansion memory page.) Indirect modes get program address and new pg value based on pointer.	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	4A hh ll pg 4B xb pg 4B xb ff pg 4B xb ee ff pg 4B xb 4B xb ee ff	gnSsPPP gnSsPPP gnSsPPP fgnSsPPP fiIgnSsPPP fiIgnSsPPP	----	----
CBA	(A) - (B) Compare 8-Bit Accumulators	INH	18 17	OO	----	Δ Δ Δ Δ
CLC	0 ⇒ C Translates to ANDCC #\$FE	IMM	10 FE	P	----	---0
CLI	0 ⇒ I Translates to ANDCC #\$EF (enables I-bit interrupts)	IMM	10 EF	P	---0	----

Table A-1. Instruction Set Summary (Sheet 5 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
CLR opr16a CLR oprx0_xysp CLR oprx9_xysp CLR oprx16_xysp CLR [D,xysp] CLR [oprx16,xysp] CLRA CLRB	0 ⇒ M Clear Memory Location 0 ⇒ A Clear Accumulator A 0 ⇒ B Clear Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	79 hh 11 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	----	0100
Note 1. PPP/P indicates this instruction takes three cycles to refill the instruction queue if the branch is taken and one program fetch cycle if the branch is not taken.						
CLRW opr16a CLRW oprx0_xysp CLRW oprx9_xysp CLRW oprx16_xysp CLRW [D,xysp] CLRW [oprx16,xysp] CLR X CLR Y	0 ⇒ M:M+1 Clear Memory Location 0 ⇒ X Clear Index Register X 0 ⇒ Y Clear Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 79 hh 11 18 69 xb 18 69 xb ff 18 69 xb ee ff 18 69 xb 18 69 xb ee ff 18 87 18 C7	OPWO OPW OPWO OPWP OPIFW OPIPW OO OO	NA NA NA NA NA NA NA NA	---- 0100
CLV	0 ⇒ V Translates to ANDCC # \$FD	IMM	10 FD	P	----	--0-
CMPA #opr8i CMPA opr8a CMPA opr16a CMPA oprx0_xysp CMPA oprx9_xysp CMPA oprx16_xysp CMPA [D,xysp] CMPA [oprx16,xysp]	(A) – (M) Compare Accumulator A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	81 ii 91 dd B1 hh 11 A1 xb A1 xb ff A1 xb ee ff A1 xb A1 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	----	Δ Δ Δ Δ
CMPB #opr8i CMPB opr8a CMPB opr16a CMPB oprx0_xysp CMPB oprx9_xysp CMPB oprx16_xysp CMPB [D,xysp] CMPB [oprx16,xysp]	(B) – (M) Compare Accumulator B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C1 ii D1 dd F1 hh 11 E1 xb E1 xb ff E1 xb ee ff E1 xb E1 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrrPf	----	Δ Δ Δ Δ
COM opr16a COM oprx0_xysp COM oprx9_xysp COM oprx16_xysp COM [D,xysp] COM [oprx16,xysp] COMA COMB	(M) ⇒ M equivalent to \$FF – (M) ⇒ M 1's Complement Memory Location (A) ⇒ A Complement Accumulator A (B) ⇒ B Complement Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	71 hh 11 61 xb 61 xb ff 61 xb ee ff 61 xb 61 xb ee ff 41 51	rPwO rPw rPwO frPwP fIfrrPw fIPrrPw O O	----	Δ Δ 0 1
COMW opr16a COMW oprx0_xysp COMW oprx9_xysp COMW oprx16_xysp COMW [D,xysp] COMW [oprx16,xysp] COM X COM Y	(M:M+1) ⇒ M:M+1 equivalent to \$FF – (M:M+1) ⇒ M:M+1 (X) ⇒ X Complement Index Register X (Y) ⇒ Y Complement Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 71 hh 11 18 61 xb 18 61 xb ff 18 61 xb ee ff 18 61 xb 18 61 xb ee ff 18 41 18 51	ORPWO ORPW ORPWO OFRRPW OfIfrrPW OfIPrrPW OO OO	NA NA NA NA NA NA NA NA	---- Δ Δ 0 1
CPD #opr16i CPD opr8a CPD opr16a CPD oprx0_xysp CPD oprx9_xysp CPD oprx16_xysp CPD [D,xysp] CPD [oprx16,xysp]	(A:B) – (M:M+1) Compare D to Memory (16-Bit)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8C jj kk 9C dd BC hh 11 AC xb AC xb ff AC xb ee ff AC xb AC xb ee ff	PO RPf RPO RPf RPO frPP fIfrrPf fIPrrPf	----	Δ Δ Δ Δ
CPED #opr16i CPED opr8a CPED opr16a CPED oprx0_xysp CPED oprx9_xysp CPED oprx16_xysp CPED [D,xysp] CPED [oprx16,xysp]	(A:B) – (M:M+1) – C Compare D to Memory with Borrow	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8C jj kk 18 9C dd 18 BC hh 11 18 AC xb 18 AC xb ff 18 AC xb ee ff 18 AC xb 18 AC xb ee ff	OPO ORPf ORPO ORPf ORPO OfRRPf OfIfrrPf OfIPrrPf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ

Table A-1. Instruction Set Summary (Sheet 6 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
CPES #opr16i CPES opr8a CPES opr16a CPES oprx0_xysp CPES oprx9_xysp CPES oprx16_xysp CPES [D,xysp] CPES [oprx16,xysp]	(SP) – (M:M+1) – C Compare SP to Memory with Borrow	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8F jj kk 18 9F dd 18 BF hh 11 18 AF xb 18 AF xb ff 18 AF xb ee ff 18 AF xb 18 AF xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPpf OfIPRpf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
CPEX #opr16i CPEX opr8a CPEX opr16a CPEX oprx0_xysp CPEX oprx9_xysp CPEX oprx16_xysp CPEX [D,xysp] CPEX [oprx16,xysp]	(X) – (M:M+1) – C Compare X to Memory with Borrow	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8E jj kk 18 9E dd 18 BE hh 11 18 AE xb 18 AE xb ff 18 AE xb ee ff 18 AE xb 18 AE xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPpf OfIPRpf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
CPEY #opr16i CPEY opr8a CPEY opr16a CPEY oprx0_xysp CPEY oprx9_xysp CPEY oprx16_xysp CPEY [D,xysp] CPEY [oprx16,xysp]	(Y) – (M:M+1) – C Compare Y to Memory with Borrow	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8D jj kk 18 9D dd 18 BD hh 11 18 AD xb 18 AD xb ff 18 AD xb ee ff 18 AD xb 18 AD xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPpf OfIPRpf	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
CPS #opr16i CPS opr8a CPS opr16a CPS oprx0_xysp CPS oprx9_xysp CPS oprx16_xysp CPS [D,xysp] CPS [oprx16,xysp]	(SP) – (M:M+1) Compare SP to Memory (16-Bit)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8F jj kk 9F dd BF hh 11 AF xb AF xb ff AF xb ee ff AF xb AF xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPF fIPRPF	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
CPX #opr16i CPX opr8a CPX opr16a CPX oprx0_xysp CPX oprx9_xysp CPX oprx16_xysp CPX [D,xysp] CPX [oprx16,xysp]	(X) – (M:M+1) Compare X to Memory (16-Bit)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8E jj kk 9E dd BE hh 11 AE xb AE xb ff AE xb ee ff AE xb AE xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPF fIPRPF	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
CPY #opr16i CPY opr8a CPY opr16a CPY oprx0_xysp CPY oprx9_xysp CPY oprx16_xysp CPY [D,xysp] CPY [oprx16,xysp]	(Y) – (M:M+1) Compare Y to Memory (16-Bit)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8D jj kk 9D dd BD hh 11 AD xb AD xb ff AD xb ee ff AD xb AD xb ee ff	PO RPf RPO RPf RPO fRPP fIfRPF fIPRPF	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
DAA	Adjust Sum to BCD Decimal Adjust Accumulator A	INH	18 07	oFo	----	Δ Δ ? Δ
DBEQ abdxys, rel9	(cntr) – 1 ⇒ cntr if (cntr) = 0, then Branch else Continue to next instruction Decrement Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
DBNE abdxys, rel9	(cntr) – 1 ⇒ cntr If (cntr) not = 0, then Branch; else Continue to next instruction Decrement Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
DEC opr16a DEC oprx0_xysp DEC oprx9_xysp DEC oprx16_xysp DEC [D,xysp] DEC [oprx16,xysp] DECA DECB	(M) – \$01 ⇒ M Decrement Memory Location (A) – \$01 ⇒ A Decrement A (B) – \$01 ⇒ B Decrement B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	73 hh 11 63 xb 63 xb ff 63 xb ee ff 63 xb 63 xb ee ff 43 53	rPwO rPw rPwO frPwP fIfrPw fIPrPw O O	NA NA NA NA NA NA NA NA	---- Δ Δ Δ –

Table A-1. Instruction Set Summary (Sheet 7 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
DECW opr16a DECW oprx0_xysp DECW oprx9_xysp DECW oprx16_xysp DECW [D,xysp] DECW [oprx16,xysp] DECX DECY	(M:M+1) – \$01 ⇒ M:M+1 Decrement Memory Location (X) – \$01 ⇒ X Decrement X (Y) – \$01 ⇒ Y Decrement Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 73 hh 11 18 63 xb 18 63 xb ff 18 63 xb ee ff 18 63 xb 18 63 xb ee ff18 18 43 18 53	ORPWO ORPW ORPWO OfRPWP OfIRRPW OfIPRPW OO OO	NA NA NA NA NA NA NA NA	---- Δ Δ Δ -
DES	(SP) – \$0001 ⇒ SP Translates to LEAS –1, SP	IDX	1B 9F	Pf	----	----
DEX	(X) – \$0001 ⇒ X Decrement Index Register X	INH	09	O	----	-Δ--
DEY	(Y) – \$0001 ⇒ Y Decrement Index Register Y	INH	03	O	----	-Δ--
EDIV	(Y:D) ÷ (X) ⇒ Y Remainder fi D 32 by 16 Bit ⇒ 16 Bit Divide (unsigned)	INH	11	fffffffff0	----	Δ Δ Δ Δ
EDIVS	(Y:D) ÷ (X) ⇒ Y Remainder fi D 32 by 16 Bit ⇒ 16 Bit Divide (signed)	INH	18 14	offffffff0	----	Δ Δ Δ Δ
EMACS* opr16a ¹	(M _(X) :M _(X+1)) × (M _(Y) :M _(Y+1)) + (M-M+3) ⇒ M-M+3 16 by 16 Bit ⇒ 32 Bit Multiply and Accumulate (signed)	Special	18 12 hh 11	ORRRWWP ORROffRRWWP	----	Δ Δ Δ Δ
* The cycle timing of CPU12V1 has been improved by removing two free cycles compared to CPU12V0: ORROffRRWWP						
EMAXD oprx0_xysp EMAXD oprx9_xysp EMAXD oprx16_xysp EMAXD [D,xysp] EMAXD [oprx16,xysp]	MAX((D), (M:M+1)) ⇒ D MAX of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1A xb 18 1A xb ff 18 1A xb ee ff 18 1A xb 18 1A xb ee ff	ORPf ORPO OfRPP OfIRPpf OfIPRPf	----	Δ Δ Δ Δ
EMAXM oprx0_xysp EMAXM oprx9_xysp EMAXM oprx16_xysp EMAXM [D,xysp] EMAXM [oprx16,xysp]	MAX((D), (M:M+1)) ⇒ M:M+1 MAX of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1E xb 18 1E xb ff 18 1E xb ee ff 18 1E xb 18 1E xb ee ff	ORPW ORPWO OfRPWP OfIRPWP OfIPRPW	----	Δ Δ Δ Δ
EMIND oprx0_xysp EMIND oprx9_xysp EMIND oprx16_xysp EMIND [D,xysp] EMIND [oprx16,xysp]	MIN((D), (M:M+1)) ⇒ D MIN of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1B xb 18 1B xb ff 18 1B xb ee ff 18 1B xb 18 1B xb ee ff	ORPf ORPO OfRPP OfIRPpf OfIPRPf	----	Δ Δ Δ Δ
EMINM oprx0_xysp EMINM oprx9_xysp EMINM oprx16_xysp EMINM [D,xysp] EMINM [oprx16,xysp]	MIN((D), (M:M+1)) ⇒ M:M+1 MIN of 2 Unsigned 16-Bit Values N, Z, V and C status bits reflect result of internal compare ((D) – (M:M+1))	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1F xb 18 1F xb ff 18 1F xb ee ff 18 1F xb 18 1F xb ee ff	ORPW ORPWO OfRPWP OfIRPWP OfIPRPW	----	Δ Δ Δ Δ
EMUL	(D) × (Y) ⇒ Y:D 16 by 16 Bit Multiply (unsigned)	INH	13	O fff0	----	Δ Δ – Δ
EMULS	(D) × (Y) ⇒ Y:D 16 by 16 Bit Multiply (signed)	INH	18 13	O ffo offfo	----	Δ Δ – D
EORA #opr8i EORA opr8a EORA opr16a EORA oprx0_xysp EORA oprx9_xysp EORA oprx16_xysp EORA [D,xysp] EORA [oprx16,xysp]	(A) ⊕ (M) ⇒ A Exclusive-OR A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	88 ii 98 dd B8 hh 11 A8 xb A8 xb ff A8 xb ee ff A8 xb A8 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ 0 –
EORB #opr8i EORB opr8a EORB opr16a EORB oprx0_xysp EORB oprx9_xysp EORB oprx16_xysp EORB [D,xysp] EORB [oprx16,xysp]	(B) ⊕ (M) ⇒ B Exclusive-OR B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C8 ii D8 dd F8 hh 11 E8 xb E8 xb ff E8 xb ee ff E8 xb E8 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ 0 –

Table A-1. Instruction Set Summary (Sheet 8 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
EORX #opr16i EORX opr8a EORX opr16a EORX oprx0_xysp EORX oprx9,xysp EORX oprx16,xysp EORX [D,xysp] EORX [opr16,xysp]	$(X) \oplus (M:M+1) \Rightarrow X$ Exclusive-OR X with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 88 jj kk 18 98 dd 18 B8 hh 11 18 A8 xb 18 A8 xb ff 18 A8 xb ee ff 18 A8 xb 18 A8 xb ee ff	OPO ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
EORY #opr16i EORY opr8a EORY opr16a EORY oprx0_xysp EORY oprx9,xysp EORY oprx16,xysp EORY [D,xysp] EORY [opr16,xysp]	$(Y) \oplus (M:M+1) \Rightarrow Y$ Exclusive-OR Y with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C8 jj kk 18 D8 dd 18 F8 hh 11 18 E8 xb 18 E8 xb ff 18 E8 xb ee ff 18 E8 xb 18 E8 xb ee ff	OPO ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
ETBL oprx0_xysp	$(M:M+1) + [(B) \times ((M+2:M+3) - (M:M+1))] \Rightarrow D$ 16-Bit Table Lookup and Interpolate Initialize B, and index before ETBL. <ea> points at first table entry (M:M+1) and B is fractional part of lookup value (no indirect addr. modes or extensions allowed)	IDX	18 3F xb	ORRffffP ORRfffffP	----	$\Delta \Delta - \Delta$
EXG abcdxys,abcdxys	$(r1) \Leftrightarrow (r2)$ (if r1 and r2 same size) or $\$00:(r1) \Rightarrow r2$ (if r1=8-bit; r2=16-bit) or $(r1_{low}) \Leftrightarrow (r2)$ (if r1=16-bit; r2=8-bit) r1 and r2 may be A, B, CCR, D, X, Y, or SP	INH	B7 eb	P	----	----
FDIV	$(D) \div (X) \Rightarrow X$; Remainder fi D 16 by 16 Bit Fractional Divide	INH	18 11	Offfffffffo	----	$- \Delta \Delta \Delta$
GLDAA opr8a GLDAA opr16a GLDAA oprx0_xysp GLDAA oprx9,xysp GLDAA oprx16,xysp GLDAA [D,xysp] GLDAA [opr16,xysp]	$G(M) \Rightarrow A$ Load Accumulator A from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 96 dd 18 B6 hh 11 18 A6 xb 18 A6 xb ff 18 A6 xb ee ff 18 A6 xb 18 A6 xb ee ff	OrPF OrPO OrPF OrPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GLDAB opr8a GLDAB opr16a GLDAB oprx0_xysp GLDAB oprx9,xysp GLDAB oprx16,xysp GLDAB [D,xysp] GLDAB [opr16,xysp]	$G(M) \Rightarrow B$ Load Accumulator B from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 D6 dd 18 F6 hh 11 18 E6 xb 18 E6 xb ff 18 E6 xb ee ff 18 E6 xb 18 E6 xb ee ff	OrPF OrPO OrPF OrPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GLDD opr8a GLDD opr16a GLDD oprx0_xysp GLDD oprx9,xysp GLDD oprx16,xysp GLDD [D,xysp] GLDD [opr16,xysp]	$G(M:M+1) \Rightarrow A:B$ Load Double Accumulator D (A:B) from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 DC dd 18 FC hh 11 18 EC xb 18 EC xb ff 18 EC xb ee ff 18 EC xb 18 EC xb ee ff	ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GLDS opr8a GLDS opr16a GLDS oprx0_xysp GLDS oprx9,xysp GLDS oprx16,xysp GLDS [D,xysp] GLDS [opr16,xysp]	$G(M:M+1) \Rightarrow SP$ Load Stack Pointer from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 DF dd 18 FF hh 11 18 EF xb 18 EF xb ff 18 EF xb ee ff 18 EF xb 18 EF xb ee ff	ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GLDX opr8a GLDX opr16a GLDX oprx0_xysp GLDX oprx9,xysp GLDX oprx16,xysp GLDX [D,xysp] GLDX [opr16,xysp]	$G(M:M+1) \Rightarrow X$ Load Index Register X from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 DE dd 18 FE hh 11 18 EE xb 18 EE xb ff 18 EE xb ee ff 18 EE xb 18 EE xb ee ff	ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$

Note: 1. opr16a is an extended address specification. Both X and Y point to source operands.

