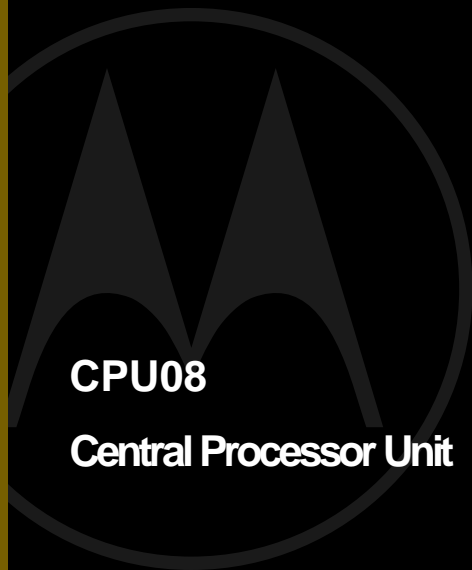


CPU08RM/AD  
REV 3



**CPU08**

**Central Processor Unit**

**Reference Manual**

 **Digital DNA**<sup>™</sup>  
from Motorola



# CPU08

## Central Processor Unit

### Reference Manual

---

---

*Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.*



## List of Sections

Section 1. General Description . . . . .	19
Section 2. Architecture . . . . .	23
Section 3. Resets and Interrupts . . . . .	37
Section 4. Addressing Modes . . . . .	55
Section 5. Instruction Set . . . . .	89
Section 6. Instruction Set Examples . . . . .	189
Glossary . . . . .	223
Index . . . . .	237

# List of Sections

## Table of Contents

### Section 1. General Description

1.1	Contents . . . . .	19
1.2	Introduction . . . . .	19
1.3	Features . . . . .	20
1.4	Programming Model . . . . .	20
1.5	Memory Space . . . . .	21
1.6	Addressing Modes . . . . .	21
1.7	Arithmetic Instructions . . . . .	22
1.8	Binary-Coded Decimal (BCD) Arithmetic Support . . . . .	22
1.9	High-Level Language Support . . . . .	22
1.10	Low-Power Modes . . . . .	22

### Section 2. Architecture

2.1	Contents . . . . .	23
2.2	Introduction . . . . .	23
2.3	CPU08 Registers . . . . .	24
2.3.1	Accumulator . . . . .	25
2.3.2	Index Register . . . . .	25
2.3.3	Stack Pointer . . . . .	26
2.3.4	Program Counter . . . . .	27
2.3.5	Condition Code Register . . . . .	28

2.4	CPU08 Functional Description . . . . .	30
2.4.1	Internal Timing . . . . .	31
2.4.2	Control Unit . . . . .	32
2.4.3	Execution Unit . . . . .	33
2.4.4	Instruction Execution . . . . .	33

## Section 3. Resets and Interrupts

3.1	Contents . . . . .	37
3.2	Introduction . . . . .	38
3.3	Elements of Reset and Interrupt Processing . . . . .	39
3.3.1	Recognition . . . . .	39
3.3.2	Stacking . . . . .	40
3.3.3	Arbitration . . . . .	41
3.3.4	Masking . . . . .	43
3.3.5	Returning to Calling Program . . . . .	45
3.4	Reset Processing . . . . .	46
3.4.1	Initial Conditions Established . . . . .	47
3.4.2	CPU . . . . .	47
3.4.3	Operating Mode Selection . . . . .	47
3.4.4	Reset Sources . . . . .	48
3.4.5	External Reset . . . . .	49
3.4.6	Active Reset from an Internal Source. . . . .	49
3.5	Interrupt Processing . . . . .	49
3.5.1	Interrupt Sources and Priority. . . . .	51
3.5.2	Interrupts in Stop and Wait Modes . . . . .	52
3.5.3	Nesting of Multiple Interrupts . . . . .	52
3.5.4	Allocating Scratch Space on the Stack . . . . .	53