Table A-1. Instruction Set Summary (Sheet 9 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
GLDY opr8a GLDY opr16a GLDY oprx0_xysp GLDY oprx9_xysp GLDY oprx16_xysp GLDY [D,xysp] GLDY [oprx16,xysp]	$G(M:M+1) \Rightarrow Y$ Load Index Register Y from Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 DD dd 18 FD hh 11 18 ED xb 18 ED xb ff 18 ED xb ee ff 18 ED xb 18 ED xb ee ff	ORPf ORPO ORPf ORPO OfRFP OfIFRPF OfIPRPF	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTAA opr8a GSTAA opr16a GSTAA oprx0_xysp GSTAA oprx9_xysp GSTAA oprx16_xysp GSTAA [D,xysp] GSTAA [oprx16,xysp]	$(A) \Rightarrow G(M)$ Store Accumulator A to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5A dd 18 7A hh 11 18 6A xb 18 6A xb ff 18 6A xb ee ff 18 6A xb 18 6A xb ee ff	OPw OPwO OPw OPwO OPwP OPIfw OPIPw	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTAB opr8a GSTAB opr16a GSTAB oprx0_xysp GSTAB oprx9_xysp GSTAB oprx16_xysp GSTAB [D,xysp] GSTAB [oprx16,xysp]	$(B) \Rightarrow G(M)$ Store Accumulator B to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5B dd 18 7B hh 11 18 6B xb 18 6B xb ff 18 6B xb ee ff 18 6B xb 18 6B xb ee ff	OPw OPwO OPw OPwO OPwP OPIfw OPIPw	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTD opr8a GSTD opr16a GSTD oprx0_xysp GSTD oprx9_xysp GSTD oprx16_xysp GSTD [D,xysp] GSTD [oprx16,xysp]	$(A) \Rightarrow G(M), (B) \Rightarrow G(M+1)$ Store Double Accumulator to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5C dd 18 7C hh 11 18 6C xb 18 6C xb ff 18 6C xb ee ff 18 6C xb 18 6C xb ee ff	OPW OPWO OPW OPWO OPWP OPIfW OPIPW	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTS opr8a GSTS opr16a GSTS oprx0_xysp GSTS oprx9_xysp GSTS oprx16_xysp GSTS [D,xysp] GSTS [oprx16,xysp]	$(SP) \Rightarrow G(M:M+1)$ Store Stack Pointer to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5F dd 18 7F hh 11 18 6F xb 18 6F xb ff 18 6F xb ee ff 18 6F xb 18 6F xb ee ff	OPW OPWO OPW OPWO OPWP OPIfW OPIPW	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTX opr8a GSTX opr16a GSTX oprx0_xysp GSTX oprx9_xysp GSTX oprx16_xysp GSTX [D,xysp] GSTX [oprx16,xysp]	$(X) \Rightarrow G(M:M+1)$ Store Index Register X to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5E dd 18 7E hh 11 18 6E xb 18 6E xb ff 18 6E xb ee ff 18 6E xb 18 6E xb ee ff	OPW OPWO OPW OPWO OPWP OPIfW OPIPW	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
GSTY opr8a GSTY opr16a GSTY oprx0_xysp GSTY oprx9_xysp GSTY oprx16_xysp GSTY [D,xysp] GSTY [oprx16,xysp]	$(Y) \Rightarrow G(M:M+1)$ Store Index Register Y to Global Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 5D dd 18 7D hh 11 18 6D xb 18 6D xb ff 18 6D xb ee ff 18 6D xb 18 6D xb ee ff	OPW OPWO OPW OPWO OPWP OPIfW OPIPW	NA NA NA NA NA NA NA	---- $\Delta \Delta 0 -$
IBEQ abdxys, rel9	$(cntr) + 1 \Rightarrow cntr$ If $(cntr) = 0$, then Branch else Continue to next instruction Increment Counter and Branch if = 0 (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
IBNE abdxys, rel9	$(cntr) + 1 \Rightarrow cntr$ if $(cntr) \neq 0$, then Branch; else Continue to next instruction Increment Counter and Branch if $\neq 0$ (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
IDIV	$(D) \div (X) \Rightarrow X$; Remainder $\Rightarrow D$ 16 by 16 Bit Integer Divide (unsigned)	INH	18 10	0fffffff0	----	$-\Delta 0 \Delta$
IDIVS	$(D) \div (X) \Rightarrow X$; Remainder $\Rightarrow D$ 16 by 16 Bit Integer Divide (signed)	INH	18 15	0fffffff0	----	$\Delta \Delta \Delta \Delta$

Table A-1. Instruction Set Summary (Sheet 10 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C	
INC opr16a INC oprx0_xysp INC oprx9,xysp INC oprx16,xysp INC [D,xysp] INC [oprx16,xysp] INCA INCB	(M) + \$01 ⇒ M Increment Memory Byte (A) + \$01 ⇒ AIncrement Acc. A (B) + \$01 ⇒ BIncrement Acc. B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	72 hh 11 62 xb 62 xb ff 62 xb ee ff 62 xb 62 xb ee ff 42 52	rPwO rPw rPwO frPwP fIfrrPw fIPrPw O O	----	Δ Δ Δ –	
INCW opr16a INCW oprx0_xysp INCW oprx9,xysp INCW oprx16,xysp INCW [D,xysp] INCW [oprx16,xysp] INCX INCY	(M:M+1) + \$01 ⇒ M:M+1 Increment Memory (X) + \$01 ⇒ XIncrement Index Register X (Y) + \$01 ⇒ YIncrement Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 72 hh 11 18 62 xb 18 62 xb ff 18 62 xb ee ff 18 62 xb 18 62 xb ee ff 18 42 18 52	ORPWO ORPW ORPWO OfRPWP OfIRPWP OfIPRPW OO OO	NA NA NA NA NA NA NA NA	----	Δ Δ Δ –
INS	(SP) + \$0001 ⇒ SP Translates to LEAS 1,SP	IDX	1B 81	Pf	----	----	
INX	(X) + \$0001 ⇒ X Increment Index Register X	INH	08	O	----	– Δ – –	
INY	(Y) + \$0001 ⇒ Y Increment Index Register Y	INH	02	O	----	– Δ – –	
JMP opr16a JMP oprx0_xysp JMP oprx9,xysp JMP oprx16,xysp JMP [D,xysp] JMP [oprx16,xysp]	Routine address ⇒ PC Jump	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	06 hh 11 05 xb 05 xb ff 05 xb ee ff 05 xb 05 xb ee ff	PPP PPP PPP fPPP fIfPPP fIfPPP	----	----	
JSR opr8a JSR opr16a JSR oprx0_xysp JSR oprx9,xysp JSR oprx16,xysp JSR [D,xysp] JSR [oprx16,xysp]	(SP) – 2 ⇒ SP; RTN _H :RTN _L ⇒ M _(SP) :M _(SP+1) ; Subroutine address ⇒ PC Jump to Subroutine	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	17 dd 16 hh 11 15 xb 15 xb ff 15 xb ee ff 15 xb 15 xb ee ff	SPPP SPPP PPPS PPPS fPPPS fIfPPPS fIfPPPS	----	----	
LBCC rel16	Long Branch if Carry Clear (if C = 0)	REL	18 24 qq rr	OPPP/OPO ¹	----	----	
LBCS rel16	Long Branch if Carry Set (if C = 1)	REL	18 25 qq rr	OPPP/OPO ¹	----	----	
LBEQ rel16	Long Branch if Equal (if Z = 1)	REL	18 27 qq rr	OPPP/OPO ¹	----	----	
LBGE rel16	Long Branch Greater Than or Equal (if N ⊕ V = 0) (signed)	REL	18 2C qq rr	OPPP/OPO ¹	----	----	
LBGT rel16	Long Branch if Greater Than (if Z + (N ⊕ V) = 0) (signed)	REL	18 2E qq rr	OPPP/OPO ¹	----	----	
LBHI rel16	Long Branch if Higher (if C + Z = 0) (unsigned)	REL	18 22 qq rr	OPPP/OPO ¹	----	----	
LBHS rel16	Long Branch if Higher or Same (if C = 0) (unsigned) same function as LBCC	REL	18 24 qq rr	OPPP/OPO ¹	----	----	
LBLE rel16	Long Branch if Less Than or Equal (if Z + (N ⊕ V) = 1) (signed)	REL	18 2F qq rr	OPPP/OPO ¹	----	----	
LBLO rel16	Long Branch if Lower (if C = 1) (unsigned) same function as LBCS	REL	18 25 qq rr	OPPP/OPO ¹	----	----	
LBLS rel16	Long Branch if Lower or Same (if C + Z = 1) (unsigned)	REL	18 23 qq rr	OPPP/OPO ¹	----	----	
LBLT rel16	Long Branch if Less Than (if N ⊕ V = 1) (signed)	REL	18 2D qq rr	OPPP/OPO ¹	----	----	
LBMI rel16	Long Branch if Minus (if N = 1)	REL	18 2B qq rr	OPPP/OPO ¹	----	----	
LBNE rel16	Long Branch if Not Equal (if Z = 0)	REL	18 26 qq rr	OPPP/OPO ¹	----	----	
LBPL rel16	Long Branch if Plus (if N = 0)	REL	18 2A qq rr	OPPP/OPO ¹	----	----	
LBRA rel16	Long Branch Always (if 1 = 1)	REL	18 20 qq rr	OPPP	----	----	
LBRN rel16	Long Branch Never (if 1 = 0)	REL	18 21 qq rr	OPO	----	----	
LBVC rel16	Long Branch if Overflow Bit Clear (if V = 0)	REL	18 28 qq rr	OPPP/OPO ¹	----	----	
LBVS rel16	Long Branch if Overflow Bit Set (if V = 1)	REL	18 29 qq rr	OPPP/OPO ¹	----	----	

Table A-1. Instruction Set Summary (Sheet 11 of 20)


Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
LDAA #opr8i LDAA opr8a LDAA opr16a LDAA oprx0_xysp LDAA oprx9_xysp LDAA oprx16_xysp LDAA [D,xysp] LDAA [opr16,xysp]	(M) ⇒ A Load Accumulator A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	86 ii 96 dd B6 hh ll A6 xb A6 xb ff A6 xb ee ff A6 xb A6 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ 0 -
LDAB #opr8i LDAB opr8a LDAB opr16a LDAB oprx0_xysp LDAB oprx9_xysp LDAB oprx16_xysp LDAB [D,xysp] LDAB [opr16,xysp]	(M) ⇒ B Load Accumulator B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C6 ii D6 dd F6 hh ll E6 xb E6 xb ff E6 xb ee ff E6 xb E6 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ 0 -
LDD #opr16i LDD opr8a LDD opr16a LDD oprx0_xysp LDD oprx9_xysp LDD oprx16_xysp LDD [D,xysp] LDD [opr16,xysp]	(M:M+1) ⇒ A:B Load Double Accumulator D (A:B)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CC jj kk DC dd FC hh ll EC xb EC xb ff EC xb ee ff EC xb EC xb ee ff	PO RPf RPO RPf RPO frPP fIfRRPf fIPRRPf	----	Δ Δ 0 -
LDS #opr16i LDS opr8a LDS opr16a LDS oprx0_xysp LDS oprx9_xysp LDS oprx16_xysp LDS [D,xysp] LDS [opr16,xysp]	(M:M+1) ⇒ SP Load Stack Pointer	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CF jj kk DF dd FF hh ll EF xb EF xb ff EF xb ee ff EF xb EF xb ee ff	PO RPf RPO RPf RPO frPP fIfRRPf fIPRRPf	----	Δ Δ 0 -
LDX #opr16i LDX opr8a LDX opr16a LDX oprx0_xysp LDX oprx9_xysp LDX oprx16_xysp LDX [D,xysp] LDX [opr16,xysp]	(M:M+1) ⇒ X Load Index Register X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CE jj kk DE dd FE hh ll EE xb EE xb ff EE xb ee ff EE xb EE xb ee ff	PO RPf RPO RPf RPO frPP fIfRRPf fIPRRPf	----	Δ Δ 0 -
LDY #opr16i LDY opr8a LDY opr16a LDY oprx0_xysp LDY oprx9_xysp LDY oprx16_xysp LDY [D,xysp] LDY [opr16,xysp]	(M:M+1) ⇒ Y Load Index Register 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 fIfRRPf fIPRRPf	----	Δ Δ 0 -
LEAS* oprx0_xysp LEAS oprx9_xysp LEAS oprx16_xysp	Effective Address ⇒ SP Load Effective Address into SP	IDX IDX1 IDX2	1B xb 1B xb ff 1B xb ee ff	P PO PP	Pf PO PP	----
LEAX* oprx0_xysp LEAX oprx9_xysp LEAX oprx16_xysp	Effective Address ⇒ X Load Effective Address into X	IDX IDX1 IDX2	1A xb 1A xb ff 1A xb ee ff	P PO PP	Pf PO PP	----
LEAY* oprx0_xysp LEAY oprx9_xysp LEAY oprx16_xysp	Effective Address ⇒ Y Load Effective Address into Y	IDX IDX1 IDX2	19 xb 19 xb ff 19 xb ee ff	P PO PP	Pf PO PP	----
* The cycle timing of CPU12V1 has been improved by removing one free cycle compared to CPU12V0 : P						
LSL opr16a LSL oprx0_xysp LSL oprx9_xysp LSL oprx16_xysp LSL [D,xysp] LSL [opr16,xysp] LSLA LSLB	 Logical Shift Left same function as ASL Logical Shift Accumulator A to Left Logical Shift Accumulator B to Left	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	78 hh ll 68 xb 68 xb ff 68 xb ee ff 68 xb 68 xb ee ff 48 58	rPwO rPw rPwO frPPw fIfrrPw fIPrrPw O O	----	Δ Δ Δ Δ

Table A-1. Instruction Set Summary (Sheet 12 of 20)

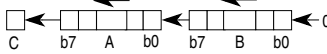
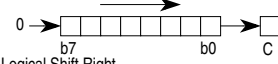
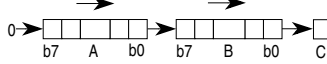
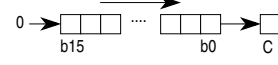
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C	
LSLD	 Logical Shift Left D Accumulator same function as ASLD	INH	59	O	----	Δ Δ Δ Δ	
LSR opr16a LSR oprx0_xysp LSR oprx9_xysp LSR oprx16_xysp LSR [D_xysp] LSR [opr16_xysp] LSRA LSRB	 Logical Shift Right Logical Shift Accumulator A to Right Logical Shift Accumulator B to Right	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	74 hh 11 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 Accumulator	INH	49	O	----	0 Δ Δ Δ	
LSRW opr16a LSRW oprx0_xysp LSRW oprx9_xysp LSRW oprx16_xysp LSRW [D_xysp] LSRW [opr16_xysp] LSRX LSRY	 Logical Shift Index Register X to Right Logical Shift Index Register Y to Right	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 74 hh 11 18 64 xb 18 64 xb ff 18 64 xb ee ff 18 64 xb 18 64 xb ee ff 18 44 18 54	ORPWO ORPW ORPWO OfRPWP OfIfrPW OfIPrPW OO OO	----	0 Δ Δ Δ	
MAXA oprx0_xysp MAXA oprx9_xysp MAXA oprx16_xysp MAXA [D_xysp] MAXA [opr16_xysp]	MAX((A), (M)) ⇒ A MAX of 2 Unsigned 8-Bit Values N, Z, V and C status bits reflect result of internal compare ((A) – (M)).	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 18 xb 18 18 xb ff 18 18 xb ee ff 18 18 xb 18 18 xb ee ff	OrPf OrPO OfrrPP OfIfrPf OfIPrPf	----	Δ Δ Δ Δ	
MAXM oprx0_xysp MAXM oprx9_xysp MAXM oprx16_xysp MAXM [D_xysp] MAXM [opr16_xysp]	MAX((A), (M)) ⇒ M MAX of 2 Unsigned 8-Bit Values N, Z, V and C status bits reflect result of internal compare ((A) – (M)).	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1C xb 18 1C xb ff 18 1C xb ee ff 18 1C xb 18 1C xb ee ff	OrPw OrPwO OfrrPwP OfIfrPw OfIPrPw	----	Δ Δ Δ Δ	
MEM (Only for CPU12V0 and CPU12XV0)	m (grade) fi M _Y ; (X) + 4 ⇒ X; (Y) + 1 ⇒ Y; A unchanged if (A) < P1 or (A) > P2 then m = 0, else m = MIN(((A) – P1) × S1, (P2 – (A)) × S2, \$FF) where: A = current crisp input value; X points at 4-byte data structure that describes a trapezoidal membership function (P1, P2, S1, S2); Y points at fuzzy input (RAM location).	Special	01	RRfOW	--?–	????	
MINA oprx0_xysp MINA oprx9_xysp MINA oprx16_xysp MINA [D_xysp] MINA [opr16_xysp]	MIN((A), (M)) ⇒ A MIN of 2 Unsigned 8-Bit Values N, Z, V and C status bits reflect result of internal compare ((A) – (M)).	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 19 xb 18 19 xb ff 18 19 xb ee ff 18 19 xb 18 19 xb ee ff	OrPf OrPO OfrrPP OfIfrPf OfIPrPf	----	Δ Δ Δ Δ	
MINM oprx0_xysp MINM oprx9_xysp MINM oprx16_xysp MINM [D_xysp] MINM [opr16_xysp]	MIN((A), (M)) ⇒ M MIN of 2 Unsigned 8-Bit Values N, Z, V and C status bits reflect result of internal compare ((A) – (M)).	IDX IDX1 IDX2 [D,IDX] [IDX2]	18 1D xb 18 1D xb ff 18 1D xb ee ff 18 1D xb 18 1D xb ee ff	OrPw OrPwO OfrrPwP OfIfrPw OfIPrPw	----	Δ Δ Δ Δ	
MOVB #opr8i, opr16a ¹ MOVB #opr8i, oprx0_xysp ¹ MOVB #opr8i, oprx9_xysp ¹ MOVB #opr8i, oprx16_xysp ¹ MOVB #opr8i, [D_xysp] ¹ MOVB #opr8i, [opr16_xysp] ¹	# ⇒ M Immediate to Memory Byte-Move (8-Bit)	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0B ii hh 11 18 08 xb ² ii 18 08 xb ² ff ii 18 08 xb ² ee ff ii 18 08 xb ² ii 18 08 xb ² ee ff ii	OPwP OPwO OPwP OPPwO OPIOW OPIOWP	OPwP OPwO NA NA NA NA	----	----
MOVB opr16a, opr16a ¹ MOVB opr16a, oprx0_xysp ¹ MOVB opr16a, oprx9_xysp ¹ MOVB opr16a, oprx16_xysp ¹ MOVB opr16a, [D_xysp] ¹ MOVB opr16a, [opr16_xysp] ¹	(M) ₁ ⇒ M ₂ Memory to Memory Byte-Move (8-Bit) EXT Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0C hh 11 hh 11 18 09 xb ² hh 11 18 09 xb ² ff hh 11 18 09 xb ² ee ff hh 11 18 09 xb ² hh 11 18 09 xb ² ee ff hh 11	OPrPwO OPrPw OPrPwO OPPrPw OPrIPw OPPrIPw	OrFwPO OPrPw NA NA NA NA	----	----

Table A-1. Instruction Set Summary (Sheet 13 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
MOVB oprx0_xysp, opr16a ¹ MOVB oprx0_xysp, oprx0_xysp ¹ MOVB oprx0_xysp, oprx9_xysp ¹ MOVB oprx0_xysp, oprx16_xysp ¹ MOVB oprx0_xysp, [D_xysp] ¹ MOVB oprx0_xysp, [opr16_xysp] ¹	(M ₁) ⇒ M ₂ Memory to Memory Byte-Move (8-Bit) IDX Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0D xb hh 11 18 0A xb xb 18 0A xbxb ff 18 0A xb xb ee ff 18 0A xb xb 18 0A xb xb ee ff	OrPPw OrPwP OrPOw OrPwO OrPPw NA OrPOPw NA OrPIOw NA OrPPIOw NA	----	----
MOVB oprx9_xysp, opr16a ¹ MOVB oprx9_xysp, oprx0_xysp ¹ MOVB oprx9_xysp, oprx9_xysp ¹ MOVB oprx9_xysp, oprx16_xysp ¹ MOVB oprx9_xysp, [D_xysp] ¹ MOVB oprx9_xysp, [opr16_xysp] ¹	(M ₁) ⇒ M ₂ Memory to Memory Byte-Move (8-Bit), IDX1 Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0D xb ff hh 11 18 0A xb ff xb 18 0A xb ff xb ff 18 0A xb ff xb ee ff 18 0A xb ff xb 18 0A xb ff xb ee ff	OPROPw NA OPROOw NA OPROPw NA OPROOPw NA OPROIOw NA OPROPIOw NA	----	----

Notes: 1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

MOVB oprx16_xysp, opr16a ¹ MOVB oprx16_xysp, oprx0_xysp ¹ MOVB oprx16_xysp, oprx9_xysp ¹ MOVB oprx16_xysp, oprx16_xysp ¹ MOVB oprx16_xysp, [D_xysp] ¹ MOVB oprx16_xysp, [opr16_xysp] ¹	(M ₁) ⇒ M ₂ Memory to Memory Byte-Move (8-Bit), IDX2 Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0D xb ee ff hh 11 18 0A xb ee ff xb 18 0A xb ee ff xb ff 18 0A xb ee ff xb ee ff 18 0A xb ee ff xb 18 0A xb ee ff xb ee ff	OPrPPw NA OPrPOw NA OPrPPw NA OPrPOPw NA OPrPIOw NA OPrPPIOw NA	----	----
MOVB [D_xysp], opr16a ¹ MOVB [D_xysp], oprx0_xysp ¹ MOVB [D_xysp], oprx9_xysp ¹ MOVB [D_xysp], oprx16_xysp ¹ MOVB [D_xysp], [D_xysp] ¹ MOVB [D_xysp], [opr16_xysp] ¹	(M ₁) ⇒ M ₂ Memory to Memory Byte-Move (8-Bit), [D,IDX] Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0D xb hh 11 18 0A xb xb 18 0A xb xb ff 18 0A xb xb ee ff 18 0A xb xb 18 0A xb xb ee ff	OIPrfPw NA OIPrfOw NA OIPrfPw NA OIPrfOPw NA OIPrfIOw NA OIPrfPIOw NA	----	----
MOVW [opr16_xysp], opr16a ¹ MOVW [opr16_xysp], oprx0_xysp ¹ MOVW [opr16_xysp], oprx9_xysp ¹ MOVW [opr16_xysp], oprx16_xysp ¹ MOVW [opr16_xysp], [D_xysp] ¹ MOVW [opr16_xysp], [opr16_xysp] ¹	(M ₁) ⇒ M ₂ Memory to Memory Word-Move (8-Bit), [IDX2] Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 0D xb ee ff hh 11 18 0A xb ee ff xb 18 0A xb ee ff xb ff 18 0A xb ee ff xb ee ff 18 0A xb ee ff xb 18 0A xb ee ff xb ee ff	OPIPrfPw NA OPIPrfO- NA wOPIPrfP- NA wOPIPrfOPwOP NA IPrfIO- NA wOPIPrfPIOw NA	----	----
MOVW #opr16i, opr16a ¹ MOVW #opr16i, oprx0_xysp ¹ MOVW #opr16i, oprx9_xysp ¹ MOVW #opr16i, oprx16_xysp ¹ MOVW #opr16i, [D_xysp] ¹ MOVW #opr16i, [opr16_xysp] ¹	# ⇒ M:M+1 ₂ Immediate to Memory Word-Move (16-Bit)	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 03 jj kk hh 11 18 00 xb ² jj kk 18 00 xb ² ff jj kk 18 00 xb ² ee ff jj kk 18 00 xb ² jj kk 18 00 xb ² ee ff jj kk	OPFWO OPFWO OPWP OPWP OPFWO NA OPFWP NA OPIWP NA OPIFWP NA	----	----
MOVW opr16a, opr16a ¹ MOVW opr16a, oprx0_xysp ¹ MOVW opr16a, oprx9_xysp ¹ MOVW opr16a, oprx16_xysp ¹ MOVW opr16a, [D_xysp] ¹ MOVW opr16a, [opr16_xysp] ¹	(M:M+1 ₁) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), EXT Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 04 hh 11 hh 11 18 01 xb ² hh 11 18 01 xb ² ff hh 11 18 01 xb ² ee ff hh 11 18 01 xb ² hh 11 18 01 xb ² ee ff hh 11	OPRPWO ORPWP OPRPW ORPWP OPRPWO NA OPRPWP NA OPRIWP NA OPPRIPW NA	----	----
MOVW oprx0_xysp, opr16a ¹ MOVW oprx0_xysp, oprx0_xysp ¹ MOVW oprx0_xysp, oprx9_xysp ¹ MOVW oprx0_xysp, oprx16_xysp ¹ MOVW oprx0_xysp, [D_xysp] ¹ MOVW oprx0_xysp, [opr16_xysp] ¹	(M:M+1 ₁) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), IDX Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 05 xb hh 11 18 02 xb xb 18 02 xb xb ff 18 02 xb xb ee ff 18 02 xb xb 18 02 xb xb ee ff	ORPPW ORPWP ORPOW ORPWO ORPPW NA ORPOPW NA ORPIOw NA ORPPIOw NA	----	----
MOVW oprx9_xysp, opr16a ¹ MOVW oprx9_xysp, oprx0_xysp ¹ MOVW oprx9_xysp, oprx9_xysp ¹ MOVW oprx9_xysp, oprx16_xysp ¹ MOVW oprx9_xysp, [D_xysp] ¹ MOVW oprx9_xysp, [opr16_xysp] ¹	(M:M+1 ₁) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), IDX1 Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 05 xb ff hh 11 18 02 xb ff xb 18 02 xb ff xb ff 18 02 xb ff xb ee ff 18 02 xb ff xb 18 02 xb ff xb ee ff	OPROPw NA OPROOw NA OPROPw NA OPROOPw NA OPROIOw NA OPROPIOw NA	----	----
MOVW oprx16_xysp, opr16a ¹ MOVW oprx16_xysp, oprx0_xysp ¹ MOVW oprx16_xysp, oprx9_xysp ¹ MOVW oprx16_xysp, oprx16_xysp ¹ MOVW oprx16_xysp, [D_xysp] ¹ MOVW oprx16_xysp, [opr16_xysp] ¹	(M:M+1 ₁) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), IDX2 Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 05 xb ee ff hh 11 18 02 xb ee ff xb 18 02 xb ee ff xb ff 18 02 xb ee ff xb ee ff 18 02 xb ee ff xb 18 02 xb ee ff xb ee ff	OPRPPW OPWP OPRPPW OPWP WOPRPPWOPR- NA WOPRPPWOPR- NA PIOWOPRPPW NA NA	----	----