## Section 4. Addressing Modes

4.2	Introduction . . . . .	55
4.3	Addressing Modes . . . . .	56
4.3.1	Inherent . . . . .	57
4.3.2	Immediate . . . . .	59
4.3.3	Direct . . . . .	61
4.3.4	Extended . . . . .	63
4.3.5	Indexed, No Offset . . . . .	65
4.3.6	Indexed, 8-Bit Offset . . . . .	65
4.3.7	Indexed, 16-Bit Offset . . . . .	66
4.3.8	Stack Pointer, 8-Bit Offset . . . . .	68
4.3.9	Stack Pointer, 16-Bit Offset . . . . .	68
4.3.10	Relative . . . . .	71
4.3.11	Memory-to-Memory Immediate to Direct . . . . .	73
4.3.12	Memory-to-Memory Direct to Direct . . . . .	73
4.3.13	Memory-to-Memory Indexed to Direct with Post Increment . . . . .	74
4.3.14	Memory-to-Memory Direct to Indexed with Post Increment . . . . .	76
4.3.15	Indexed with Post Increment . . . . .	77
4.3.16	Indexed, 8-Bit Offset with Post Increment . . . . .	78
4.4	Instruction Set Summary . . . . .	79
4.5	Opcode Map . . . . .	87

## Section 5. Instruction Set

5.1	Contents . . . . .	89
5.2	Introduction . . . . .	92
5.3	Nomenclature . . . . .	92
5.4	Convention Definitions . . . . .	96

## Table of Contents

5.5	Instruction Set	96
	ADC Add with Carry	97
	ADD Add without Carry	98
	AIS Add Immediate Value (Signed) to Stack Pointer	99
	AIX Add Immediate Value (Signed) to Index Register	100
	AND Logical AND	101
	ASL Arithmetic Shift Left	102
	ASR Arithmetic Shift Right	103
	BCC Branch if Carry Bit Clear	104
	BCLR <i>n</i> Clear Bit <i>n</i> in Memory	105
	BCS Branch if Carry Bit Set	106
	BEQ Branch if Equal	107
	BGE Branch if Greater Than or Equal To	108
	BGT Branch if Greater Than	109
	BHCC Branch if Half Carry Bit Clear	110
	BHCS Branch if Half Carry Bit Set	111
	BHI Branch if Higher	112
	BHS Branch if Higher or Same	113
	BIH Branch if $\overline{\text{IRQ}}$ Pin High	114
	BIL Branch if $\overline{\text{IRQ}}$ Pin Low	115
	BIT Bit Test	116
	BLE Branch if Less Than or Equal To	117
	BLO Branch if Lower	118
	BLS Branch if Lower or Same	119
	BLT Branch if Less Than	120
	BMC Branch if Interrupt Mask Clear	121
	BMI Branch if Minus	122
	BMS Branch if Interrupt Mask Set	123
	BNE Branch if Not Equal	124
	BPL Branch if Plus	125
	BRA Branch Always	126
	BRA Branch Always	127
	BRCLR <i>n</i> Branch if Bit <i>n</i> in Memory Clear	128
	BRN Branch Never	129
	BRSET <i>n</i> Branch if Bit <i>n</i> in Memory Set	130
	BSET <i>n</i> Set Bit <i>n</i> in Memory	131

BSR	Branch to Subroutine . . . . .	132
CBEQ	Compare and Branch if Equal . . . . .	133
CLC	Clear Carry Bit . . . . .	134
CLI	Clear Interrupt Mask Bit . . . . .	135
CLR	Clear . . . . .	136
CMP	Compare Accumulator with Memory . . . . .	137
COM	Complement (One's Complement) . . . . .	138
CPHX	Compare Index Register with Memory . . . . .	139
CPX	Compare X (Index Register Low) with Memory . . . . .	140
DAA	Decimal Adjust Accumulator . . . . .	141
DBNZ	Decrement and Branch if Not Zero . . . . .	143
DEC	Decrement . . . . .	144
DIV	Divide . . . . .	145
EOR	Exclusive-OR Memory with Accumulator . . . . .	146
INC	Increment . . . . .	147
JMP	Jump . . . . .	148
JSR	Jump to Subroutine . . . . .	149
LDA	Load Accumulator from Memory . . . . .	150
LDHX	Load Index Register from Memory . . . . .	151
LDX	Load X (Index Register Low) from Memory . . . . .	152
LSL	Logical Shift Left . . . . .	153
LSR	Logical Shift Right . . . . .	154
MOV	Move . . . . .	155
MUL	Unsigned Multiply . . . . .	156
NEG	Negate (Two's Complement) . . . . .	157
NOP	No Operation . . . . .	158
NSA	Nibble Swap Accumulator . . . . .	159
ORA	Inclusive-OR Accumulator and Memory . . . . .	160
PSHA	Push Accumulator onto Stack . . . . .	161
PSHH	Push H (Index Register High) onto Stack . . . . .	162
PSHX	Push X (Index Register Low) onto Stack . . . . .	163
PULA	Pull Accumulator from Stack . . . . .	164
PULH	Pull H (Index Register High) from Stack . . . . .	165
PULX	Pull X (Index Register Low) from Stack . . . . .	166
ROL	Rotate Left through Carry . . . . .	167
ROR	Rotate Right through Carry . . . . .	168
RSP	Reset Stack Pointer . . . . .	169

## Table of Contents

RTI	Return from Interrupt . . . . .	170
RTS	Return from Subroutine . . . . .	171
SBC	Subtract with Carry . . . . .	172
SEC	Set Carry Bit . . . . .	173
SEI	Set Interrupt Mask Bit . . . . .	174
STA	Store Accumulator in Memory . . . . .	175
STHX	Store Index Register . . . . .	176
STOP	Enable $\overline{\text{IRQ}}$ Pin, Stop Oscillator . . . . .	177
STX	Store X (Index Register Low) in Memory . . . . .	178
SUB	Subtract . . . . .	179
SWI	Software Interrupt . . . . .	180
TAP	Transfer Accumulator to Processor Status Byte . . . . .	181
TAX	Transfer Accumulator to X (Index Register Low) . . . . .	182
TPA	Transfer Processor Status Byte to Accumulator . . . . .	183
TST	Test for Negative or Zero . . . . .	184
TSX	Transfer Stack Pointer to Index Register . . . . .	185
TXA	Transfer X (Index Register Low) to Accumulator . . . . .	186
TXS	Transfer Index Register to Stack Pointer . . . . .	187
WAIT	Enable Interrupts; Stop Processor . . . . .	188

## Section 6. Instruction Set Examples

6.1	Contents . . . . .	189
6.2	Introduction . . . . .	190
6.3	M68HC08 Unique Instructions . . . . .	190
6.4	Code Examples . . . . .	191
AIS	Add Immediate Value (Signed) to Stack Pointer . . . . .	192
AIX	Add Immediate Value (Signed) to Index Register . . . . .	194
BGE	Branch if Greater Than or Equal To . . . . .	195
BGT	Branch if Greater Than . . . . .	196

BLE	Branch if Less Than or Equal To . . . . .	197
BLT	Branch if Less Than . . . . .	198
CBEQ	Compare and Branch if Equal . . . . .	199
CBEQA	Compare A with Immediate . . . . .	200
CBEQX	Compare X with Immediate . . . . .	201
CLRH	Clear H (Index Register High) . . . . .	202
CPHX	Compare Index Register with Memory . . . . .	203
DAA	Decimal Adjust Accumulator . . . . .	204
DBNZ	Decrement and Branch if Not Zero . . . . .	205
DIV	Divide . . . . .	206
LDHX	Load Index Register with Memory . . . . .	209
MOV	Move . . . . .	210
NSA	Nibble Swap Accumulator . . . . .	211
PSHA	Push Accumulator onto Stack . . . . .	212
PSHH	Push H (Index Register High) onto Stack . . . . .	213
PSHX	Push X (Index Register Low) onto Stack . . . . .	214
PULA	Pull Accumulator from Stack . . . . .	215
PULH	Pull H (Index Register High) from Stack . . . . .	216
PULX	Pull X (Index Register Low) from Stack . . . . .	217
STHX	Store Index Register . . . . .	218
TAP	Transfer Accumulator to Condition Code Register . . . . .	219
TPA	Transfer Condition Code Register to Accumulator . . . . .	220
TSX	Transfer Stack Pointer to Index Register . . . . .	221
TXS	Transfer Index Register to Stack Pointer . . . . .	222