Notes: 1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

Table A-1. Instruction Set Summary (Sheet 14 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
MOVW [D_xysp], opr16a ¹ MOVW [D_xysp], oprx0_xysp ¹ MOVW [D_xysp], oprx9_xysp ¹ MOVW [D_xysp], oprx16_xysp ¹ MOVW [D_xysp], [D_xysp] ¹ MOVW [D_xysp], [opr16_xysp] ¹	(M:M+1) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), [D,IDX] Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 05 xb hh 11 18 02 xb xb 18 02 xb xb ff 18 02 xb xb ee ff 18 02 xb xb 18 02 xb xb ee ff	OIPRfPW OIPRfOW OIPRfPW OIPRfOPW OIPRfIOW OIPRfPIOW	NA NA NA NA NA NA	---- ----
MOVW [opr16_xysp], opr16a ¹ MOVW [opr16_xysp], oprx0_xysp ¹ MOVW [opr16_xysp], oprx9_xysp ¹ MOVW [opr16_xysp], oprx16_xysp ¹ MOVW [opr16_xysp], [D_xysp] ¹ MOVW [opr16_xysp], [opr16_xysp] ¹	(M:M+1) ⇒ M:M+1 ₂ Memory to Memory Word-Move (16-Bit), [IDX2] Source fi Addr. Mode Destination	EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 05 xb ee ff hh 11 18 02 xb ee ff xb 18 02 xb ee ff xb ff 18 02 xb ee ff xb ee ff 18 02 xb ee ff xb 18 02 xb ee ff xb ee ff	OPIPRfPW OPIPRfO- WOPIPRfP- WOPIPRfOPWOP IPRfIO- WOPIPRfPIOW	NA NA NA NA NA NA	---- ----
MUL	(A) × (B) ⇒ A:B 8 by 8 Unsigned Multiply	INH	12	O	----	---Δ
NEG opr16a NEG oprx0_xysp NEG oprx9_xysp NEG oprx16_xysp NEG [D_xysp] NEG [opr16_xysp] NEGA NEGB	0 - (M) ⇒ M equivalent to (M) + 1 ⇒ M Two's Complement Negate 0 - (A) ⇒ A equivalent to (A) + 1 ⇒ A Negate Accumulator A 0 - (B) ⇒ B equivalent to (B) + 1 ⇒ B Negate Accumulator B	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	70 hh 11 60 xb 60 xb ff 60 xb ee ff 60 xb 60 xb ee ff 40 50	rPwO rPw rPwO frPwP fIfPrPw fIPrPw O O	---- O	Δ Δ Δ Δ
NEGW opr16a NEGW oprx0_xysp NEGW oprx9_xysp NEGW oprx16_xysp NEGW [D_xysp] NEGW [opr16_xysp] NEGX NEGY	0 - (M:M+1) ⇒ M:M+1 equivalent to (M:M+1) + 1 ⇒ M:M+1 Two's Complement Negate 0 - (X) ⇒ X equivalent to (X) + 1 ⇒ X Negate Index Register X 0 - (Y) ⇒ Y equivalent to (Y) + 1 ⇒ Y Negate Index Register Y	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 70 hh 11 18 60 xb 18 60 xb ff 18 60 xb ee ff 18 60 xb 18 60 xb ee ff 18 40 18 50	ORPWO ORPW ORPWO OfRPWP OfIRPWP OfIPRPW OO OO	NA NA NA NA NA NA NA NA	---- Δ Δ Δ Δ
NOP	No Operation	INH	A7	O	----	----
ORAA #opr8i ORAA opr8a ORAA opr16a ORAA oprx0_xysp ORAA oprx9_xysp ORAA oprx16_xysp ORAA [D_xysp] ORAA [opr16_xysp]	(A) (M) ⇒ A Logical OR A with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	8A ii 9A dd BA hh 11 AA xb AA xb ff AA xb ee ff AA xb AA xb ee ff	P rPf rPO rPf rPO frPP fIfPrPf fIPrPf	---- fIfPrPf	Δ Δ 0 -
ORAB #opr8i ORAB opr8a ORAB opr16a ORAB oprx0_xysp ORAB oprx9_xysp ORAB oprx16_xysp ORAB [D_xysp] ORAB [opr16_xysp]	(B) (M) ⇒ B Logical OR B with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	CA ii DA dd FA hh 11 EA xb EA xb ff EA xb ee ff EA xb EA xb ee ff	P rPf rPO rPf rPO frPP fIfPrPf fIPrPf	---- fIPrPf	Δ Δ 0 -
ORCC #opr8i	(CCR) M ⇒ CCR Logical OR CCR with Memory	IMM	14 ii	P	↑↑↑↑	↑↑↑↑
ORX #opr16i ORX opr8a ORX opr16a ORX oprx0_xysp ORX oprx9_xysp ORX oprx16_xysp ORX [D_xysp] ORX [opr16_xysp]	(X) (M:M+1) ⇒ X Logical OR X with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 8A jj kk 18 9A dd 18 BA hh 11 18 AA xb 18 AA xb ff 18 AA xb ee ff 18 AA xb 18 AA xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIRPPf OfIPRPf	NA NA NA NA NA NA NA NA	---- Δ Δ 0 -

Notes: 1. The first operand in the source code statement specifies the source for the move.
2. The IDX destination code is listed before the source for backwards compatibility.

Table A-1. Instruction Set Summary (Sheet 15 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
ORY #opr16i ORY opr8a ORY opr16a ORY oprx0_xysp ORY oprx9_xysp ORY oprx16_xysp ORY [D,xysp] ORY [opr16,xysp]	(Y) (M:M+1) ⇒ Y Logical OR Y with Memory	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 CA jj kk 18 DA dd 18 FA hh ll 18 EA xb 18 EA xb ff 18 EA xb ee ff 18 EA xb 18 EA xb ee ff	OP ORPF ORPO ORPF ORPO OfRPP OfIFRPF OfIFRPF	NA NA NA NA NA NA NA NA	---- Δ Δ 0 -
PSHA	(SP) - 1 ⇒ SP; (A) ⇒ M _(SP) Push Accumulator A onto Stack	INH	3 6	Os	----	----
PSHB	(SP) - 1 ⇒ SP; (B) ⇒ M _(SP) Push Accumulator B onto Stack	INH	3 7	Os	----	----
PSHC	(SP) - 1 ⇒ SP; (CCR) ⇒ M _(SP) Push CCR onto Stack	INH	3 9	Os	----	----
PSHCW	(SP) - 2 ⇒ SP; (CCR _H :CCR _L) ⇒ M _(SP) :M _(SP+1) Push CCR onto Stack	INH	18 39	OOS	NA	----
PSHD	(SP) - 2 ⇒ SP; (A:B) ⇒ M _(SP) :M _(SP+1) Push D Accumulator onto Stack	INH	3 B	OS	----	----
PSHX	(SP) - 2 ⇒ SP; (X _H :X _L) ⇒ M _(SP) :M _(SP+1) Push Index Register X onto Stack	INH	3 4	OS	----	----
PSHY	(SP) - 2 ⇒ SP; (Y _H :Y _L) ⇒ M _(SP) :M _(SP+1) Push Index Register Y onto Stack	INH	3 5	OS	----	----
PULA	(M _(SP)) ⇒ A; (SP) + 1 ⇒ SP Pull Accumulator A from Stack	INH	3 2	ufo	----	----
PULB	(M _(SP)) ⇒ B; (SP) + 1 ⇒ SP Pull Accumulator B from Stack	INH	3 3	ufo	----	----
PULC	(M _(SP)) ⇒ CCR; (SP) + 1 ⇒ SP Pull CCR from Stack	INH	3 8	ufo	Δ ↓ Δ Δ	Δ Δ Δ Δ
PULCW	(M _(SP) :M _(SP+1)) ⇒ CCR _H :CCR _L ; (SP) + 2 ⇒ SP Pull CCR from Stack	INH	18 38	OUFO	NA	Δ ↓ Δ Δ
PULD	(M _(SP) :M _(SP+1)) ⇒ A:B; (SP) + 2 ⇒ SP Pull D from Stack	INH	3 A	Ufo	----	----
PULX	(M _(SP) :M _(SP+1)) ⇒ X _H :X _L ; (SP) + 2 ⇒ SP Pull Index Register X from Stack	INH	3 0	Ufo	----	----
PULY	(M _(SP) :M _(SP+1)) ⇒ Y _H :Y _L ; (SP) + 2 ⇒ SP Pull Index Register Y from Stack	INH	3 1	Ufo	----	----
REV (Only for CPU12V0 and CPU12XV0)	MIN-MAX rule evaluation Find smallest rule input (MIN). Store to rule outputs unless fuzzy output is already larger (MAX). For rule weights see REVW. Each rule input is an 8-bit offset from the base address in Y. Each rule output is an 8-bit offset from the base address in Y. \$FE separates rule inputs from rule outputs. \$FF terminates the rule list. REV may be interrupted.	Special	18 3A	Orf(t, tx)O (exit + re-entry replaces comma above if interrupted) ff + Orf(t,	--? -	?? Δ ?
REVW (Only for CPU12V0 and CPU12XV0)	MIN-MAX rule evaluation Find smallest rule input (MIN). Store to rule outputs unless fuzzy output is already larger (MAX). Rule weights supported, optional. Each rule input is the 16-bit address of a fuzzy input. Each rule output is the 16-bit address of a fuzzy output. The value \$FFFE separates rule inputs from rule outputs. \$FFFF terminates the rule list. REVW may be interrupted.	Special	18 3B	ORf(t, Tx)O (loop to read weight if enabled) (r, RfRf) (exit + re-entry replaces comma above if interrupted) ffff + ORf(t,	--? -	?? Δ !

Table A-1. Instruction Set Summary (Sheet 16 of 20)

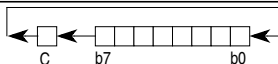

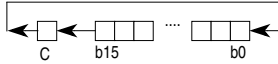

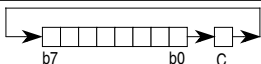
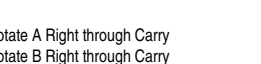
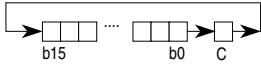
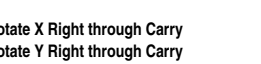
Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C	
ROL opr16a ROL oprx0_xysp ROL oprx9_xysp ROL oprx16_xysp ROL [D,xysp] ROL [oprx16,xysp] ROLA ROLB	 Rotate Memory Left through Carry  Rotate A Left through Carry Rotate B Left through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	75 hh 11 65 xb 65 xb ff 65 xb ee ff 65 xb 65 xb ee ff 45 55	rPwO rPw rPwO frPwP fIfrrPw fIPrPw O O	----	Δ Δ Δ Δ	
ROLW opr16a ROLW oprx0_xysp ROLW oprx9_xysp ROLW oprx16_xysp ROLW [D,xysp] ROLW [oprx16,xysp] ROLX ROLY	 Rotate Memory Left through Carry  Rotate X Left through Carry Rotate Y Left through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 75 hh 11 18 65 xb 18 65 xb ff 18 65 xb ee ff 18 65 xb 18 65 xb ee ff 18 45 18 55	ORPWO ORPW ORPWO OfRPWP OfIfRPW fOIPRPW OO OO	NA NA NA NA NA NA NA NA	----	Δ Δ Δ Δ
ROR opr16a ROR oprx0_xysp ROR oprx9_xysp ROR oprx16_xysp ROR [D,xysp] ROR [oprx16,xysp] RORA RORB	 Rotate Memory Right through Carry  Rotate A Right through Carry Rotate B Right through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	76 hh 11 66 xb 66 xb ff 66 xb ee ff 66 xb 66 xb ee ff 46 56	rPwO rPw rPwO frPwP fIfrrPw fIPrPw O O	----	Δ Δ Δ Δ	
RORW opr16a RORW oprx0_xysp RORW oprx9_xysp RORW oprx16_xysp RORW [D,xysp] RORW [oprx16,xysp] RORX RORY	 Rotate Memory Right through Carry  Rotate X Right through Carry Rotate Y Right through Carry	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 76 hh 11 18 66 xb 18 66 xb ff 18 66 xb ee ff 18 66 xb 18 66 xb ee ff 18 46 18 56	ORPWO ORPW ORPWO OfRPWP OfIfRPW fOIPRPW OO OO	NA NA NA NA NA NA NA NA	----	Δ Δ Δ Δ
RTC	$(M_{(SP)} \Rightarrow PPAGE; (SP) + 1 \Rightarrow SP;$ $(M_{(SP)}:M_{(SP+1)} \Rightarrow PC_H:PC_L;$ $(SP) + 2 \Rightarrow SP$ Return from Call	INH	0A	uUnfPPP	----	----	
RTI	$(M_{(SP)}:M_{(SP+1)} \Rightarrow CCRW; (SP) + 2 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)} \Rightarrow B:A; (SP) + 2 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)} \Rightarrow X_H:X_L; (SP) + 4 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)} \Rightarrow PC_H:PC_L; (SP) - 2 \Rightarrow SP$ $(M_{(SP)}:M_{(SP+1)} \Rightarrow Y_H:Y_L; (SP) + 4 \Rightarrow SP$ Return from Interrupt	INH	0B	UUUUUPPP uUUUUUPPP (with interrupt pending) UUUUUVfPPP uUUUUUVfPPP	Δ fl Δ Δ	Δ Δ Δ Δ	
RTS	$(M_{(SP)}:M_{(SP+1)} \Rightarrow PC_H:PC_L;$ $(SP) + 2 \Rightarrow SP$ Return from Subroutine	INH	3D	UfPPP	----	----	
SBA	$(A) - (B) \Rightarrow A$ Subtract B from A	INH	18 16	OO	----	Δ Δ Δ Δ	
SBCA #opr8i SBCA opr8a SBCA opr16a SBCA oprx0_xysp SBCA oprx9_xysp SBCA oprx16_xysp SBCA [D,xysp] SBCA [oprx16,xysp]	$(A) - (M) - C \Rightarrow A$ Subtract with Borrow from A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	82 ii 92 dd B2 hh 11 A2 xb A2 xb ff A2 xb ee ff A2 xb A2 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ Δ Δ	
SBCB #opr8i SBCB opr8a SBCB opr16a SBCB oprx0_xysp SBCB oprx9_xysp SBCB oprx16_xysp SBCB [D,xysp] SBCB [oprx16,xysp]	$(B) - (M) - C \Rightarrow B$ Subtract with Borrow from B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C2 ii D2 dd F2 hh 11 E2 xb E2 xb ff E2 xb ee ff E2 xb E2 xb ee ff	P rPf rPO rPf rPO frPP fIfrrPf fIPrPf	----	Δ Δ Δ Δ	

Table A-1. Instruction Set Summary (Sheet 17 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail		S X H I	N Z V C
CPU12X	CPU12						
SBED #opr16i SBED opr8a SBED opr16a SBED oprx0_xysp SBED oprx9_xysp SBED oprx16_xysp SBED [D,xysp] SBED [opr16,xysp]	(D) – (M:M+1) – C ⇒ D Subtract with Borrow from D	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 83 jj kk 18 93 dd 18 B3 hh 11 18 A3 xb 18 A3 xb ff 18 A3 xb ee ff 18 A3 xb 18 A3 xb ee ff	OPO ORP _f ORPO ORP _f ORPO OfRPP OfIRP _f OfIRP _f	NA NA NA NA NA NA NA NA	----	Δ Δ Δ Δ
SBEX #opr16i SBEX opr8a SBEX opr16a SBEX oprx0_xysp SBEX oprx9_xysp SBEX oprx16_xysp SBEX [D,xysp] SBEX [opr16,xysp]	(X) – (M:M+1) – C ⇒ X Subtract with Borrow from X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 82 jj kk 18 92 dd 18 B2 hh 11 18 A2 xb 18 A2 xb ff 18 A2 xb ee ff 18 A2 xb 18 A2 xb ee ff	OPO ORP _f ORPO ORP _f ORPO OfRPP OfIRP _f OfIRP _f	NA NA NA NA NA NA NA NA	----	Δ Δ Δ Δ
SBEY #opr16i SBEY opr8a SBEY opr16a SBEY oprx0_xysp SBEY oprx9_xysp SBEY oprx16_xysp SBEY [D,xysp] SBEY [opr16,xysp]	(Y) – (M:M+1) – C ⇒ Y Subtract with Borrow from Y	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C2 jj kk 18 D2 dd 18 F2 hh 11 18 E2 xb 18 E2 xb ff 18 E2 xb ee ff 18 E2 xb 18 E2 xb ee ff	OPO ORP _f ORPO ORP _f ORPO OfRPP OfIRP _f OfIRP _f	NA NA NA NA NA NA NA NA	----	Δ Δ Δ Δ
SEC	1 ⇒ C Translates to ORCC #S01	IMM	14 01	P		----	---1
SEI	1 ⇒ I; (inhibit I interrupts) Translates to ORCC #S10	IMM	14 10	P		---1	----
SEV	1 ⇒ V Translates to ORCC #S02	IMM	14 02	P		----	--1-
SEX abc,dxys	\$00:(r1) ⇒ r2 if r1, bit 7 is 0 or \$FF:(r1) ⇒ r2 if r1, bit 7 is 1 Sign Extend 8-bit r1 to 16-bit r2 r1 may be A, B, or CCR r2 may be D, X, Y, or SP Alternate mnemonic for TFR r1, r2	INH	B7 eb	P		----	----
SEX d,xy	\$0000 ⇒ r2 if D, bit 15 is 0 or \$FFFF ⇒ r2 if D, bit 15 is 1 Sign Extend 16-bit D to 32-bit register pair (r2:D) r2 may be X or Y	INH	B7 eb	P	NA	----	----
STAA opr8a STAA opr16a STAA oprx0_xysp STAA oprx9_xysp STAA oprx16_xysp STAA [D,xysp] STAA [opr16,xysp]	(A) ⇒ M Store Accumulator A to Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5A dd 7A hh 11 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_xysp STAB oprx9_xysp STAB oprx16_xysp STAB [D,xysp] STAB [opr16,xysp]	(B) ⇒ M Store Accumulator B to Memory	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5B dd 7B hh 11 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_xysp STD oprx9_xysp STD oprx16_xysp STD [D,xysp] STD [opr16,xysp]	(A) ⇒ M, (B) ⇒ M+1 Store Double Accumulator	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5C dd 7C hh 11 6C xb 6C xb ff 6C xb ee ff 6C xb 6C xb ee ff	PW PW0 PW PW0 PW0 PIfW PIPW		----	Δ Δ 0 -