## Glossary

Glossary . . . . .	223
--------------------	-----

## Index

Index . . . . .	237
-----------------	-----

# Table of Contents

## List of Figures

Figure	Title	Page
2-1	CPU08 Programming Model . . . . .	24
2-2	Accumulator (A) . . . . .	25
2-3	Index Register (H:X) . . . . .	25
2-4	Stack Pointer (SP) . . . . .	26
2-5	Program Counter (PC) . . . . .	27
2-6	Condition Code Register (CCR) . . . . .	28
2-7	CPU08 Block Diagram . . . . .	30
2-8	Internal Timing Detail . . . . .	31
2-9	Control Unit Timing . . . . .	33
2-10	Instruction Boundaries . . . . .	34
2-11	Instruction Execution Timing Diagram . . . . .	35
3-1	Interrupt Stack Frame . . . . .	40
3-2	H Register Storage . . . . .	41
3-3	Interrupt Processing Flow and Timing . . . . .	42
3-4	Interrupt Recognition Example 1 . . . . .	43
3-5	Interrupt Recognition Example 2 . . . . .	44
3-6	Interrupt Recognition Example 3 . . . . .	44
3-7	Exiting Reset. . . . .	46

## List of Figures



## List of Tables

Table	Title	Page
3-1	Mode Selection . . . . .	48
3-2	M68HC08 Vectors . . . . .	51
4-1	Inherent Addressing Instructions . . . . .	58
4-2	Immediate Addressing Instructions . . . . .	60
4-3	Direct Addressing Instructions . . . . .	62
4-4	Extended Addressing Instructions . . . . .	64
4-5	Indexed Addressing Instructions . . . . .	67
4-6	Stack Pointer Addressing Instructions . . . . .	70
4-7	Relative Addressing Instructions . . . . .	72
4-8	Memory-to-Memory Move Instructions . . . . .	77
4-9	Indexed and Indexed, 8-Bit Offset with Post Increment Instructions . . . . .	78
4-10	Instruction Set Summary . . . . .	79
4-11	Opcode Map . . . . .	88
5-1	Branch Instruction Summary . . . . .	127
5-2	DAA Function Summary . . . . .	142



## Section 1. General Description

### 1.1 Contents

1.2	Introduction . . . . .	19
1.3	Features . . . . .	20
1.4	Programming Model . . . . .	20
1.5	Memory Space . . . . .	21
1.6	Addressing Modes . . . . .	21
1.7	Arithmetic Instructions . . . . .	22
1.8	Binary-Coded Decimal (BCD) Arithmetic Support . . . . .	22
1.9	High-Level Language Support . . . . .	22
1.10	Low-Power Modes . . . . .	22

### 1.2 Introduction

The CPU08 is the central processor unit (CPU) of the Motorola M68HC08 Family of microcontroller units (MCU). The fully object code compatible CPU08 offers M68HC05 users increased performance with no loss of time or software investment in their M68HC05-based applications. The CPU08 also appeals to users of other MCU architectures who need the CPU08 combination of speed, low power, processing capabilities, and cost effectiveness.

### 1.3 Features

CPU08 features include:

- Full object-code compatibility with M68HC05 Family
- 16-bit stack pointer with stack manipulation instructions
- 16-bit index register (H:X) with high-byte and low-byte manipulation instructions
- 8-MHz CPU standard bus frequency
- 64-Kbyte program/data memory space
- 16 addressing modes
- 78 new opcodes
- Memory-to-memory data moves without using accumulator
- Fast 8-bit by 8-bit multiply and 16-bit by 8-bit divide instructions
- Enhanced binary-coded decimal (BCD) data handling
- Expandable internal bus definition for extension of addressing range beyond 64 Kbytes
- Flexible internal bus definition to accommodate CPU performance-enhancing peripherals such as a direct memory access (DMA) controller
- Low-power stop and wait modes

### 1.4 Programming Model

The CPU08 programming model consists of:

- 8-bit accumulator
- 16-bit index register
- 16-bit stack pointer
- 16-bit program counter
- 8-bit condition code register

See [Figure 2-1. CPU08 Programming Model](#).

## 1.5 Memory Space

Program memory space and data memory space are contiguous over a 64-Kbyte addressing range. Addition of a page-switching peripheral allows extension of the addressing range beyond 64 Kbytes.

## 1.6 Addressing Modes

The CPU08 has a total of 16 addressing modes:

- Inherent
- Immediate
- Direct
- Extended
- Indexed
  - No offset
  - No offset, post increment
  - 8-bit offset
  - 8-bit offset, post increment
  - 16-bit offset
- Stack pointer
  - 8-bit offset
  - 16-bit offset
- Relative
- Memory-to-memory (four modes)

Refer to [Section 4. Addressing Modes](#) for a detailed description of the CPU08 addressing modes.