Table A-1. Instruction Set Summary (Sheet 18 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
STOP	$(SP) - 2 \Rightarrow SP;$ $RTN_H; RTN_L \Rightarrow M_{(SP)}; M_{(SP+1)};$ $(SP) - 2 \Rightarrow SP; (Y_H; Y_L) \Rightarrow M_{(SP)}; M_{(SP+1)};$ $(SP) - 2 \Rightarrow SP; (X_H; X_L) \Rightarrow M_{(SP)}; M_{(SP+1)};$ $(SP) - 2 \Rightarrow SP; (B; A) \Rightarrow M_{(SP)}; M_{(SP+1)};$ $(SP) - 2 \Rightarrow SP; (CCR_W) \Rightarrow M_{(SP)}; M_{(SP+1)};$ STOP All Clocks Registers stacked to allow quicker recovery by interrupt. If S control bit = 1, the STOP instruction is disabled and acts like a two-cycle NOP.	INH	18 3E	(entering STOP) 00SSSSSff 00SSSSSff (exiting STOP) fVfPPP (continue) ff (if STOP disabled) 00	----	----
STS opr8a STS opr16a STS oprx0_xysp STS oprx9_xysp STS oprx16_xysp STS [D,xysp] STS [oprx16,xysp]	$(SP_H; SP_L) \Rightarrow M:M+1$ Store Stack Pointer	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5F dd 7F hh 11 6F xb 6F xb ff 6F xb ee ff 6F xb 6F xb ee ff	PW PWO PW PWO PWP PIfW PIPW	----	$\Delta \Delta 0 -$
STX opr8a STX opr16a STX oprx0_xysp STX oprx9_xysp STX oprx16_xysp STX [D,xysp] STX [oprx16,xysp]	$(X_H; X_L) \Rightarrow M:M+1$ Store Index Register X	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5E dd 7E hh 11 6E xb 6E xb ff 6E xb ee ff 6E xb 6E xb ee ff	PW PWO PW PWO PWP PIfW PIPW	----	$\Delta \Delta 0 -$
STY opr8a STY opr16a STY oprx0_xysp STY oprx9_xysp STY oprx16_xysp STY [D,xysp] STY [oprx16,xysp]	$(Y_H; Y_L) \Rightarrow M:M+1$ Store Index Register Y	DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	5D dd 7D hh 11 6D xb 6D xb ff 6D xb ee ff 6D xb 6D xb ee ff	PW PWO PW PWO PWP PIfW PIPW	----	$\Delta \Delta 0 -$
SUBA #opr8i SUBA opr8a SUBA opr16a SUBA oprx0_xysp SUBA oprx9_xysp SUBA oprx16_xysp SUBA [D,xysp] SUBA [oprx16,xysp]	$(A) - (M) \Rightarrow A$ Subtract Memory from Accumulator A	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	80 ii 90 dd B0 hh 11 A0 xb A0 xb ff A0 xb ee ff A0 xb A0 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	$\Delta \Delta \Delta \Delta$
SUBB #opr8i SUBB opr8a SUBB opr16a SUBB oprx0_xysp SUBB oprx9_xysp SUBB oprx16_xysp SUBB [D,xysp] SUBB [oprx16,xysp]	$(B) - (M) \Rightarrow B$ Subtract Memory from Accumulator B	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	C0 ii D0 dd F0 hh 11 E0 xb E0 xb ff E0 xb ee ff E0 xb E0 xb ee ff	P rPf rPO rPf rPO frPP fIfrPf fIPrPf	----	$\Delta \Delta \Delta \Delta$
SUBD #opr16i SUBD opr8a SUBD opr16a SUBD oprx0_xysp SUBD oprx9_xysp SUBD oprx16_xysp SUBD [D,xysp] SUBD [oprx16,xysp]	$(D) - (M:M+1) \Rightarrow D$ Subtract Memory from D (A:B)	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	83 jj kk 93 dd B3 hh 11 A3 xb A3 xb ff A3 xb ee ff A3 xb A3 xb ee ff	PO RPf RPO RPf RPO frPP fIfRPf fIPRPf	----	$\Delta \Delta \Delta \Delta$
SUBX #opr16i SUBX opr8a SUBX opr16a SUBX oprx0_xysp SUBX oprx9_xysp SUBX oprx16_xysp SUBX [D,xysp] SUBX [oprx16,xysp]	$(X) - (M:M+1) \Rightarrow X$ Subtract Memory from X	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 80 jj kk 18 90 dd B0 hh 11 A0 xb A0 xb ff A0 xb ee ff A0 xb A0 xb ee ff	OPO ORPf ORPO ORPf ORPO OfRPP OfIfRPf OfIPRPf	NA NA NA NA NA NA NA NA	---- $\Delta \Delta \Delta \Delta$

Table A-1. Instruction Set Summary (Sheet 19 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
SUBY #opr16i SUBY opr8a SUBY opr16a SUBY oprx0_xysp SUBY oprx9,xysp SUBY oprx16,xysp SUBY [D,xysp] SUBY [oprx16,xysp]	$(Y) - (M:M+1) \Rightarrow Y$ Subtract Memory from Y	IMM DIR EXT IDX IDX1 IDX2 [D,IDX] [IDX2]	18 C0 jj kk 18 D0 dd 18 F0 hh ll 18 E0 xb 18 E0 xb ff 18 E0 xb ee ff 18 E0 xb 18 E0 xb ee ff	OPO ORPF ORPO ORPF ORPO OfRPP OfIRPF OfIRPF	NA NA NA NA NA NA NA NA	---- $\Delta \Delta \Delta \Delta$
SWI	$(SP) - 2 \Rightarrow SP$; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(B:A) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(CCRW) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $1 \Rightarrow I$; (SWI Vector) $\Rightarrow PC$ Software Interrupt	INH	3F	VSPSSPSSP* VSPSSPSsP* (for Reset) VFPPP	---1 11-1	---- ----
*The CPU12 Family also uses the SWI microcode sequence for hardware interrupts and unimplemented opcode traps. Reset uses the VFPPP variation of this sequence.						
SYS*	$(SP) - 2 \Rightarrow SP$; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(B:A) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(CCRW) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $1 \Rightarrow I$; (SYS Vector) $\Rightarrow PC$ System Call Instruction	INH	18 A7	OVSPSSPSSP NA	---1	----
*The SYS instruction is available only on CPU12XV1 and CPU12XV2						
TAB	$(A) \Rightarrow B$ Transfer A to B	INH	18 0E	OO	----	$\Delta \Delta 0 -$
TAP	$(A) \Rightarrow CCR$ Translates to TFR A, CCR	INH	B7 02	P	$\Delta \Downarrow \Delta \Delta$	$\Delta \Delta \Delta \Delta$
TBA	$(B) \Rightarrow A$ Transfer B to A	INH	18 0F	OO	----	$\Delta \Delta 0 -$
TBEQ abdxys,rel9	If (cntr) = 0, then Branch; else Continue to next instruction Test Counter and Branch if Zero (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
TBL oprx0_xysp	$(M) + [(B) \times ((M+1) - (M))] \Rightarrow A$ 8-Bit Table Lookup and Interpolate Initialize B, and index before TBL. <ea> points at first 8-bit table entry (M) and B is fractional part of lookup value. (no indirect addressing modes or extensions allowed)	IDX	18 3D xb	ORffFP	----	$\Delta \Delta - \Delta$
TBNE abdxys,rel9	If (cntr) not = 0, then Branch; else Continue to next instruction Test Counter and Branch if Not Zero (cntr = A, B, D, X, Y, or SP)	REL (9-bit)	04 1b rr	PPP (branch) PPO (no branch)	----	----
TFR abcdxys,abcdxys	$(r1) \Rightarrow r2$ or $\$00:(r1) \Rightarrow r2$ or $(r1[7:0]) \Rightarrow r2$ Transfer Register to Register r1 and r2 may be A, B, CCR, D, X, Y, or SP	INH	B7 eb	P	---- or $\Delta \Downarrow \Delta \Delta$	---- $\Delta \Delta \Delta \Delta$
TPA	$(CCR) \Rightarrow A$ Translates to TFR CCR, A	INH	B7 20	P	----	----
TRAP trapnum	$(SP) - 2 \Rightarrow SP$; $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(Y_H:Y_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(X_H:X_L) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(B:A) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $(SP) - 2 \Rightarrow SP$; $(CCRW) \Rightarrow M_{(SP)}:M_{(SP+1)}$; $1 \Rightarrow I$; (TRAP Vector) $\Rightarrow PC$ Unimplemented opcode trap	INH	18 tn tn = \$30-\$39 or \$40-\$FF	OVSPSSPSSP OVSPSSPSsP	---1	----

Table A-1. Instruction Set Summary (Sheet 20 of 20)

Source Form	Operation	Addr. Mode	Machine Coding (hex)	Access Detail CPU12X CPU12	S X H I	N Z V C
TST opr16a TST oprx0_xysp TST oprx9_xysp TST oprx16_xysp TST [D,xysp] TST [oprx16,xysp] TSTA TSTB	(M) – 0 Test Memory for Zero or Minus (A) – 0Test A for Zero or Minus (B) – 0Test B for Zero or Minus	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	F7 hh 11 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
TSTW opr16a TSTW oprx0_xysp TSTW oprx9_xysp TSTW oprx16_xysp TSTW [D,xysp] TSTW [oprx16,xysp] TSTX TSTY	(M:M+1) – 0 Test Memory for Zero or Minus (X) – 0Test X for Zero or Minus (Y) – 0Test Y for Zero or Minus	EXT IDX IDX1 IDX2 [D,IDX] [IDX2] INH INH	18 F7 hh 11 18 E7 xb 18 E7 xb ff 18 E7 xb ee ff 18 E7 xb 18 E7 xb ee ff 18 97 18 D7	ORPO ORPf ORPO OfRPP OfIfRPF OfIPRPF OO OO	NA NA NA NA NA NA NA NA	---- Δ Δ 0 0
TSX	(SP) ⇒ X Translates to TFR SP,X	INH	B7 75	P	----	----
TSY	(SP) ⇒ Y Translates to TFR SP,Y	INH	B7 76	P	----	----
TXS	(X) ⇒ SP Translates to TFR X,SP	INH	B7 57	P	----	----
TYS	(Y) ⇒ SP Translates to TFR Y,SP	INH	B7 67	P	----	----
WAI	(SP) – 2 ⇒ SP; RTN _H :RTN _L ⇒ M _(SP) :M _(SP+1) ; (SP) – 2 ⇒ SP; (Y _H :Y _L) ⇒ M _(SP) :M _(SP+1) ; (SP) – 2 ⇒ SP; (X _H :X _L) ⇒ M _(SP) :M _(SP+1) ; (SP) – 2 ⇒ SP; (B:A) ⇒ M _(SP) :M _(SP+1) ; (SP) – 2 ⇒ SP; (CCR _W) ⇒ M _(SP) :M _(SP+1) ; WAIT for interrupt	INH	3E	OSSSSSf (after interrupt) fVfPPP	---- or ---1 or -1-1	---- or ---- ----
WAV (Only for CPU12V0 and CPU12XV0)	$\sum_{i=1}^B S_i F_i \text{ fi } Y:D \text{ and } \sum_{i=1}^B F_i \text{ fi } X$ Calculate Sum of Products and Sum of Weights for Weighted Average Calculation Initialize B, X, and Y before WAV. B specifies number of elements. X points at first element in S _i list. Y points at first element in F _i list. All S _i and F _i elements are 8-bits. If interrupted, six extra bytes of stack used for intermediate values	Special	18 3C	Of(frr,ffff)O (add if interrupt) SSS + UUUrr,	--?- ?Δ??	?
wavr (Only for CPU12V0 and CPU12XV0) pseudo-instruction	see WAV Resume executing an interrupted WAV instruction (recover intermediate results from stack rather than initializing them to zero)	Special	3C	UUUrr,ffff (frr,ffff)O (exit + re-entry replaces comma above if interrupted) SSS + UUUrr,	--?- ?Δ??	?
XGDX	(D) ⇔ (X) Translates to EXG D, X	INH	B7 C5	P	----	----
XGDY	(D) ⇔ (Y) Translates to EXG D, Y	INH	B7 C6	P	----	----

Table A-2. Opcode Map (Sheet 1 of 3) — CPU12 Family Page 1 Opcodes

00 15 IH BGND 1 1	10 1 ANDCC 2 2	20 3 BRA 2 2	30 3 PULX 1 1	40 1 NEGA 1 1	50 1 NEGB 1 1	60 3-6 NEG 2-4 3	70 4 NEG 3 3	80 1 SUBA 2 2	90 3 SUBA 2 2	A0 3-6 SUBA 2-4 2	B0 3 SUBA 3 3	C0 1 SUBB 2 2	D0 3 SUBB 2 2	E0 3-6 SUBB 2-4 2	F0 3 SUBB 3 3
01 5 IH MEM 1 1	11 11 EDIV 1 1	21 1 BRN 2 2	31 3 PULY 1 1	41 1 COMA 1 1	51 1 COMB 1 1	61 3-6 COM 2-4 3	71 4 COM 3 3	81 1 CMPA 2 2	91 3 CMPA 2 2	A1 3-6 CMPA 2-4 2	B1 3 CMPA 3 3	C1 1 CMPB 2 2	D1 3 CMPB 2 2	E1 3-6 CMPB 2-4 2	F1 3 CMPB 3 3
02 1 IH INY 1 1	12 1 MUL 1 1	22 3/1 BHI 2 2	32 3 PULA 1 1	42 1 INCA 1 1	52 1 INCB 1 1	62 3-6 INC 2-4 3	72 4 INC 3 3	82 1 SBCA 2 2	92 3 SBCA 2 2	A2 3-6 SBCA 2-4 2	B2 3 SBCA 3 3	C2 1 SBCB 2 2	D2 3 SBCB 2 2	E2 3-6 SBCB 2-4 2	F2 3 SBCB 3 3
03 1 IH DEY 1 1	13 1 EMUL 1 1	23 3/1 BLS 2 2	33 3 PULB 1 1	43 1 DECA 1 1	53 1 DECB 1 1	63 3-6 DEC 2-4 3	73 4 DEC 3 3	83 2 SUBD 3 3	93 3 SUBD 2 2	A3 3-6 SUBD 2-4 2	B3 3 SUBD 3 3	C3 2 ADDD 2 2	D3 3 ADDD 2 2	E3 3-6 ADDD 2-4 2	F3 3 ADDD 3 3
04 3 loop* RL 3	14 1 ORCC 2 2	24 3/1 BCC 2 2	34 2 PSHX 1 1	44 1 LSRA 1 1	54 1 LSRB 1 1	64 3-6 LSR 2-4 3	74 4 LSR 3 3	84 1 ANDA 2 2	94 3 ANDA 2 2	A4 3-6 ANDA 2-4 2	B4 3 ANDA 3 3	C4 1 ANDB 2 2	D4 3 ANDB 2 2	E4 3-6 ANDB 2-4 2	F4 3 ANDB 3 3
05 3-6 ID JMP 2-4 3	15 4-7 JSR 2-4 2	25 3/1 BCS 2 2	35 2 PSHY 1 1	45 1 ROLA 1 1	55 1 ROLB 1 1	65 3-6 ROL 2-4 3	75 4 ROL 3 3	85 1 BITA 2 2	95 3 BITA 2 2	A5 3-6 BITA 2-4 2	B5 3 BITA 3 3	C5 1 BITB 2 2	D5 3 BITB 2 2	E5 3-6 BITB 2-4 2	F5 3 BITB 3 3
06 3 EX JMP 3 3	16 4 JSR 3 3	26 3/1 BNE 2 2	36 2 PSHA 1 1	46 1 RORA 1 1	56 1 RORB 1 1	66 3-6 ROR 2-4 3	76 4 ROR 3 3	86 1 LDAA 2 2	96 3 LDAA 2 2	A6 3-6 LDAA 2-4 2	B6 3 LDAA 3 3	C6 1 LDAB 2 2	D6 3 LDAB 2 2	E6 3-6 LDAB 2-4 2	F6 3 LDAB 3 3
07 4 RL BSR 2 2	17 4 JSR 2 2	27 3/1 BEQ 2 2	37 2 PSHB 1 1	47 1 ASRA 1 1	57 1 ASRB 1 1	67 3-6 ASR 2-4 3	77 4 ASR 3 3	87 1 CLRA 1 1	97 1 TSTA 1 1	A7 1 NOP 1 1	B7 1 TFR/EXG 2 2	C7 1 CLRB 1 1	D7 1 TSTB 1 1	E7 3-6 TST 2-4 2	F7 3 TST 3 3
08 1 IH INX 1 1	18 - Page 2 - -	28 3/1 BVC 2 2	38 3 PULC 1 1	48 1 ASLA 1 1	58 1 ASLB 1 1	68 3-6 ASL 2-4 3	78 4 ASL 3 3	88 1 EORA 2 2	98 3 EORA 2 2	A8 3-6 EORA 2-4 2	B8 3 EORA 3 3	C8 1 EORB 2 2	D8 3 EORB 2 2	E8 3-6 EORB 2-4 2	F8 3 EORB 3 3
09 1 IH DEX 1 1	19 2 LEAY 2-4 2	29 3/1 BVS 2 2	39 2 PSHC 1 1	49 1 LSRD 1 1	59 1 ASLD 1 1	69 2-4 CLR 2-4 3	79 3 CLR 3 3	89 1 ADCA 2 2	99 3 ADCA 2 2	A9 3-6 ADCA 2-4 2	B9 3 ADCA 3 3	C9 1 ADCB 2 2	D9 3 ADCB 2 2	E9 3-6 ADCB 2-4 2	F9 3 ADCB 3 3
0A 7 IH RTC 1 1	1A 2 LEAX 2-4 2	2A 3/1 BPL 2 2	3A 3 PULD 1 1	4A 7 CALL 4 4	5A 2 STAA 2 2	6A 2-4 STAA 2-4 3	7A 3 STAA 3 3	8A 1 ORAA 2 2	9A 3 ORAA 2 2	AA 3-6 ORAA 2-4 2	BA 3 ORAA 3 3	CA 1 ORAB 2 2	DA 3 ORAB 2 2	EA 3-6 ORAB 2-4 2	FA 3 ORAB 3 3
0B 8 IH RTI 1 1	1B 2 LEAS 2-4 2	2B 3/1 BMI 2 2	3B 2 PSHD 1 1	4B 7-10 CALL 2-5 2	5B 2 STAB 2 2	6B 2-4 STAB 2-4 3	7B 3 STAB 3 3	8B 1 ADDA 2 2	9B 3 ADDA 2 2	AB 3-6 ADDA 2-4 2	BB 3 ADDA 3 3	CB 1 ADDB 2 2	DB 3 ADDB 2 2	EB 3-6 ADDB 2-4 2	FB 3 ADDB 3 3
0C 4-6 ID BSET 3-5 3	1C 4 BSET 4 4	2C 3/1 BGE 2 2	3C +5 wavr SP 1	4C 4 BSET 3 3	5C 2 STD 2 2	6C 2-4 STD 2-4 3	7C 3 STD 3 3	8C 2 CPD 3 3	9C 3 CPD 2 2	AC 3-6 CPD 2-4 2	BC 3 CPD 3 3	CC 2 LDD 2 2	DC 3 LDD 2 2	EC 3-6 LDD 2-4 2	FC 3 LDD 3 3
0D 4-6 ID BCLR 3-5 3	1D 4 BCLR 4 4	2D 3/1 BLT 2 2	3D 5 RTS 1 1	4D 4 BCLR 3 3	5D 2 STY 2 2	6D 2-4 STY 2-4 3	7D 3 STY 3 3	8D 2 CPY 3 3	9D 3 CPY 2 2	AD 3-6 CPY 2-4 2	BD 3 CPY 3 3	CD 2 LDY 2 2	DD 3 LDY 2 2	ED 3-6 LDY 2-4 2	FD 3 LDY 3 3
0E 4-6 ID BRSET 4-6 3	1E 5 BRSET 5 5	2E 3/1 BGT 2 2	3E 7 WAI 1 1	4E 4 BRSET 4 4	5E 2 STX 2 2	6E 2-4 STX 2-4 3	7E 3 STX 3 3	8E 2 CPX 3 3	9E 3 CPX 2 2	AE 3-6 CPX 2-4 2	BE 3 CPX 3 3	CE 2 LDX 2 2	DE 3 LDX 2 2	EE 3-6 LDX 2-4 2	FE 3 LDX 3 3
0F 4-6 ID BRCLR 4-6 3	1F 5 BRCLR 5 5	2F 3/1 BLE 2 2	3F 9 SWI 1 1	4F 4 BRCLR 4 4	5F 2 STS 2 2	6F 2-4 STS 2-4 3	7F 3 STS 3 3	8F 2 CPS 3 3	9F 3 CPS 2 2	AF 3-6 CPS 2-4 2	BF 3 CPS 3 3	CF 2 LDS 2 2	DF 3 LDS 2 2	EF 3-6 LDS 2-4 2	FF 3 LDS 3 3

Key to Table A-2



* The opcode \$04 (on sheet 1 of 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.
Page 2 When the CPU encounters a page 2 opcode (\$18 on page 1 of the opcode map), it treats the next byte of object code as a page 2 instruction opcode.
† Refer to instruction summary for more information.
§ EMUL requires 3 cycles for CPU12.

Table A-2. Opcode Map (Sheet 2 of 3) — CPU12 Page 2 Opcodes

00	4	10	12	20	4	30	10	40	10	50	10	60	10	70	10	80	10	90	10	A0	10	B0	10	C0	10	D0	10	E0	10	F0	10
MOVW		IDIV		LBRA		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IM-ID	5	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
01	5	11	12	21	3	31	10	41	10	51	10	61	10	71	10	81	10	91	10	A1	10	B1	10	C1	10	D1	10	E1	10	F1	10
MOVW		FDIV		LBRN		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
EX-ID	5	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
02	5	12	13	22	4/3	32	10	42	10	52	10	62	10	72	10	82	10	92	10	A2	10	B2	10	C2	10	D2	10	E2	10	F2	10
MOVW		EMACS		LBHI		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
ID-ID	4	SP	4	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
03	5	13	3	23	4/3	33	10	43	10	53	10	63	10	73	10	83	10	93	10	A3	10	B3	10	C3	10	D3	10	E3	10	F3	10
MOVW		EMULS		LBSL		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IM-EX	6	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
04	6	14	12	24	4/3	34	10	44	10	54	10	64	10	74	10	84	10	94	10	A4	10	B4	10	C4	10	D4	10	E4	10	F4	10
MOVW		EDIVS		LBCC		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
EX-EX	6	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
05	5	15	12	25	4/3	35	10	45	10	55	10	65	10	75	10	85	10	95	10	A5	10	B5	10	C5	10	D5	10	E5	10	F5	10
MOVW		IDIVS		LBSC		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
ID-EX	5	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
06	2	16	2	26	4/3	36	10	46	10	56	10	66	10	76	10	86	10	96	10	A6	10	B6	10	C6	10	D6	10	E6	10	F6	10
ABA		SBA		LBNE		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IH	2	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
07	3	17	2	27	4/3	37	10	47	10	57	10	67	10	77	10	87	10	97	10	A7	10	B7	10	C7	10	D7	10	E7	10	F7	10
DAA		CBA		LBEQ		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IH	2	IH	2	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
08	5	18	4-7	28	4/3	38	10	48	10	58	10	68	10	78	10	88	10	98	10	A8	10	B8	10	C8	10	D8	10	E8	10	F8	10
MOVB		MAXA		LBVC		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IM-ID	4	ID	3-5	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
09	5	19	4-7	29	4/3	39	10	49	10	59	10	69	10	79	10	89	10	99	10	A9	10	B9	10	C9	10	D9	10	E9	10	F9	10
MOVB		MINA		LBVS		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
EX-ID	5	ID	3-5	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0A	5	1A	4-7	2A	4/3	3A	†3n	4A	10	5A	10	6A	10	7A	10	8A	10	9A	10	AA	10	BA	10	CA	10	DA	10	EA	10	FA	10
MOVB		EMAXD		LBPL		REV		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
ID-ID	4	ID	3-5	RL	4	SP	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0B	4	1B	4-7	2B	4/3	3B	†5n/3n	4B	10	5B	10	6B	10	7B	10	8B	10	9B	10	AB	10	BB	10	CB	10	DB	10	EB	10	FB	10
MOVB		EMIND		LBMI		REVV		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IM-EX	5	ID	3-5	RL	4	SP	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0C	6	1C	4-7	2C	4/3	3C	†7B	4C	10	5C	10	6C	10	7C	10	8C	10	9C	10	AC	10	BC	10	CC	10	DC	10	EC	10	FC	10
MOVB		MAXM		LBGE		WAV		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
EX-EX	6	ID	3-5	RL	4	SP	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0D	5	1	D4-7	2D	4/3	3D	6	4D	10	5D	10	6D	10	7D	10	8D	10	9D	10	AD	10	BD	10	CD	10	DD	10	ED	10	FD	10
MOVB		MINM		LBTL		TBL		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
ID-EX	5	ID	3-5	RL	4	ID	3	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0E	2	1E	4-7	2E	4/3	3E	8	4E	10	5E	10	6E	10	7E	10	8E	10	9E	10	AE	10	BE	10	CE	10	DE	10	EE	10	FE	10
TAB		EMAXM		LBGT		STOP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IH	2	ID	3-5	RL	4	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2
0F	2	1F	4-7	2F	4/3	3F	10	4F	10	5F	10	6F	10	7F	10	8F	10	9F	10	AF	10	BF	10	CF	10	DF	10	EF	10	FF	10
TBA		EMINM		LBLE		ETBL		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP		TRAP	
IH	2	ID	3-5	RL	4	ID	3	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2	IH	2

* The opcode \$04 (on sheet 1 of 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

† Refer to instruction summary for more information.

Page 2 When the CPU12 encounters a page 2 opcode (\$18 on page 1 of the opcode map), it treats the next byte of object code as a page 2 instruction opcode.

Table A-2. Opcode Map (Sheet 3 of 3) — CPU12X Page 2 Opcodes

00 4-6 MOVW IM-ID 5	10 12 IDIV IH 2	20 4 LBRA RL 4	30 10 TRAP IH 2	40 2 NEGX IH 2	50 2 NEGY IH 2	60 4-7 NEGW ID 3-5	70 5 NEGW EX 4	80 3 SUBX IM 4	90 4 SUBX DI 3	A0 4-7 SUBX ID 3-5	B0 4 SUBX EX 4	C0 3 SUBY IM 4	D0 4 SUBY DI 3	E0 4-7 SUBY ID 3-5	F0 4 SUBY EX 4
01 5-7 MOVW EX-ID 5	11 12 FDIV IH 2	21 3 LBRL RL 4	31 10 TRAP IH 2	41 2 COMX IH 2	51 2 COMY IH 2	61 4-7 COMW ID 3-5	71 5 COMW EX 4	81 10 TRAP IH 2	91 10 TRAP IH 2	A1 10 TRAP IH 2	B1 10 TRAP IH 2	C1 10 TRAP IH 2	D1 10 TRAP IH 2	E1 10 TRAP IH 2	F1 10 TRAP IH 2
02 5-10 MOVW ID-ID 4	12 9 EMACS SP 4	22 4/3 LBHI RL 4	32 10 TRAP IH 2	42 2 INCX IH 2	52 2 INCY IH 2	62 4-7 INCW ID 3-5	72 5 INCW EX 4	82 3 SBEX IM 4	92 4 SBEX DI 3	A2 4-7 SBEX ID 3-5	B2 4 SBEX EX 4	C2 3 SBEY IM 4	D2 4 SBEY DI 3	E2 4-7 SBEY ID 3-5	F2 4 SBEY EX 4
03 5 MOVW IM-EX 6	13 3 EMULS IH 2	23 4/3 LBLS RL 4	33 10 TRAP IH 2	43 2 DECX IH 2	53 2 DECY IH 2	63 4-7 DECW ID 3-5	73 5 DECW EX 4	83 3 SBED IM 4	93 4 SBED DI 3	A3 4-7 SBED ID 3-5	B3 4 SBED EX 4	C3 3 ADED IM 4	D3 4 ADED DI 3	E3 4-7 ADED ID 3-5	F3 4 ADED EX 4
04 6 MOVW EX-EX 6	14 12 EDIVS IH 2	24 4/3 LBCC RL 4	34 10 TRAP IH 2	44 2 LSRX IH 2	54 2 LSRY IH 2	64 4-7 LSRW ID 3-5	74 5 LSRW EX 4	84 3 ANDX IM 4	94 4 ANDX DI 3	A4 4-7 ANDX ID 3-5	B4 3 ANDX EX 4	C4 3 ANDY IM 4	D4 4 ANDY DI 3	E4 4-7 ANDY ID 3-5	F4 3 ANDY EX 4
05 5-8 MOVW ID-EX 5	15 12 IDIVS IH 2	25 4/3 LBCC RL 4	35 5 BTAS DI 4	45 2 ROLX IH 2	55 2 ROLY IH 2	65 4-7 ROLW ID 3-5	75 5 ROLW EX 4	85 3 BITX IM 4	95 4 BITX DI 3	A5 4-7 BITX ID 3-5	B5 3 BITX EX 4	C5 3 BITY IM 4	D5 4 BITY DI 3	E5 4-7 BITY ID 3-5	F5 3 BITY EX 4
06 2 ABA IH 2	16 2 SBA IH 2	26 4/3 LBNE RL 4	36 6 BTAS EX 5	46 2 RORX IH 2	56 2 RORY IH 2	66 4-7 RORW ID 3-5	76 5 RORW EX 4	86 10 TRAP IH 2	96 4 GLDAA DI 3	A6 4-7 GLDAA ID 3-5	B6 4 GLDAA EX 4	C6 10 TRAP IH 2	D6 4 GLDAB DI 3	E6 4-7 GLDAB ID 3-5	F6 4 GLDAB EX 4
07 3 DAA IH 2	17 2 CBA IH 2	27 4/3 LBEQ RL 4	37 5-7 BTAS ID 4-6	47 2 ASRX IH 2	57 2 ASRY IH 2	67 4-7 ASRW ID 3-5	77 5 ASRW EX 4	87 2 CLRZ IH 2	97 2 TSTX IH 2	A7 10 SYSTRAP IH 2	B7 10 TRAP IH 2	C7 2 CLRY IH 2	D7 2 TSTY IH 2	E7 4-7 TSTW ID 3-5	F7 4 TSTW EX 4
08 4-6 MOVB IM-ID 4	18 4-7 MAXA ID 3-5	28 4/3 LBVC RL 4	38 4 PULCW IH 2	48 2 ASLX IH 2	58 2 ASLY IH 2	68 4-7 ASLW ID 3-5	78 5 ASLW EX 4	88 3 EORX IM 4	98 4 EORX DI 3	A8 4-7 EORX ID 3-5	B8 3 EORX EX 4	C8 3 EORY IM 4	D8 4 EORY DI 3	E8 4-7 EORY ID 3-5	F8 3 EORY EX 4
09 5-7 MOVB EX-ID 5	19 4-7 MINA ID 3-5	29 4/3 LBVS RL 4	39 3 PSHCW IH 2	49 10 TRAP IH 2	59 10 TRAP IH 2	69 4-7 CLRZ ID 3-5	79 5 CLRZ EX 4	89 3 ADEX IM 4	99 4 ADEX DI 3	A9 4-7 ADEX ID 3-5	B9 4 ADEX EX 4	C9 3 ADEY IM 4	D9 4 ADEY DI 3	E9 4-7 ADEY ID 3-5	F9 4 ADEY EX 4
0A 5-10 MOVB ID-ID 4	1A 4-7 EMAXD ID 3-5	2A 4/3 LBPL RL 4	3A 3n REV SP 2	4A 10 TRAP IH 2	5A 3 GSTAA DI 3	6A 3-5 GSTAA ID 3-5	7A 4 GSTAA EX 4	8A 3 ORX IM 4	9A 4 ORX DI 3	AA 4-7 ORX ID 3-5	BA 3 ORX EX 4	CA 3 ORY IM 4	DA 4 ORY DI 3	EA 4-7 ORY ID 3-5	FA 3 ORY EX 4
0B 4 MOVB IM-EX 5	1B 4-7 EMIND ID 3-5	2B 4/3 LBMI RL 4	3B 3n/3n REVW SP 2	4B 10 TRAP IH 2	5B 3 GSTAB DI 3	6B 3-5 GSTAB ID 3-5	7B 4 GSTAB EX 4	8B 3 ADDX IM 4	9B 4 ADDX DI 3	AB 4-7 ADDX ID 3-5	BB 4 ADDX EX 4	CB 3 ADDY IM 4	DB 4 ADDY DI 3	EB 4-7 ADDY ID 3-5	FB 4 ADDY EX 4
0C 6 MOVB EX-EX 6	1C 4-7 MAXM ID 3-5	2C 4/3 LBGE RL 4	3C 7B WAV SP 2	4C 10 TRAP IH 2	5C 3 GSTD DI 3	6C 3-5 GSTD ID 3-5	7C 4 GSTD EX 4	8C 3 CPED IM 4	9C 4 CPED DI 3	AC 4-7 CPED ID 3-5	BC 4 CPED EX 4	CC 10 TRAP IH 2	DC 4 GLDD DI 3	EC 4-7 GLDD ID 3-5	FC 4 GLDD EX 4
0D 5-8 MOVB ID-EX 5	1D 4-7 MINM ID 3-5	2D 4/3 LBTL RL 4	3D 6 TBL ID 3	4D 10 TRAP IH 2	5D 3 GSTY DI 3	6D 3-5 GSTY ID 3-5	7D 4 GSTY EX 4	8D 3 CPEY IM 4	9D 4 CPEY DI 3	AD 4-7 CPEY ID 3-5	BD 4 CPEY EX 4	CD 10 TRAP IH 2	DD 4 GLDY DI 3	ED 4-7 GLDY ID 3-5	FD 4 GLDY EX 4
0E 2 TAB IH 2	1E 4-7 EMAXM ID 3-5	2E 4/3 LBGT RL 4	3E 8 STOP IH 2	4E 10 TRAP IH 2	5E 3 GSTX DI 3	6E 3-5 GSTX ID 3-5	7E 4 GSTX EX 4	8E 3 CPEX IM 4	9E 4 CPEX DI 3	AE 4-7 CPEX ID 3-5	BE 4 CPEX EX 4	CE 10 TRAP IH 2	DE 4 GLDX DI 3	EE 4-7 GLDX ID 3-5	FE 4 GLDX EX 4
0F 2 TBA IH 2	1F 4-7 EMINM ID 3-5	2F 4/3 LBLE RL 4	3F 8 ETBL ID 3	4F 10 TRAP IH 2	5F 3 GSTS DI 3	6F 3-5 GSTS ID 3-5	7F 4 GSTS EX 4	8F 3 CPES IM 4	9F 4 CPES DI 3	AF 4-7 CPES ID 3-5	BF 4 CPES EX 4	CF 10 TRAP IH 2	DF 4 GLDS DI 3	EF 4-7 GLDS ID 3-5	FF 4 GLDS EX 4

* The opcode \$04 (on sheet 1 of 3) corresponds to one of the loop primitive instructions DBEQ, DBNE, IBEQ, IBNE, TBEQ, or TBNE.

† Refer to instruction summary for more information.

Page 2 When the CPU12X encounters a page 2 opcode (\$18 on page 1 of the opcode map), it treats the next byte of object code as a page 2 instruction opcode.

Table A-3. Indexed Addressing Mode Postbyte Encoding (xb)

00	0,X 5b const	10	-16,X 5b const	20	1,+X pre-inc	30	1,X+ post-inc	40	0,Y 5b const	50	-16,Y 5b const	60	1,+Y pre-inc	70	1,Y+ post-inc	80	0,SP 5b const	90	-16,SP 5b const	A0	1,+SP pre-inc	B0	1,SP+ post-inc	C0	0,PC 5b const	D0	-16,PC 5b const	E0	n,X 9b const	F0	n,SP 9b const
01	1,X 5b const	11	-15,X 5b const	21	2,+X pre-inc	31	2,X+ post-inc	41	1,Y 5b const	51	-15,Y 5b const	61	2,+Y pre-inc	71	2,Y+ post-inc	81	1,SP 5b const	91	-15,SP 5b const	A1	2,+SP pre-inc	B1	2,SP+ post-inc	C1	1,PC 5b const	D1	-15,PC 5b const	E1	-n,X 9b const	F1	-n,SP 9b const
02	2,X 5b const	12	-14,X 5b const	22	3,+X pre-inc	32	3,X+ post-inc	42	2,Y 5b const	52	-14,Y 5b const	62	3,+Y pre-inc	72	3,Y+ post-inc	82	2,SP 5b const	92	-14,SP 5b const	A2	3,+SP pre-inc	B2	3,SP+ post-inc	C2	2,PC 5b const	D2	-14,PC 5b const	E2	n,X 16b const	F2	n,SP 16b const
03	3,X 5b const	13	-13,X 5b const	23	4,+X pre-inc	33	4,X+ post-inc	43	3,Y 5b const	53	-13,Y 5b const	63	4,+Y pre-inc	73	4,Y+ post-inc	83	3,SP 5b const	93	-13,SP 5b const	A3	4,+SP pre-inc	B3	4,SP+ post-inc	C3	3,PC 5b const	D3	-13,PC 5b const	E3	[n,X] 16b indir	F3	[n,SP] 16b indir
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 indir	FB	[n,PC] 16b indir
0C	12,X 5b const	1C	-4,X 5b const	2C	4,-X pre-dec	3C	4,X- post-dec	4C	12,Y 5b const	5C	-4,Y 5b const	6C	4,-Y pre-dec	7C	4,Y- post-dec	8C	12,SP 5b const	9C	-4,SP 5b const	AC	4,-SP pre-dec	BC	4,SP- post-dec	CC	12,PC 5b const	DC	-4,PC 5b const	EC	A,Y A offset	FC	A,PC A offset
0D	13,X 5b const	1D	-3,X 5b const	2D	3,-X pre-dec	3D	3,X- post-dec	4D	13,Y 5b const	5D	-3,Y 5b const	6D	3,-Y pre-dec	7D	3,Y- post-dec	8D	13,SP 5b const	9D	-3,SP 5b const	AD	3,-SP pre-dec	BD	3,SP- post-dec	CD	13,PC 5b const	DD	-3,PC 5b const	ED	B,Y B offset	FD	B,PC B offset
0E	14,X 5b const	1E	-2,X 5b const	2E	2,-X pre-dec	3E	2,X- post-dec	4E	14,Y 5b const	5E	-2,Y 5b const	6E	2,-Y pre-dec	7E	2,Y- post-dec	8E	14,SP 5b const	9E	-2,SP 5b const	AE	2,-SP pre-dec	BE	2,SP- post-dec	CE	14,PC 5b const	DE	-2,PC 5b const	EE	D,Y D offset	FE	D,PC D offset
0F	15,X 5b const	1F	-1,X 5b const	2F	1,-X pre-dec	3F	1,X- post-dec	4F	15,Y 5b const	5F	-1,Y 5b const	6F	1,-Y pre-dec	7F	1,Y- post-dec	8F	15,SP 5b const	9F	-1,SP 5b const	AF	1,-SP pre-dec	BF	1,SP- post-dec	CF	15,PC 5b const	DF	-1,PC 5b const	EF	[D,Y] D indirect	FF	[D,PC] D indirect

Key to Table A-3

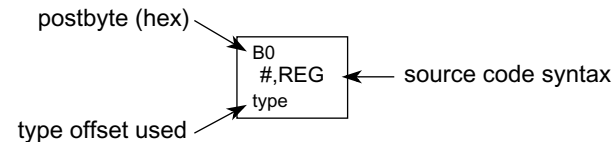


Table A-4. Indexed Addressing Mode Summary

Postbyte Code (xb)	Operand Syntax	Comments
rr0nnnnn	,r n,r -n,r	5-bit constant offset n = -16 to +15 rr can specify X, Y, SP, or PC
111rr0zs	n,r -n,r	Constant offset (9- or 16-bit signed) z- 0 = 9-bit with sign in LSB of postbyte (s) 1 = 16-bit if z = s = 1, 16-bit offset indexed-indirect (see below) rr can specify X, Y, SP, or PC
rr1pnnnn	n,-r n,+r n,r- n,r+	Auto predecrement, preincrement, postdecrement, or postincrement; p = pre-(0) or post-(1), n = -8 to -1, +1 to +8 rr can specify X, Y, or SP (PC not a valid choice)
111rr1aa	A,r B,r D,r	Accumulator offset (unsigned 8-bit or 16-bit) aa -00 = A 01 = B 10 = D (16-bit) 11 = see accumulator D offset indexed-indirect rr can specify X, Y, SP, or PC
111rr011	[n,r]	16-bit offset indexed-indirect rr can specify X, Y, SP, or PC
111rr111	[D,r]	Accumulator D offset indexed-indirect rr can specify X, Y, SP, or PC