### 1.7 Arithmetic Instructions

The CPU08 arithmetic functions include:

- Addition with and without carry
- Subtraction with and without carry
- A fast 16-bit by 8-bit unsigned division
- A fast 8-bit by 8-bit unsigned multiply

### 1.8 Binary-Coded Decimal (BCD) Arithmetic Support

To support binary-coded decimal (BCD) arithmetic applications, the CPU08 has a decimal adjust accumulator (DAA) instruction and a nibble swap accumulator (NSA) instruction.

### 1.9 High-Level Language Support

The 16-bit index register, 16-bit stack pointer, 8-bit signed branch instructions, and associated instructions are designed to support the efficient use of high-level language (HLL) compilers with the CPU08.

### 1.10 Low-Power Modes

The WAIT and STOP instructions reduce the power consumption of the CPU08-based MCU. The WAIT instruction stops only the CPU clock and therefore uses more power than the STOP instruction, which stops both the CPU clock and the peripheral clocks. In most modules, clocks can be shut off in wait mode.

## Section 2. Architecture

### 2.1 Contents

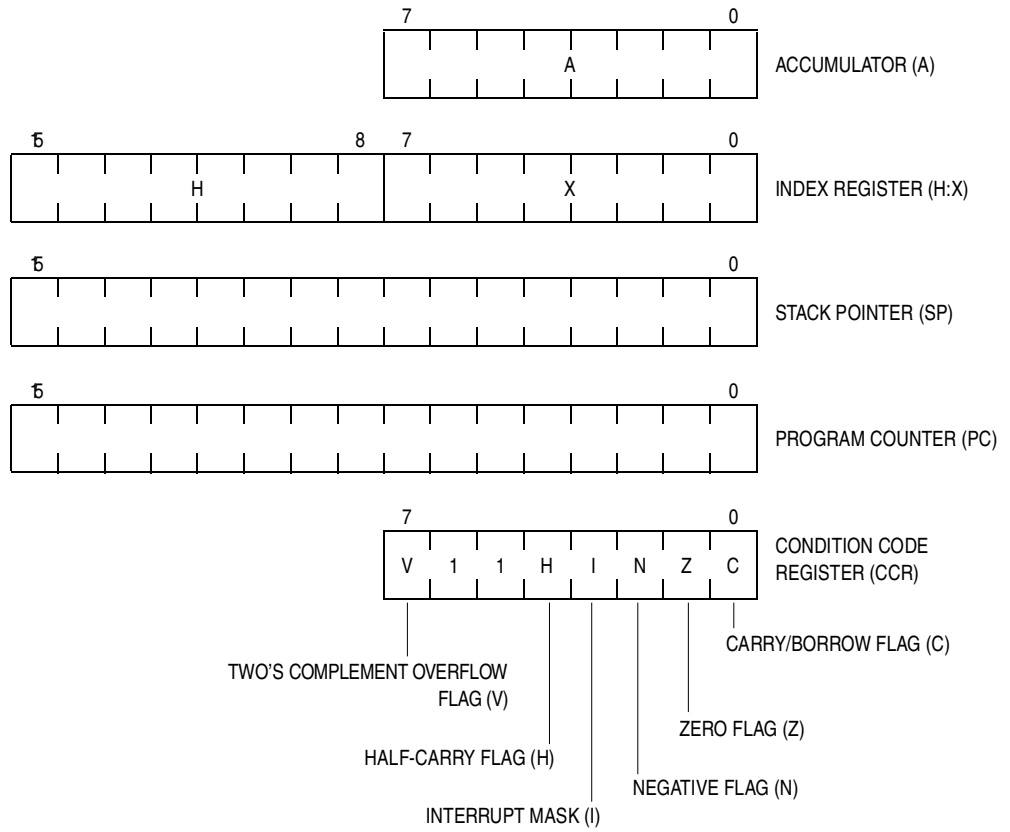
2.2	Introduction . . . . .	23
2.3	CPU08 Registers . . . . .	24
2.3.1	Accumulator . . . . .	25
2.3.2	Index Register . . . . .	25
2.3.3	Stack Pointer . . . . .	26
2.3.4	Program Counter . . . . .	27
2.3.5	Condition Code Register . . . . .	28
2.4	CPU08 Functional Description . . . . .	30
2.4.1	Internal Timing . . . . .	31
2.4.2	Control Unit . . . . .	32
2.4.3	Execution Unit . . . . .	33
2.4.4	Instruction Execution . . . . .	33

### 2.2 Introduction

This section describes the CPU08 registers.