Table A-5. Transfer and Exchange Postbyte Encoding

TRANSFERS									
	MS ⇒	0	1	2	3	4	5	6	7
↓ LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR _L ⇒ A TFR CCR,A TFR CCRL,A	TMP3 _L ⇒ A TFR TMP3,A TFR TMP3L,A	B ⇒ A TFR D,A	X _L ⇒ A TFR X, A TFR XL,A	Y _L ⇒ A TFR Y,A TFR YL,A	SP _L ⇒ A TFR SP,A TFR SPL,A
1	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR _L ⇒ B TFR CCR,B TFR CCRL,B	TMP3 _L ⇒ B TFR TMP3,B TFR TMP3L,B	B ⇒ B TFR D,B	X _L ⇒ B TFR X, B TFR XL,B	Y _L ⇒ B TFR Y,B TFR YL,B	SP _L ⇒ B TFR SP,B TFR SPL,B
2	CCR	A ⇒ CCR TFR A,CCR TFR A,CCRL	B ⇒ CCR TFR B,CCR TFR B,CCRL	CCR _L ⇒ CCR _L TFR CCR,CCR TFR CCRL,CCRL	TMP3 _L ⇒ CCR TFR TMP3,CCR TFR TMP3L,CCRL	B ⇒ CCR TFR D,CCR TFR D,CCRL	X _L ⇒ CCR TFR X,CCR TFR XL,CCRL	Y _L ⇒ CCR TFR Y,CCR TFR YL,CCRL	SP _L ⇒ CCR TFR SP,CCR TFR SPL,CCRL
3	TMP 2	sex:A ⇒ TMP2 SEX A,TMP2	sex:B ⇒ TMP2 SEX B,TMP2	sex:CCR _L ⇒ TMP2 SEX CCR,TMP2 SEX CCRL,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP2 TFR D,TMP2	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
4	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	sex:CCR _L ⇒ D SEX CCR _L ,D SEX CCRL,D	TMP3 ⇒ D TFR TMP3,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
5	X	sex:A ⇒ X SEX A,X	sex:B ⇒ X SEX B,X	sex:CCR _L ⇒ X SEX CCR,X SEX CCRL,X	TMP3 ⇒ X TFR TMP3,X	D ⇒ X TFR D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
6	Y	sex:A ⇒ Y SEX A,Y	sex:B ⇒ Y SEX B,Y	sex:CCR _L ⇒ Y SEX CCR,Y SEX CCRL,Y	TMP3 ⇒ Y TFR TMP3,Y	D ⇒ Y TFR D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
7	SP	sex:A ⇒ SP SEX A,SP	sex:B ⇒ SP SEX B,SP	sex:CCR _L ⇒ SP SEX CCR,SP SEX CCRL,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP
8	A	A ⇒ A TFR A,A	B ⇒ A TFR B,A	CCR _H ⇒ A TFR CCRH,A	TMP3 _H ⇒ A TFR TMP3H,A	B ⇒ A TFR D,A	X _H ⇒ A TFR XH, A	Y _H ⇒ A TFR YH,A	SP _H ⇒ A TFR SPH,A
9	B	A ⇒ B TFR A,B	B ⇒ B TFR B,B	CCR _L ⇒ B TFR CCRL,B	TMP3 _L ⇒ B TFR TMP3L,B	B ⇒ B TFR D,B	X _L ⇒ B TFR XL, B	Y _L ⇒ B TFR YL,B	SP _L ⇒ B TFR SPL,B
A	CCR	A ⇒ CCR _H TFR A,CCRH	B ⇒ CCR _L TFR B,CCRL	CCR _W ⇒ CCR _W TFR CCRW,CCRW	TMP3 ⇒ CCR _{H:L} TFR TMP3,CCRW	D ⇒ CCR _{H:L} TFR D,CCRW	X ⇒ CCR _{H:L} TFR X,CCRW	Y ⇒ CCR _{H:L} TFR Y,CCRW	SP ⇒ CCR _{H:L} TFR SP,CCRW
B	TMP x	A ⇒ TMP2 _H TFR A,TMP2H	B ⇒ TMP2 _L TFR B,TMP2L	CCR _{H:L} ⇒ TMP2 TFR CCRW,TMP2	TMP3 ⇒ TMP2 TFR TMP3,TMP2	D ⇒ TMP1 TFR D,TMP1	X ⇒ TMP2 TFR X,TMP2	Y ⇒ TMP2 TFR Y,TMP2	SP ⇒ TMP2 TFR SP,TMP2
C	D	sex:A ⇒ D SEX A,D	sex:B ⇒ D SEX B,D	CCR _{H:L} ⇒ D TFR CCRW,D	TMP1 ⇒ D TFR TMP1,D	D ⇒ D TFR D,D	X ⇒ D TFR X,D	Y ⇒ D TFR Y,D	SP ⇒ D TFR SP,D
D	X	A ⇒ X _H TFR A,XH	B ⇒ X _L TFR B,XL	CCR _{H:L} ⇒ X TFR CCRW,X	TMP3 ⇒ X TFR TMP3,X	sex:D ⇒ X SEX D,X	X ⇒ X TFR X,X	Y ⇒ X TFR Y,X	SP ⇒ X TFR SP,X
E	Y	A ⇒ Y _H TFR A,YH	B ⇒ Y _L TFR B,YL	CCR _{H:L} ⇒ Y TFR CCRW,Y	TMP3 ⇒ Y TFR TMP3,Y	sex:D ⇒ Y SEX D,Y	X ⇒ Y TFR X,Y	Y ⇒ Y TFR Y,Y	SP ⇒ Y TFR SP,Y
F	SP	A ⇒ SP _H TFR A,SPH	B ⇒ SP _L TFR B,SPL	CCR _{H:L} ⇒ SP TFR CCRW,SP	TMP3 ⇒ SP TFR TMP3,SP	D ⇒ SP TFR D,SP	X ⇒ SP TFR X,SP	Y ⇒ SP TFR Y,SP	SP ⇒ SP TFR SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the CPU12X.

Table A-5. Transfer and Exchange Postbyte Encoding (continued)

EXCHANGES									
	MS⇒	8	9	A	B	C	D	E	F
↓ LS		A	B	CCR	TMPx	D	X	Y	SP
0	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR _L ⇔ A EXG CCR,A EXG CCRL,A	TMP3 _L ⇔ A \$00:A ⇒ TMP3 EXG A, TMP3	B ⇔ A EXG D,A	X _L ⇔ A \$00:A ⇒ X EXG X,A	Y _L ⇔ A \$00:A ⇒ Y EXG Y,A	SP _L ⇔ A \$00:A ⇒ SP EXG SP,A
1	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR _L ⇔ B EXG CCR,B EXG CCRL,B	TMP3 _L ⇔ B \$FF:B ⇒ TMP3 EXG B,TMP3	B ⇔ B \$FF:B ⇒ A EXG D,B	X _L ⇔ B \$FF:B ⇒ X EXG X,B	Y _L ⇔ B \$FF:B ⇒ Y EXG Y,B	SP _L ⇔ B \$FF:B ⇒ SP EXG SP,B
2	CCR	A ⇔ CCR _L EXG A, CCR EXG A,CCRL	B ⇔ CCR _L EXG B,CCR EXG B,CCRL	CCR _L ⇔ CCR _L EXG CCR,CCR EXG CCRL,CCRL	TMP3 _L ⇔ CCR _L \$FF:CCR _L ⇒ TMP3 EXG, TMP3,CCR EXG TMP3,CCRL	B ⇔ CCR _L \$FF:CCR _L ⇒ D EXG D,CCR EXG D,CCRL	X _L ⇔ CCR _L \$FF:CCR _L ⇒ X EXG X,CCR EXG X,CCRL	Y _L ⇔ CCR _L \$FF:CCR _L ⇒ Y EXG Y,CCR EXG Y,CCRL	SP _L ⇔ CCR _L \$FF:CCR _L ⇒ SP EXG SP,CCR EXG SP,CCRL
3	TMP 2	\$00:A ⇒ TMP2 TMP2 _L ⇒ A EXG A,TMP2	\$00:B ⇒ TMP2 TMP2 _L ⇒ B EXG B,TMP2	\$00:CCR _L ⇒ TMP2 TMP2 _L ⇒ CCR EXG CCR,TMP2	TMP3 ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP2 EXG D,TMP2	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
4	D	\$00:A ⇒ D EXG A,D	\$00:B ⇒ D EXG B,D	\$00:CCR _L ⇒ D B ⇒ CCR _L EXG CCR,D EXG CCRL,D	TMP3 ⇔ D EXG TMP3,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
5	X	\$00:A ⇒ X X _L ⇒ A EXG A,X	\$00:B ⇒ X X _L ⇒ B EXG B,X	\$00:CCR _L ⇒ X X _L ⇒ CCR _L EXG CCR,X EXG CCRL,X	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
6	Y	\$00:A ⇒ Y Y _L ⇒ A EXG A,Y	\$00:B ⇒ Y Y _L ⇒ B EXG B,Y	\$00:CCR _L ⇒ Y Y _L ⇒ CCR _L EXG CCR,Y EXG CCRL,Y	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
7	SP	\$00:A ⇒ SP SP _L ⇒ A EXG A,SP	\$00:B ⇒ SP SP _L ⇒ B EXG B,SP	\$00:CCR _L ⇒ SP SP _L ⇒ CCR _L EXG CCR,SP EXG CCRL,SP	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP
8	A	A ⇔ A EXG A,A	B ⇔ A EXG B,A	CCR _H ⇔ A EXG CCRH,A	TMP3 _H ⇔ A EXG TMP3H,A	B ⇔ A EXG D,A	X _H ⇔ A EXG XH,A	Y _H ⇔ A EXG YH,A	SP _H ⇔ A EXG SPH,A
9	B	A ⇔ B EXG A,B	B ⇔ B EXG B,B	CCR _L ⇔ B EXG CCRL,B	TMP3 _L ⇔ B EXG TMP3L,B	\$FF ⇒ A, B ⇒ B EXG D,B	X _L ⇔ B EXG XL,B	Y _L ⇔ B EXG YL,B	SP _L ⇔ B EXG SPL,B
A	CCR	A ⇔ CCR _H EXG A,CCRH	B ⇔ CCR _L EXG B,CCRL	CCR _{H:L} ⇔ CCR _{H:L} EXG CCRW,CCRW	TMP3 ⇔ CCR _{H:L} EXG TMP3,CCRW	D ⇔ CCR _{H:L} EXG D,CCRW	X ⇔ CCR _{H:L} EXG X,CCRW	Y ⇔ CCR _{H:L} EXG Y,CCRW	SP ⇔ CCR _{H:L} EXG SP,CCRW
B	TMP x	A ⇔ TMP2 _H EXG A,TMP2H	B ⇔ TMP2 _L EXG B,TMP2L	CCR _{H:L} ⇔ TMP2 EXG CCRW,TMP2	TMP3 ⇔ TMP2 EXG TMP3,TMP2	D ⇔ TMP1 EXG D,TMP1	X ⇔ TMP2 EXG X,TMP2	Y ⇔ TMP2 EXG Y,TMP2	SP ⇔ TMP2 EXG SP,TMP2
C	D	\$00:A ⇒ D EXG A,D	\$00:B ⇒ D EXG B,D	CCR _{H:L} ⇔ D EXG CCRW,D	TMP1 ⇔ D EXG TMP1,D	D ⇔ D EXG D,D	X ⇔ D EXG X,D	Y ⇔ D EXG Y,D	SP ⇔ D EXG SP,D
D	X	A ⇔ X _H EXG A,XH	B ⇔ X _L EXG B,XL	CCR _{H:L} ⇔ X EXG CCRW,X	TMP3 ⇔ X EXG TMP3,X	D ⇔ X EXG D,X	X ⇔ X EXG X,X	Y ⇔ X EXG Y,X	SP ⇔ X EXG SP,X
E	Y	A ⇔ Y _H EXG A,YH	B ⇔ Y _L EXG B,YL	CCR _{H:L} ⇔ Y EXG CCRW,Y	TMP3 ⇔ Y EXG TMP3,Y	D ⇔ Y EXG D,Y	X ⇔ Y EXG X,Y	Y ⇔ Y EXG Y,Y	SP ⇔ Y EXG SP,Y
F	SP	A ⇔ SP _H EXG A,SPH	B ⇔ SP _L EXG B,SPL	CCR _{H:L} ⇔ SP EXG CCRW,SP	TMP3 ⇔ SP EXG TMP3,SP	D ⇔ SP EXG D,SP	X ⇔ SP EXG X,SP	Y ⇔ SP EXG Y,SP	SP ⇔ SP EXG SP,SP

Note: Encodings in the shaded area (LS = 8–F) are only available on the CPU12X.

Table A-6. Loop Primitive Postbyte Encoding (Ib)

00	A	10	A	20	A	30	A	40	A	50	A	60	A	70	A	80	A	90	A	A0	A	B0	A
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	
01	B	11	B	21	B	31	B	41	B	51	B	61	B	71	B	81	B	91	B	A1	B	B1	B
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	
02		12		22		32		42		52		62		72		82		92		A2		B2	
—		—		—		—		—		—		—		—		—		—		—		—	
03		13		23		33		43		53		63		73		83		93		A3		B3	
—		—		—		—		—		—		—		—		—		—		—		—	
04	D	14	D	24	D	34	D	44	D	54	D	64	D	74	D	84	D	94	D	A4	D	B4	D
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	
05	X	15	X	25	X	35	X	45	X	55	X	65	X	75	X	85	X	95	X	A5	X	B5	X
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	
06	Y	16	Y	26	Y	36	Y	46	Y	56	Y	66	Y	76	Y	86	Y	96	Y	A6	Y	B6	Y
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	
07	SP	17	SP	27	SP	37	SP	47	SP	57	SP	67	SP	77	SP	87	SP	97	SP	A7	SP	B7	SP
DBEQ		DBEQ		DBNE		DBNE		TBEQ		TBEQ		TBNE		TBNE		IBEQ		IBEQ		IBNE		IBNE	
(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)		(+)		(-)	

Key to Table A-6

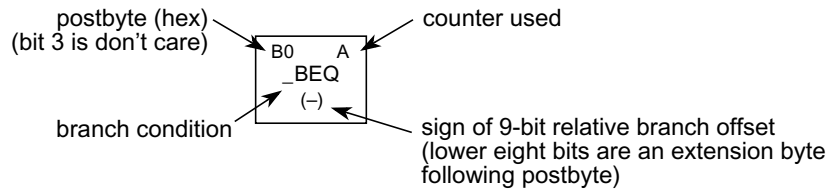


Table A-7. Branch/Complementary Branch

Branch				Complementary Branch			
Test	Mnemonic	Opcode	Boolean	Test	Mnemonic	Opcode	Comment
$r > m$	BGT	2E	$Z + (N \oplus V) = 0$	$r \leq m$	BLE	2F	Signed
$r \geq m$	BGE	2C	$N \oplus V = 0$	$r < m$	BLT	2D	Signed
$r = m$	BEQ	27	$Z = 1$	$r \neq m$	BNE	26	Signed
$r \leq m$	BLE	2F	$Z + (N \oplus V) = 1$	$r > m$	BGT	2E	Signed
$r < m$	BLT	2D	$N \oplus V = 1$	$r \geq m$	BGE	2C	Signed
$r > m$	BHI	22	$C + Z = 0$	$r \leq m$	BLS	23	Unsigned
$r \geq m$	BHS/BCC	24	$C = 0$	$r < m$	BLO/BCS	25	Unsigned
$r = m$	BEQ	27	$Z = 1$	$r \neq m$	BNE	26	Unsigned
$r \leq m$	BLS	23	$C + Z = 1$	$r > m$	BHI	22	Unsigned
$r < m$	BLO/BCS	25	$C = 1$	$r \geq m$	BHS/BCC	24	Unsigned
Carry	BCS	25	$C = 1$	No Carry	BCC	24	Simple
Negative	BMI	2B	$N = 1$	Plus	BPL	2A	Simple
Overflow	BVS	29	$V = 1$	No Overflow	BVC	28	Simple
$r = 0$	BEQ	27	$Z = 1$	$r \neq 0$	BNE	26	Simple
Always	BRA	20	—	Never	BRN	21	Unconditional

For 16-bit offset long branches precede opcode with a \$18 page prebyte.

Table A-8. Hexadecimal to ASCII Conversion

Hex	ASCII	Hex	ASCII	Hex	ASCII	Hex	ASCII
\$00	NUL	\$20	SP <i>space</i>	\$40	@	\$60	` <i>grave</i>
\$01	SOH	\$21	!	\$41	A	\$61	a
\$02	STX	\$22	" <i>quote</i>	\$42	B	\$62	b
\$03	ETX	\$23	#	\$43	C	\$63	c
\$04	EOT	\$24	\$	\$44	D	\$64	d
\$05	ENQ	\$25	%	\$45	E	\$65	e
\$06	ACK	\$26	&	\$46	F	\$66	f
\$07	BEL <i>beep</i>	\$27	' <i>apost.</i>	\$47	G	\$67	g
\$08	BS <i>back sp</i>	\$28	(\$48	H	\$68	h
\$09	HT <i>tab</i>	\$29)	\$49	I	\$69	i
\$0A	LF <i>linefeed</i>	\$2A	*	\$4A	J	\$6A	j
\$0B	VT	\$2B	+	\$4B	K	\$6B	k
\$0C	FF	\$2C	, <i>comma</i>	\$4C	L	\$6C	l
\$0D	CR <i>return</i>	\$2D	- <i>dash</i>	\$4D	M	\$6D	m
\$0E	SO	\$2E	. <i>period</i>	\$4E	N	\$6E	n
\$0F	SI	\$2F	/	\$4F	O	\$6F	o
\$10	DLE	\$30	0	\$50	P	\$70	p
\$11	DC1	\$31	1	\$51	Q	\$71	q
\$12	DC2	\$32	2	\$52	R	\$72	r
\$13	DC3	\$33	3	\$53	S	\$73	s
\$14	DC4	\$34	4	\$54	T	\$74	t
\$15	NAK	\$35	5	\$55	U	\$75	u
\$16	SYN	\$36	6	\$56	V	\$76	v
\$17	ETB	\$37	7	\$57	W	\$77	w
\$18	CAN	\$38	8	\$58	X	\$78	x
\$19	EM	\$39	9	\$59	Y	\$79	y
\$1A	SUB	\$3A	:	\$5A	Z	\$7A	z
\$1B	ESCAPE	\$3B	;	\$5B	[\$7B	{
\$1C	FS	\$3C	<	\$5C	\	\$7C	
\$1D	GS	\$3D	=	\$5D]	\$7D	}
\$1E	RS	\$3E	>	\$5E	^	\$7E	~
\$1F	US	\$3F	?	\$5F	_ <i>under</i>	\$7F	DEL <i>delete</i>

A.5 Hexadecimal-to-Decimal Conversion

To convert a hexadecimal number (up to four hexadecimal digits) to decimal, look up the decimal equivalent of each hexadecimal digit in [Table A-9](#). The decimal equivalent of the original hexadecimal number is the sum of the weights found in the table for all hexadecimal digits.

Table A-9. Hexadecimal to/from Decimal Conversion

15 Bit 8				7 Bit 0			
15 12		11 8		7 4		3 0	
4th Hex Digit		3rd Hex Digit		2nd Hex Digit		1st Hex Digit	
Hex	Decimal	Hex	Decimal	Hex	Decimal	Hex	Decimal
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,484	E	224	E	14
F	61,440	F	3,840	F	240	F	15

A.6 Decimal-to-Hexadecimal Conversion

To convert a decimal number (up to 65,535₁₀) to hexadecimal, find the largest decimal number in [Table A-9](#) that is less than or equal to the number you are converting. The corresponding hexadecimal digit is the most significant hexadecimal digit of the result. Subtract the decimal number found from the original decimal number to get the *remaining decimal value*. Repeat the procedure using the remaining decimal value for each subsequent hexadecimal digit.

Appendix B

CPU12V0 - CPU12V1

B.1 Introduction

This appendix discusses the differences between the CPU12V0 and the CPU12V1. In general, the CPU12V1 is a proper subset of the CPU12V0. Changes have been made to support the S12X bus architecture and improve the efficiency of several CPU functions.

B.2 CPU12V1 Modifications

CPU12V0 similarities:

- Source code compatible, with the exception of removed instructions
- Same stacking operations
- Same programmer's model
- Improved performance

CPU12V0 differences:

- 4 fuzzy instructions removed (MEM, REV, REVW, WAV/WAVR)
- bus signature - 'Access Details'

B.3 Source Code Compatibility

All CPU12V1 instruction mnemonic and source code statements can be assembled directly with a CPU12V0 assembler with no modifications.

The removed MEM, REV, REVW, WAV, and WAVR instructions perform a TRAP on the CPU12V1. The operation of these instructions may be reproduced in software.

CPU12V1 object code is identical to CPU12V0 object code. For CPU12V0 compatibility, the TFR and EXG instructions must have post-byte bit 3 set to zero. TFR/EXG post-byte bit 3 (object code description "eb", bit 3) is not used on the CPU12V0 and must be set to zero on the CPU12V1 to guarantee identical operation.

B.4 Programmer's Model and Stacking

The CPU12V1 programming model and stacking order are identical to those of the CPU12V0. The CPU12V1 uses a nine byte stack frame.

B.5 Improved Performance

The CPU12V1 employs many of the architectural enhancements that are found on the CPU12X. Dedicated address generation logic and pipelined execution improves the CPU12V1 performance relative to the CPU12V0 architecture.

B.6 Bus Signatures - Access Details

The CPU12V1 uses the same bus access details as the CPU12V0 with the exception of the EMACS and EMUL instructions. These bus access details are of equal or smaller length than the CPU12V0.

Table B-1 shows the bus access detail differences.

Table B-1. Access Detail Differences

Instruction	CPU12V0 Access Detail	CPU12V1 Access Detail
EMACS	ORROffRRfWWP	ORROfRRfWWP
LEA oprx0_xysp	Pf	P

Appendix C

CPU12 - CPU12X

C.1 Introduction

This appendix is intended for developers creating new applications for the CPU12 who wish to maximize compatibility with the CPU12X. It is also useful for developers creating applications for the CPU12X, who are familiar with the CPU12X family. It describes how the new functionality of the CPU12X differs from the CPU12 and how to take advantage of these new features. In general, the CPU12X is a superset of the CPU12. Changes have been made to enhance the 16 bits operations, to support the CPU12X bus architecture and improve the efficiency of several CPU functions.

C.2 CPU12X Modifications

CPU12 similarities:

- Source code compatible, with the exception of removed and/or added instructions.
- Improved performance

CPU12 differences:

- Different stacking operations
- Different programmer's model
- bus signature - 'Access Details'

C.3 Source Code Compatibility

The CPU12X retains all of the instruction set of the CPU12 unchanged. The only exception that requires consideration is that the interrupt stack frame on the CPU12X is increased by one byte due to the extended Condition Code Register. CPU12 users should carefully check the maximum number of interrupt stack frames that can simultaneously exist and increment the stack space by one byte for every frame.

C.3.1 Instruction Set

The CPU12X features an enhanced instruction set over the CPU12. All of the existing CPU12 instructions are retained. There are four classes of new instructions.

1. New 16-bit, where only an 8-bit accumulator operation existed
2. New memory access instructions, allowing access to linear banks of up to 64KB
3. New instruction designed to optimize semaphore handling
4. New addressing modes for MOVE instructions

Class 1 improves the data manipulation capabilities of the CPU12X by allowing direct operation on larger data sizes. On the CPU12, most arithmetic and logical operations, such as addition, can only take place by using the A, B or D accumulators. The CPU12X extends this capability to the X and Y registers and adds new instructions for the D register. All arithmetic and logical functions using the A or B accumulator will now have a 16-bit counterpart using the X and Y register. New instructions of this type are: ADE (add with carry), ADD (add without carry), SBE (subtract with carry), DEC (decrement) and INC (increment), SUB (subtract without carry), AND (logical AND), BIT (logical bit test), OR (logical OR), EOR (logical EXCLUSIVE OR), NEG (two's complement), COM (one's complement), CLR (clear register), TST (test register), LSL (logical shift left), ROL (rotate left), ASR (arithmetic shift right), LSR (logical shift right), and ROR (rotate right). The same addressing modes as for their counterparts with the A, B accumulator are available.

To improve the 32-bit capability of the D-Accumulator, ADED (add with carry) and SBED (subtract with carry) are added. In addition, the CPU12X provides a set of compare instructions carrying forward the carry and also the zero flag (CPED, CPEX, CPEY, CPES). This improves the capability to perform 32-bit compares.

While the existing architecture allows 8-bit read-modify-write instructions, the CPU12X extends this capability to 16-bit words and provides NEGW (two's complement), COMW (one's complement), DECW (decrement 16-bit), INCW (increment 16-bit), RORW (rotate right), LSRW (logical shift right), ARSW (arithmetic shift right), ROLW (rotate left), LSLW (logical shift left), CLRW (clear memory) and TSTW (test memory). Addressing modes are the same as for their 8-bit counterparts. In general, these new 16-bit operations allow significantly faster manipulation of data compared to the CPU12.

Class 2 provides access to a new mode available on the S12X Memory Management Controller (Refer to S12XMMC BlockGuide). This allows access to any 64K byte page in global memory based on a new MCU register in the core block (GPAGE). The new instructions include all the available addressing modes and concatenate the GPAGE register with the 16-bit address data. Global instructions are available for the following instructions: LDAA (load accumulator A), LDAB (load accumulator B), LDD (load accumulator D), LDX (load X register), LDY (load Y register), LDS (load stack pointer), STAA (store accumulator A), STAB (store accumulator B), STD (store accumulator D), STX (store X register), STY (store Y register), and STS (store stack pointer).

The GPAGE register is seven bits wide, so that global memory runs from \$00_0000 to \$7F_FFFF, and each location is accessible with a single instruction from anywhere in a program (once the GPAGE register is configured for that 64K byte page).

NOTE

Users developing code on the CPU12 for later use on the CPU12X should carefully note areas of opportunity/requirement to use this feature and modify their code, and their compiler and linker settings, once using the CPU12X.

Class 3 allows more efficient use of semaphores, which are important for real time operating systems (RTOS) and for sharing resources between the CPU and the XGATE. The new instruction is BTAS (bit test and set). Since this is a single instruction, it cannot be interrupted; therefore, it is useful when requesting access to resources.

Resources are usually locked via a status bit in RAM when the bit is set the resource is in use. On the CPU12, users must take care that two tasks cannot both appear to have allocated the resource. This can occur if one task interrupts another immediately after a bit test instruction. Therefore, tasks typically disable interrupts while checking and allocating resources. The BTAS instruction removes this need, as it tests and sets the resource bit in a single instruction step. BTAS follows the same syntax and allows the same addressing modes as the BSET instruction, except that the test is based on the original data and not on the data written back. A typical use for a BTAS instruction is shown below :

```
BTAS $1020, #$20
BNE ResourceNotAvailable
<ResourceLocked>
```

Class 4 is designed to improve the opportunity for compilers to use the memory-to-memory move instructions by allowing the use of all relevant CPU12X addressing modes, and not only those fitting in a single postbyte xb.

C.4 Programmer's Model and Stacking

The CPU12X features an enhanced Condition Code Register (CCR). This register has been extended to 16 bits to allow stacking of the interrupt priority. IPL[2:0] indicate the interrupt level of the CPU12X prior to the current interrupt. The CPU12X automatically updates the value of IPL[2:0] to the value of the interrupt currently being serviced.

Since the CCR has been extended to two bytes from one byte; this in turn causes the interrupt stack frame to be extended by one byte from nine bytes to ten bytes. Therefore all stack relative accesses are modified by one byte.

In practice, the requirement to extract information such as the Program Counter from an interrupt stack frame is an unusual activity (typically related to debug tools or perhaps task schedulers). Therefore for the vast majority of users this difference between the CPU12 and CPU12X will have little impact.

NOTE

Users developing code on the CPU12 for later use on the CPU12X need to be aware of the maximum stack size and any unusual requirements such as that described. Since each interrupt stack frame is one byte larger on the CPU12X, users must take account of this in increasing the memory reserved for the stack when moving to the CPU12X.

C.5 Improved Performance

Dedicated address generation logic and pipelined execution improves the CPU12X performance relative to the CPU12 architecture.

Example:

```
...
LEAS 1,SP ; takes 1 cycle on CPU12X (2 cycles on CPU12V0)
...
EMACS ; takes 9 cycles on CPU12X (13 cycles on CPU12V0)
...
```

C.6 Bus Signatures - Access Details

The CPU12X uses the same bus access details as the CPU12 with the exception of the different instructions shown on [Table C-1](#). These bus access details are of equal or smaller length than the CPU12.

[Table C-1](#) shows the bus access detail differences.

Table C-1. Access Detail Differences

Instruction	CPU12 Access Detail	CPU12X Access Detail
EMUL	ffO	O
EMULS	OffO	OfO
EMACS	ORROffRRfWWP	ORRORRWWP
ETBL	ORRfffffP	ORRffffP
RTI	uUUUUPPP	UUUUUPPP
STOP	OOSSSSsf	OOSSSSsf
SWI	VSPSSPSsP	VSPSSPSSP
WAI	OSSSSsf	OSSSSsf
LEA oprx0_xysp	Pf	P
MOVB	(EXT,IDX)	(EXT,IDX,IDX1,IDX2, [D,IDX], [IDX2])
MOVW	(EXT,IDX)	(EXT,IDX,IDX1,IDX2, [D,IDX], [IDX2])

Appendix D

CPU12XV0 - CPU12XV2

D.1 Introduction

This appendix discusses the differences between the CPU12XV0 and the CPU12XV2. In general, the CPU12XV2 is a proper superset of the CPU12XV0. Changes have been made to support user state.

D.2 CPU12XV2 Modifications

CPU12XV0 similarities:

- Source code compatible, with the exception of removed or added instructions
- Same stacking operations
- Same programmer's model

CPU12XV0 differences:

- 4 fuzzy instructions removed (MEM, REV, REVW, WAV/WAVR)
- new system-call instruction (SYS)
- User state support
 - U-bit in CCRW
 - Restrictions imposed on execution of some instructions in user state

D.3 Source Code Compatibility

All (but one) CPU12XV2 instruction mnemonic and source code statements can be assembled directly with an CPU12XV0 assembler with no modifications. Only exception is the new system-call instruction "SYS" (18 A7). The SYS instruction can be generated by an CPU12XV0 assembler using the "TRAP \$A7" statement.

The removed MEM, REV, REVW, WAV, and WAVR instructions perform a TRAP on the CPU12XV2. The operation of these instructions may be reproduced in software.

CPU12XV2 object code is identical to CPU12XV0 object code. If strict CPU12XV0 compatibility is desired, the SYS instruction should not be used to guarantee identical operation.

D.4 Programmer's Model and Stacking

The CPU12XV2 programming model and stacking order are identical to those of the CPU12XV0. Like the CPU12XV0, the CPU12XV2 uses a 10 byte stack frame.

D.5 Performance

Performance of the CPU12XV2 is identical to the CPU12XV0.

D.6 Bus Signatures - Access Details

The CPU12XV2 uses the same bus access details as the CPU12XV0. These bus access details are of equal length compared to the CPU12XV0.

D.7 Instruction Execution in User State

If the CPU12XV2 operates in user state, restrictions apply for the execution of several CPU instructions:

1. Write access to the system control bits in the Condition Code Register (U, IPL[2:0], S, X, I) is blocked, i.e. any attempts to change these bits are ignored. This affects the following instructions:
 - ANDCC (including the alias instruction CLI)
 - ORCC (including the alias instruction SEI)
 - EXG with CCR, CCRH or CCRW
 - TFR with CCR, CCRH or CCRW as destination (including the alias instruction TAP)
 - PULC
 - PULCW
 - RTI
2. Instructions which would cause the CPU to suspend instruction execution are treated as No-Operation instructions (NOP). This affects the following instructions:
 - STOP
 - WAI

Appendix E

CPU12XV0 - CPU12XV1

E.1 Introduction

This appendix discusses the differences between the CPU12XV0 and the CPU12XV1. In general, the CPU12XV1 is a proper superset of the CPU12XV0.

E.2 CPU12XV1 Modifications

CPU12XV0 similarities:

- Source code compatible, with the exception of removed or added instructions
- Same stacking operations
- Same programmer's model

CPU12XV0 differences:

- 4 fuzzy instructions removed (MEM, REV, REVW, WAV/WAVR)
- new system-call instruction (SYS)

E.3 Source Code Compatibility

All (but one) CPU12XV1 instruction mnemonic and source code statements can be assembled directly with an CPU12XV0 assembler with no modifications. Only exception is the new system-call instruction "SYS" (18 A7). The SYS instruction can be generated by an CPU12XV0 assembler using the "TRAP \$A7" statement.

The removed MEM, REV, REVW, WAV, and WAVR instructions perform a TRAP on the CPU12XV1. The operation of these instructions may be reproduced in software.

CPU12XV1 object code is identical to CPU12XV0 object code. If strict CPU12XV0 compatibility is desired, the SYS instruction should not be used to guarantee identical operation.

E.4 Programmer's Model and Stacking

The CPU12XV1 programming model and stacking order are identical to those of the CPU12XV0. Like the CPU12XV0, the CPU12XV1 uses a 10 byte stack frame.

E.5 Performance

Performance of the CPU12XV1 is identical to the CPU12XV0.

E.6 Bus Signatures - Access Details

The CPU12XV1 uses the same bus access details as the CPU12XV0. These bus access details are of equal length compared to the CPU12XV0.

Appendix F High-Level Language Support

F.1 Introduction

Many programmers are turning to high-level languages such as C as an alternative to coding in native assembly languages. High-level language (HLL) programming can improve productivity and produce code that is more easily maintained than assembly language programs. The most serious drawback to the use of HLL in MCUs has been the relatively large size of programs written in HLL. Larger program ROM size requirements translate into increased system costs.

This appendix identifies CPU12 Family instructions and addressing modes that provide improved support for high-level language. C language examples are provided to demonstrate how these features support efficient HLL structures and concepts. Since the CPU12 Family instruction set is a superset of the M68HC11 instruction set, some of the discussions use the M68HC11 as a basis for comparison.

F.2 Data Types

The CPU12 Family supports the bit-sized data type with bit manipulation instructions which are available in extended, direct, and indexed variations. The char data type is a simple 8-bit value that is commonly used to specify variables in a small microcontroller system because it requires less memory space than a 16-bit integer (provided the variable has a range small enough to fit into eight bits). The 16-bit CPU12 Family can easily handle 16-bit integer types and the available set of conditional branches (including long branches) allow branching based on signed or unsigned arithmetic results. Some of the higher math functions allow for division and multiplication involving 32-bit values, although it is somewhat less common to use such long values in a microcontroller system.

The CPU12 Family has special sign extension instructions to allow easy type-casting from smaller data types to larger ones, such as from char to integer. This sign extension is automatically performed when an 8-bit value is transferred to a 16-bit register.

F.3 Parameters and Variables

High-level languages make extensive use of the stack, both to pass variables and for temporary and local storage. It follows that there should be easy ways to push and pull each CPU register, stack pointer based indexing should be allowed, and that direct arithmetic manipulation of the stack pointer value should be allowed. The CPU12 Family instruction set provided for all of these needs with improved indexed addressing, the addition of an LEAS instruction, and the addition of push and pull instructions for the D accumulator and the CCR.

F.4 Register Pushes and Pulls

The M68HC11 has push and pull instructions for A, B, X, and Y, but requires separate 8-bit pushes and pulls of accumulators A and B to stack or unstack the 16-bit D accumulator (the concatenated combination of A:B). The PSHD and PULD instructions allow directly stacking the D accumulator in the expected 16-bit order.

Adding PSHC and PULC improved orthogonality by completing the set of stacking instructions so that any of the CPU registers can be pushed or pulled. These instructions are also useful for preserving the CCR value during a function call subroutine.

F.5 Allocating and Deallocating Stack Space

The LEAS instruction can be used to allocate or deallocate space on the stack for temporary variables:

```
LEAS    -10,S    ;Allocate space for 5 16-bit integers
LEAS    10,S     ;Deallocate space for 5 16-bit ints
```

The (de)allocation can even be combined with a register push or pull as in this example:

```
LDX     8,S+     ;Load return value and deallocate
```

X is loaded with the 16-bit integer value at the top of the stack, and the stack pointer is adjusted up by eight to deallocate space for eight bytes worth of temporary storage. Post-increment indexed addressing is used in this example, but all four combinations of pre/post increment/decrement are available (offsets from -8 to +8 inclusive, from X, Y, or SP). This form of indexing can often be used to get an index (or stack pointer) adjustment for free during an indexed operation (the instruction requires no more code space or cycles than a zero-offset indexed instruction).

F.6 Frame Pointer

In the C language, it is common to have a frame pointer in addition to the CPU stack pointer. The frame is an area of memory within the system stack which is used for parameters and local storage of variables used within a function subroutine. The following is a description of how a frame pointer can be set up and used.

First, parameters (typically values in CPU registers) are pushed onto the system stack prior to using a JSR or CALL to get to the function subroutine. At the beginning of the called subroutine, the frame pointer of the calling program is pushed onto the stack. Typically, an index register, such as X, is used as the frame pointer, so a PSHX instruction would save the frame pointer from the calling program.

Next, the called subroutine establishes a new frame pointer by executing a TFR S,X. Space is allocated for local variables by executing an LEAS -n,S, where n is the number of bytes needed for local variables.

Notice that parameters are at positive offsets from the frame pointer while locals are at negative offsets. In the M68HC11, the indexed addressing mode uses only positive offsets, so the frame pointer always points to the lowest address of any parameter or local. After the function subroutine finishes, calculations are required to restore the stack pointer to the mid-frame position between the locals and the parameters before returning to the calling program. The CPU12 Family only requires execution of TFR X,S to deallocate the local storage and return.

The concept of a frame pointer is supported in the CPU12 Family 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.

F.7 Increment and Decrement Operators

In C, the notation `++i` or `i--` is often used to form loop counters. Within limited constraints, the CPU12 Family loop primitives can be used to speed up the loop count and branch function.

The CPU12 Family includes a set of six basic loop control instructions which decrement, increment, or test a loop count register, and then branch if it is either equal to zero or not equal to zero. The loop count register can be A, B, D, X, Y, or SP. A or B could be used if the loop count fits in an 8-bit char variable; the other choices are all 16-bit registers. The relative offset for the loop branch is a 9-bit signed value, so these instructions can be used with loops as long as 256 bytes.

In some cases, the pre- or post-increment operation can be combined with an indexed instruction to eliminate the cost of the increment operation. This is typically done by post-compile optimization because the indexed instruction that could absorb the increment/decrement operation may not be apparent at compile time.

F.8 Higher Math Functions

In the CPU12 Family, subtle characteristics of higher math operations such as IDIVS and EMUL are arranged so a compiler can handle inputs and outputs more efficiently.

The most apparent case is the IDIVS instruction, which divides two 16-bit signed numbers to produce a 16-bit result. While the same function can be accomplished with the EDIVS instruction (a 32 by 16 divide), doing so is much less efficient because extra steps are required to prepare inputs to the EDIVS, and because EDIVS uses the Y index register. EDIVS uses a 32-bit signed numerator and the C compiler would typically want to use a 16-bit value (the size of an integer data type). The 16-bit C value would need to be sign-extended into the upper 16 bits of the 32-bit EDIVS numerator before the divide operation.

Operand size is also a potential problem in the extended multiply operations but the difficulty can be minimized by putting the results in CPU registers. Having higher precision math instructions is not necessarily a requirement for supporting high-level language because these functions can be performed as library functions. However, if an application requires these functions, the code is much more efficient if the MCU can use native instructions instead of relatively large, slow routines.

F.9 Conditional If Constructs

In the CPU12 Family instruction set, most arithmetic and data manipulation instructions automatically update the condition code register, unlike other architectures that only change condition codes during a few specific compare instructions. The CPU12 Family includes branch instructions that perform conditional branching based on the state of the indicators in the condition codes register. Short branches use a single

byte relative offset that allows branching to a destination within about ± 128 locations from the branch. Long branches use a 16-bit relative offset that allows conditional branching to any location in the 64KB map.

F.10 Case and Switch Statements

Case and switch statements (and computed GOTOs) can use PC-relative indirect addressing to determine which path to take. Depending upon the situation, cases can use either the constant offset variation or the accumulator D offset variation of indirect indexed addressing.

F.11 Pointers

The CPU12 Family supports pointers by allowing direct arithmetic operations on the 16-bit index registers (LEAS, LEAX, and LEAY instructions) and by allowing indexed indirect addressing modes.

F.12 Function Calls

Bank switching is a fairly common way of adapting a CPU with a 16-bit address bus to accommodate more than 64KBs of program memory space. One of the most significant drawbacks of this technique has been the requirement to mask (disable) interrupts while the bank page value was being changed. Another problem is that the physical location of the bank page register can change from one MCU derivative to another (or even due to a change to mapping controls by a user program). In these situations, an operating system program has to keep track of the physical location of the page register. The CPU12 Family addresses both of these problems with the uninterruptible CALL and return-from-call (RTC) instructions.

The CALL instruction is similar to a JSR instruction, except that the programmer supplies a destination page value as part of the instruction. When CALL executes, the old page value is saved on the stack and the new page value is written to the bank page register. Since the CALL instruction is uninterruptible, this eliminates the need to separately mask off interrupts during the context switch.

The CPU12 Family has dedicated signal lines that allow the CPU to access the bank page register without having to use an address in the normal 64KB address space. This eliminates the need for the program to know where the page register is physically located.

The RTC instruction is similar to the RTS instruction, except that RTC uses the byte of information that was saved on the stack by the corresponding CALL instruction to restore the bank page register to its old value. Although a CALL/RTC pair can be used to access any function subroutine regardless of the location of the called routine (on the current bank page or a different page), it is most efficient to access some subroutines with JSR/RTS instructions when the called subroutine is on the current page or in an area of memory that is always visible in the 64KB map regardless of the bank page selection.

Push and pull instructions can be used to stack some or all the CPU registers during a function call. A CPU from the CPU12 Family can push and pull any of the CPU registers A, B, CCR, D, X, Y, or SP.