## 2.3 CPU08 Registers

**Figure 2-1** shows the five CPU08 registers. The CPU08 registers are not part of the memory map.

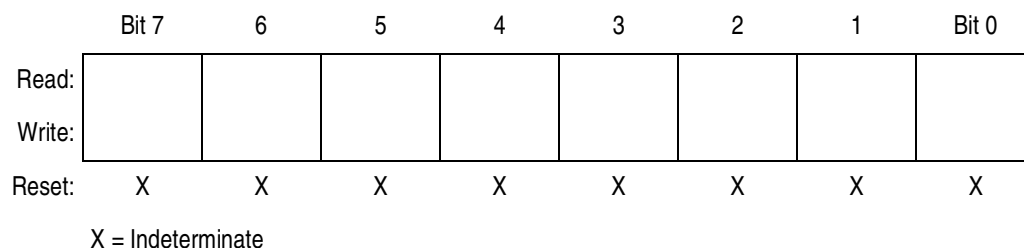


**Figure 2-1. CPU08 Programming Model**



### 2.3.1 Accumulator

The accumulator (A) shown in [Figure 2-2](#) is a general-purpose 8-bit register. The central processor unit (CPU) uses the accumulator to hold operands and results of arithmetic and non-arithmetic operations.

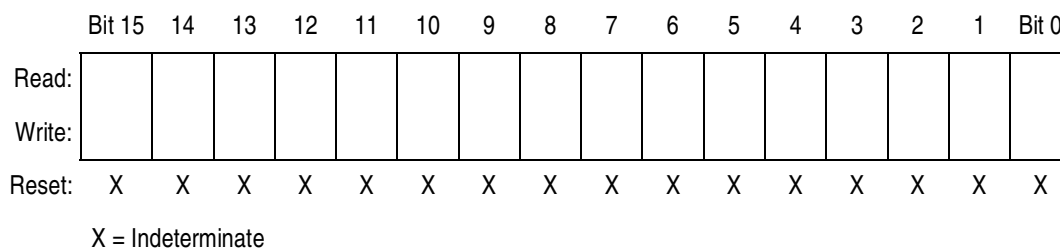


**Figure 2-2. Accumulator (A)**

### 2.3.2 Index Register

The 16-bit index register (H:X) shown in [Figure 2-3](#) allows the user to index or address a 64-Kbyte memory space. The concatenated 16-bit register is called H:X. The upper byte of the index register is called H. The lower byte of the index register is called X. H is cleared by reset. When H = 0 and no instructions that affect H are used, H:X is functionally identical to the IX register of the M6805 Family.

In the indexed addressing modes, the CPU uses the contents of H:X to determine the effective address of the operand. H:X can also serve as a temporary data storage location. See [4.3.5 Indexed, No Offset](#); [4.3.6 Indexed, 8-Bit Offset](#); and [4.3.7 Indexed, 16-Bit Offset](#).



**Figure 2-3. Index Register (H:X)**

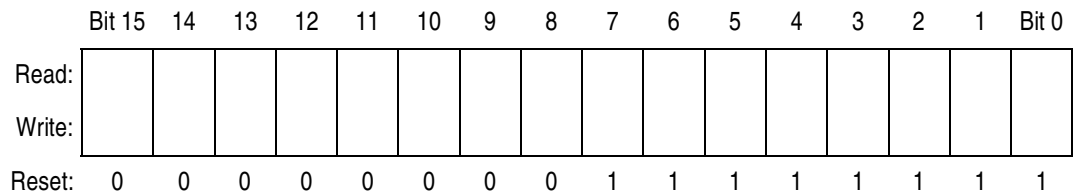
## 2.3.3 Stack Pointer

The stack pointer (SP) shown in [Figure 2-4](#) is a 16-bit register that contains the address of the next location on the stack. During a reset, the stack pointer is preset to \$00FF to provide compatibility with the M6805 Family.

**NOTE:** *The reset stack pointer (RSP) instruction sets the least significant byte to \$FF and does not affect the most significant byte.*

The address in the stack pointer decrements as data is pushed onto the stack and increments as data is pulled from the stack. The SP always points to the next available (empty) byte on the stack.