F.13 Instruction Set Orthogonality

One helpful aspect of the CPU12 Family instruction set, orthogonality, is difficult to quantify in terms of direct benefit to an HLL compiler. Orthogonality refers to the regularity of the instruction set. A

completely orthogonal instruction set would allow any instruction to operate in any addressing mode, would have identical code sizes and execution times for similar operations on different registers, and would include both signed and unsigned versions of all mathematical instructions. Greater regularity of the instruction set makes it possible to implement compilers more efficiently, because operation is more consistent, and fewer special cases must be handled.

Index

A

ABA instruction	84
Abbreviations for system resources	21
ABX instruction	85
ABY instruction	86
Access details	80–83, 431
Accumulator offset indexed addressing mode	42
Accumulator offset indexed indirect addressing mode	41
Accumulators	26
A	25, 37
B	25, 37
D	25, 37
ADCA instruction	87
ADCB instruction	88
ADCD instruction	94
ADCX instruction	95
ADCY instruction	96
ADDA instruction	89
ADDB instruction	90
ADDD instruction	91
Addition instructions	55
ADDR mnemonic	23
Addressing modes	33
Direct	35
Extended	36
Immediate	34
Indexed	26, 37
Inherent	34
Relative	36
ADDX instruction	92
ADDY instruction	93
ANDA instruction	97
ANDB instruction	98
ANDCC instruction	99
ANDX instruction	100
ANDY instruction	101
Arithmetic shift	109, 112, 113, 114
ASL instruction	102
ASLA instruction	103
ASLB instruction	104

ASLD instruction	105
ASLW instruction	106, 258
ASLX instruction	107, 259
ASLY instruction	108, 260
ASR instruction	109
ASRA instruction	110
ASRB instruction	111
ASRW instruction	112
ASRX instruction	113
ASRY instruction	114
Asserted	23
Auto increment	40

B

Background debug mode	74
Instruction	74, 120
Base index register	38–42
BCC instruction	115
BCD instructions	56, 171
BCLR instruction	116
BCS instruction	117
BEQ instruction	118
BGE instruction	119
BGND instruction	74, 120
BGT instruction	121
BHI instruction	122
BHS instruction	123
Binary-coded decimal instructions	56, 171
Bit manipulation instructions	60, 116, 139, 141, 473
Mask operand	43, 116, 136, 138, 139, 141
Multiple addressing modes	43
Bit test instructions	60, 67, 124, 125, 126, 127, 136, 138
BITA instruction	124
BITB instruction	125
BITBY instruction	127
Bit-condition branches	67, 136, 138
BITX instruction	126
BLE instruction	128
BLO instruction	129
BLS instruction	130
BLT instruction	131
BMI instruction	132
BNE instruction	133
Boolean logic instructions	58
AND	97, 98, 99, 100, 101
Complement	157, 158, 159, 160, 161, 162
Exclusive OR	192, 193, 194, 195, 205

Inclusive OR	295, 296, 297, 298, 299
Negate	288, 289, 290, 291, 292, 293
BPL instruction	134
BRA instruction	135
Branch instructions	36, 47–48, 65, 475
Bit-condition	48, 67, 136, 138
Long	48, 66
Loop primitive	49, 68, 460
Offset values	66, 67, 68
Offsets	36
Short	48, 65
Signed	65–67
Simple	65–67
Subroutine	68, 140
Summary of complementary branches	115, 227
Taken/not-taken cases	48, 83
Unary	65–67
Unsigned	65–67
Branch offset	36–37
BRCLR instruction	136
BRN instruction	137
BRSET instruction	138
BSET instruction	139
BSR instruction	47, 140
BTAS instruction	141
Bus cycles	80
BVC instruction	142
BVS instruction	143
Byte moves	54, 273, 274, 275, 276, 277, 278, 279
Byte order in memory	31
Byte-sized instructions	48

C

C	102
C status bit	30, 60, 115, 117
CALL instruction	44–??, 47, 68, 144, 476
Case statements	476
CBA instruction	145
CCR (<i>see Condition codes register</i>)	
Changes in execution flow	46–49
CLC instruction	146
Clear instructions	58
Clear memory	148, 151, 152, 153
Cleared	23
CLI instruction	147
CLR instruction	148
CLRA instruction	149

CLRB instruction	150
CLRW instruction	151
CLRX instruction	152
CLRY instruction	153
CLV instruction	154
COMPA instruction	155
COMPB instruction	156
COM instruction	157
COMA instruction	158
COMB instruction	159
Compare instructions	57
Complement instructions	58
COMW instruction	160
COMX instruction	161
COMY instruction	162
Condition codes instructions	73, 99, 297, 302, 303, 309, 310, 359, 366
Condition codes register	25, 27–30
C status bit	30, 60, 115, 117
H status bit	29, 171
I mask bit	30, 99, 147, 340, 384, 385
Manipulation	73, 99, 297, 340
N status bit	30
S control bit	346
V status bit	30
X mask bit	29, 197, 309, 310, 331, 346, 359, 364, 384, 385
Z status bit	30, 118, 133
Conditional 16-bit read cycle	82, 432
Conditional 8-bit read cycle	82, 432
Conditional 8-bit write cycle	82, 432
Conserving power	73, 346, 378
Constant indirect indexed addressing mode	39
Constant offset indexed addressing mode	38, 39
CPCD instruction	164
CPCS instruction	165
CPCX instruction	166
CPCY instruction	167
CPD instruction	163
CPS instruction	168
CPX instruction	169
CPY instruction	170
Cycle code letters	80, 431
Cycle-by-cycle operation	80, 431

D

DAA instruction	171
DATA mnemonic	23
Data types	30, 473

DBEQ instruction	172, 460
DBNE instruction	173, 460
DEC instruction	174
DECA instruction	175
DECB instruction	176
Decrement instructions	56
DECW instruction	177
DECX instruction	178
DECY instruction	179
Defuzzification	404, 419–421
DES instruction	180
DEX instruction	181
DEY instruction	182
Direct addressing mode	35
Division instructions	59, 475
16-bit fractional	199
16-bit integer	214, 215
32-bit extended	183, 184
Double accumulator	25, 26

E

EDIV instruction	183
EDIVS instruction	184
Effective address	33, 37, 72, 251, 252, 253, 473–475
EMACS instruction	64, 185
EMAXD instruction	186
EMAXM instruction	187, 399
EMIND instruction	188, 399
EMINM instruction	189
EMUL instruction	190
EMULS instruction	191
Enabling maskable interrupts	30, 147
EORA instruction	192
EORB instruction	193, 205
EORX instruction	194
EORY instruction	195
ETBL instruction	64, 196, 399
Even bytes	31
Exceptions	46, 383
Interrupts	385
Maskable interrupts	385
Non-maskable interrupts	385
Priority	384
Processing flow	387
Resets	383, 384–??
Software interrupts	69, 356, 357, 387
Unimplemented opcode trap	383, 384, 386

Vectors	383, 387
Exchange instructions	53, 197, 198
Postbyte encoding	458
Execution cycles	80
Execution time	80
EXG instruction	197, 198
Expanded memory	44, 47, 476
Bank switching	44
Instructions	44, 68, 144, 330
Page registers	44
Subroutines	68, 476
Extended addressing mode	36
Extended division	59
Extension byte	37
External queue reconstruction	391
HCS12 queue reconstruction	395
HCS12 timing detail	392, 393

F

f-cycle (free cycle)	80, 431
FDIV instruction	59, 199
Fractional division	59, 199
Frame pointer	474, 475
Free cycle	80, 431
Fuzzy logic	399–426
Antecedents	403, 425
Consequents	403, 425
Custom programming	422
Defuzzification	62, 404, 419–??
Fuzzification	62, 401, 422
Inference kernel	400, 405
Inputs	425
Instructions	62, 270, 314–??, 379, 399, 406–??
Interrupts	416, 420–421
Knowledge base	400, 403, 425
Membership functions	62, 270, 400, 401, 406–410, 423–424
Outputs	62, 425
Rule evaluation	62, 314–??, 402, 410–419, 425
Rules	400, 403, 425
Sets	400
Tabular membership functions	65, 423
Weighted average	62, 379, 399, 404, 419–??

G

g-cycle (read PPAGE)	80, 431
General purpose accumulators	25

GLDAA instruction	200
GLDAB instruction	201
GLDD instruction	202
GLDS instruction	203
GLDX instruction	204
Global interrupt mask	30, 384
GSTAA instruction	206
GSTAB instruction	207
GSTD instruction	208
GSTS instruction	209
GSTX instruction	210
GSTY instruction	211

H

H status bit	29, 171
Highest priority interrupt	384
High-level language	473–477
Addressing modes	473, 474, 476
Condition codes register	475
Expanded memory	476
Instructions	473
Loop primitives	475
Stack	473, 474

I

I mask bit	30, 99, 147, 340, 384
IBEQ instruction	212, 460
IBNE instruction	213, 460
I-cycle (16-bit read indirect)	80, 431
i-cycle (8-bit read indirect)	80, 431
IDIV instruction	214
IDIVS instruction	215, 475
Immediate addressing mode	34
INC instruction	216
INCA instruction	217
INCB instruction	218
Increment instructions	56
INCW instruction	219
INCX instruction	220
INCY instruction	221
Index calculation instructions	72
Index manipulation instructions	70
Index registers	25, 70, 72, 474
PC (as an index register)	27, 38, 80
SP (as an index register)	27, 38, 80
X	26, 38, 80

Y	26, 38, 80
Indexed addressing modes	26, 37–44, 456, 464–??, 468–??, 470–??, 472–??
16-bit constant indirect	39
16-bit constant offset	39
5-bit constant offset	38
9-bit constant offset	39
Accumulator direct	42
Accumulator offset	41
Auto increment/decrement indexing	40
Base index register	38–42
Extension byte	37
Limitations for BIT and MOV instructions	116, 136, 138, 139, 141, 273, 274, 276, 277, 278, 279, 281, 282, 283, 284, 285, 286
Postbyte	38
Postbyte encoding	37, 456
Inference kernel, fuzzy logic	405
Inherent addressing mode	34
INS instruction	222
Instruction queue	31, 45, 391
Debugging	391
Reconstruction	391–??
Status registers	396
Status signals	391–??
Instruction set	51, 75, 433
Integer division	59, 214–215
Interrupt instructions	69
Interrupts	385–389
Enabling and disabling	29, 30, 147, 340, 385
I mask bit	30, 147, 340, 386
Instructions	69, 70, 147, 331, 340, 356, 357, 367
Low-power stop	74, 346
Maskable	30, 385
Non-maskable	29, 383–384, 385
Recognition	385
Return	29, 30, 70, 331, 386
Service routines	386
Software	69, 356, 357, 387
Stacking order	386
Vectors	386, 387
Wait instruction	74, 378
X mask bit	29, 346, 386
INX instruction	223
INY instruction	224

J

JMP instruction	49, 225
JSR instruction	47, 226

Jump instructions	49, 68
-------------------	--------

K

Knowledge base	400
----------------	-----

L

Label	78
LBCC instruction	227
LBCS instruction	228
LBEQ instruction	229
LBGE instruction	230
LBGT instruction	231
LBHI instruction	232
LBHS instruction	233
LBLE instruction	234
LBLO instruction	235
LBLS instruction	236
LBLT instruction	237
LBMI instruction	238
LBNE instruction	239
LBPL instruction	240
LBRA instruction	241
LBRN instruction	242
LBVC instruction	243
LBVS instruction	244
LDAA instruction	245
LDAB instruction	246
LDD instruction	247
LDS instruction	248
LDX instruction	249
LDY instruction	250
LEAS instruction	251, 474, 476
Least significant byte	23
Least significant word	23
LEAX instruction	252, 476
LEAY instruction	253, 476
Legal label	78
Literal expression	78
Load instructions	51
Logic level one	23
Logic level zero	23
Loop primitive instructions	49, 68, 460, 475
Offset values	68
Postbyte encoding	460
Low-power stop	73, 74, 346
LSL instruction	60, 254

LSLA instruction	255
LSLB instruction	256
LSLD instruction	257
LSR instruction	261
LSRA instruction	262
LSRB instruction	263
LSRD instruction	264
LSRW instruction	265
LSRX instruction	266
LSRY instruction	267

M

Maskable interrupts	30, 385
MAXA instruction	268
Maximum instructions	63
16-bit	186, 187
8-bit	268, 269
MAXM instruction	269, 399
MEM instruction	62, 270, 399, 406–410
Membership functions	400, 406–410
Memory and addressing symbols	22
MINA instruction	271, 399
Minimum instructions	63
16-bit	188, 189
8-bit	271, 272
MINM instruction	272
Misaligned instructions	48
Mnemonic	75
Most significant byte	23
Most significant word	23
MOVB instruction	273, 274, 275, 276, 277, 278, 279
Move instructions	54, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286
Destination	42
Multiple addressing modes	42
PC relative addressing	42
Reference index register	43
Source	42
MOVW instruction	280, 281, 282, 283, 284, 285, 286
MUL instruction	287
Multiple addressing modes	
Bit manipulation instructions	43
Move instructions	42
Multiplication instructions	59
16-bit	190, 191
8-bit	287
Multiply and accumulate instructions	64, 185, 379, 426

N

N status bit	30
n-cycle (write PPAGE)	81, 431
NEG instruction	288
NEGA instruction	289
Negate instructions	58
Negated	23
Negative integers	31
NEGB instruction	290
NEGW instruction	291
NEGX instruction	292
NEGY instruction	293
Non-maskable interrupts	29, 384, 385
NOP instruction	74, 294
Notation	
Branch taken/not taken	83, 432
Changes in CCR bits	77
Cycle-by-cycle operation	80
Memory and addressing	22
Object code	77
Operators	22, 430
Source forms	78
System resources	21
Null operation instruction	74, 294
Numeric range of branch offsets	36, 66–68

O

Object code notation	77
O-cycle (optional program word fetch)	48, 81, 431
Odd bytes	31
Offset	
Branch	36–37
Index	37–40
Opcode map ??–	453, 453–454, ??–455
Operators	22, 430
Optional cycles	48, 81, 431
ORAA instruction	295
ORAB instruction	296
ORCC instruction	297
Orthogonality	476
ORX instruction	298
ORY instruction	299

P

Page 2 prebyte	48, 81, 453, 454, 455
P-cycle (program word fetch)	81, 431

Pipeline	31
Pointer calculation instructions	72, 251, 252, 253
Pointers	476
Postbyte encoding	
Exchange instructions	197, 198, 458
Indexed addressing instructions	38
Indexed addressing modes	38, 456
Loop primitive instructions	460
Transfer instructions	342, 364, 458
Post-decrement indexed addressing mode	40
Post-increment indexed addressing mode	40
Power conservation	73, 346, 378
Prebyte	48, 81, 453, 454, 455
Pre-decrement indexed addressing mode	40
Pre-increment indexed addressing mode	40
Priority, exception	384
Program counter	25, 27, 37, 120
Program word access cycle	81, 431
Programming model	25, 469
Pseudo-non-maskable interrupt	384
PSHA instruction	300
PSHB instruction	301
PSHC instruction	302
PSHCW instruction	303
PSHD instruction	304, 474
PSHX instruction	305
PSHY instruction	306
PULA instruction	307
PULB instruction	308
PULC instruction	309, 474
PULCW instruction	310
PULD instruction	311, 474
Pull instructions	476
PULX instruction	312
PULY instruction	313
Push instructions	476

Q

Queue reconstruction	391
HCS12 queue reconstruction	395
HCS12 timing detail	392, 393

R

R-cycle (16-bit data read)	81, 431
r-cycle (8-bit data read)	81, 431
Read 16-bit data cycle	81, 431

Read 8-bit data cycle	81, 431
Read indirect pointer cycle	80, 431
Read indirect PPAGE value cycle	80, 431
Read PPAGE cycle	80, 431
Register designators	78
Relative addressing mode	36
Relative offset	36
Resets	383, 384
Return from call	330
Return from interrupt	331
Return from subroutine	332
REV instruction	62, 314–315, 399, 402, 410–415, 425
REW instruction	62, 316–??, 399, 402, 415–419, 425
ROL instruction	318
ROLA instruction	319
ROLB instruction	320
ROLW instruction	321
ROLX instruction	322
ROLY instruction	323
ROR instruction	324
RORA instruction	325
RORB instruction	326
RORW instruction	327
RORX instruction	328
RORY instruction	329
Rotate instructions	60
RTC instruction	44, 47, 68, 330, 476
RTI instruction	30, 70, 331, 386
RTS instruction	47, 332

S

S control bit	346
SBA instruction	333
SBCA instruction	334
SBCB instruction	335
SBCD instruction	336
SBCX instruction	337
SBCY instruction	338
S-cycle (16-bit stack write)	81, 432
s-cycle (8-bit stack write)	81, 432
SEC instruction	339
SEI instruction	340
Set	23
Setting memory bits	139, 141
SEV instruction	341
SEX instruction	53, 342
Shift instructions	60

Arithmetic	109, 112, 113, 114
Sign extension instruction	53, 342, 473
Signed branches	65–67
Signed integers	31
Signed multiplication	59
Simple branches	65–67
Software interrupts	356, 357
Source code compatibility	463, 465, 469, 471
Source form notation	78
STAA instruction	343
STAB instruction	344
Stack	26
Stack 16-bit data cycle	81, 432
Stack 8-bit data cycle	81, 432
Stack operation instructions	71
Stack pointer	25, 26, 37, 473
Initialization	27
Manipulation	71
Stacking order	386
Stack pointer instructions	71, 473
Standard CPU12 address space	31
STD instruction	345
STOP continue	346
STOP disable	29, 346
STOP instruction	73, 346
Store instructions	51
STS instruction	348
STX instruction	349
STY instruction	350
SUBA instruction	351
SUBB instruction	352
SUBD instruction	353
Subroutine instructions	68
Subroutines	47, 476
Expanded memory	47, 68, 144, 330, 476
Instructions	68, 140, 144, 226, 476
Return	330, 332
Subtraction instructions	55
SUBX instruction	354
SUBY instruction	355
SWI instruction	69, 356, 357, 387
Switch statements	476
Symbols and notation	21, 429

T

TAB instruction	358
Table interpolation instructions	64, 196, 362

Tabular membership functions	423–424
TAP instruction	359
TBA instruction	360
TBEQ instruction	361, 460
TBL instruction	64, 362, 399, 423–424
TBNE instruction	363, 460
T-cycle (16-bit conditional read)	82, 432
t-cycle (8-bit conditional read)	82, 432
Termination of interrupt service routines	70, 331, 386
Termination of subroutines	330, 332
Test instructions	57
TFR instruction	364, 365
TPA instruction	366
Transfer instructions	53
Postbyte encoding	458
TRAP instruction	69, 367, 386, 454
TST instruction	368
TSTA instruction	369
TSTB instruction	370
TSTW instruction	371
TSTX instruction	372
TSTY instruction	373
TSX instruction	374
TSY instruction	375
Twos-complement form	31
TXS instruction	376
Types of instructions	
Addition and Subtraction	55
Background and null	74
Binary-coded decimal	56
Bit test and manipulation	60
Boolean logic	58
Branch	65
Clear, complement, and negate	58
Compare and test	57
Condition code	73
Decrement and increment	56
Fuzzy logic	62
Index manipulation	70
Interrupt	69
Jump and subroutine	68
Load and store	51
Loop primitives	68
Maximum and minimum	63
Move	54
Multiplication and division	59
Multiply and accumulate	64
Pointer and index calculation	72
Shift and rotate	60

Sign extension	53
Stacking	71
Stop and wait	73
Table interpolation	64
Transfer and exchange	53
TYS instruction	377

U

U-cycle (16-bit stack read)	82, 432
u-cycle (8-bit stack read)	82, 432
Unary branches	65–67
Unimplemented opcode trap	69, 367, 383, 384, 454
Unsigned branches	65–67
Unsigned multiplication	59
Unstack 16-bit data cycle	82, 432
Unstack 8-bit data cycle	82, 432
Unweighted rule evaluation	314–315, 402, 410–414, 425

V

V status bit	30, 73
V-cycle (vector fetch)	82, 432
Vector fetch cycle	82, 432
Vectors, exception	383, 387

W

WAI instruction	73, 378
Wait instruction	73, 378
WAV instruction	62, 379, 399, 404, 419–421
HCS12	422
wavr pseudo-instruction	420–421
HCS12	422
W-cycle (16-bit data write)	82, 432
w-cycle (8-bit data write)	82, 432
Weighted average	379
Weighted rule evaluation	316–??, 402, 410–412, 415–419, 425
Word moves	55, 280, 281, 282, 283, 284, 285, 286
Write 16-bit data cycle	82, 432
Write 8-bit data cycle	82, 432
Write PPAGE cycle	81, 431

X

X mask bit	29, 197, 309, 310, 331, 346, 359, 364
x-cycle (8-bit conditional write)	82, 432
XGDX instruction	380

XGDY instruction	381
------------------------	-----

Z

Z status bit	30, 118, 133
Zero-page addressing	35



How to Reach Us:

Home Page:

www.freescale.com

E-mail:

support@freescale.com

USA/Europe or Locations Not Listed:

Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:

Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:

Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:

Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:

Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© Freescale Semiconductor, Inc. 2004. All rights reserved.