The CPU08 has stack pointer 8- and 16-bit offset addressing modes that allow the stack pointer to be used as an index register to access temporary variables on the stack. The CPU uses the contents in the SP register to determine the effective address of the operand. See [4.3.8 Stack Pointer, 8-Bit Offset](#) and [4.3.9 Stack Pointer, 16-Bit Offset](#).



**Figure 2-4. Stack Pointer (SP)**

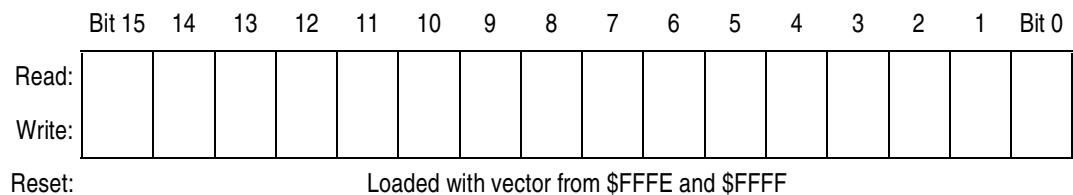
**NOTE:** *Although preset to \$00FF, the location of the stack is arbitrary and may be relocated by the user to anywhere that random-access memory (RAM) resides within the memory map. Moving the SP out of page 0 (\$0000 to \$00FF) will free up address space, which may be accessed using the efficient direct addressing mode.*

### 2.3.4 Program Counter

The program counter (PC) shown in **Figure 2-5** is a 16-bit register that contains the address of the next instruction or operand to be fetched.

Normally, the address in the program counter automatically increments to the next sequential memory location every time an instruction or operand is fetched. Jump, branch, and interrupt operations load the program counter with an address other than that of the next sequential location.

During reset, the PC is loaded with the contents of the reset vector located at \$FFFE and \$FFFF. This represents the address of the first instruction to be executed after the reset state is exited.



**Figure 2-5. Program Counter (PC)**

## 2.3.5 Condition Code Register

The 8-bit condition code register (CCR) shown in [Figure 2-6](#) contains the interrupt mask and five flags that indicate the results of the instruction just executed. Bits five and six are permanently set to logic 1.

	Bit 7	6	5	4	3	2	1	Bit 0
Read:	V	1	1	H	I	N	Z	C
Write:								
Reset:	X	1	1	X	1	X	X	X

X = Indeterminate

**Figure 2-6. Condition Code Register (CCR)**

### V — Overflow Flag

The CPU sets the overflow flag when a two's complement overflow occurs as a result of an operation. The overflow flag bit is utilized by the signed branch instructions:

- Branch if greater than, BGT
- Branch if greater than or equal to, BGE
- Branch if less than or equal to, BLE
- Branch if less than, BLT

This bit is set by these instructions, although its resulting value holds no meaning:

- Arithmetic shift left, ASL
- Arithmetic shift right, ASR
- Logical shift left, LSL
- Logical shift right, LSR
- Rotate left through carry, ROL
- Rotate right through carry, ROR

### H — Half-Carry Flag

The CPU sets the half-carry flag when a carry occurs between bits 3 and 4 of the accumulator during an add-without-carry (ADD) or add-with-carry (ADC) operation. The half-carry flag is required for

binary-coded (BCD) arithmetic operations. The decimal adjust accumulator (DAA) instruction uses the state of the H and C flags to determine the appropriate correction factor.

#### I — Interrupt Mask

When the interrupt mask is set, all interrupts are disabled. Interrupts are enabled when the interrupt mask is cleared. When an interrupt occurs, the interrupt mask is automatically set after the CPU registers are saved on the stack, but before the interrupt vector is fetched.

**NOTE:** *To maintain M6805 compatibility, the H register is not stacked automatically. If the interrupt service routine uses X (and H is not clear), then the user must stack and unstack H using the push H (index register high) onto stack (PSHH) and pull H (index register high) from stack (PULH) instructions within the interrupt service routine.*

If an interrupt occurs while the interrupt mask is set, the interrupt is latched. Interrupts in order of priority are serviced as soon as the I bit is cleared.

A return-from-interrupt (RTI) instruction pulls the CPU registers from the stack, restoring the interrupt mask to its cleared state. After any reset, the interrupt mask is set and can only be cleared by a software instruction. See [Section 3. Resets and Interrupts](#).

#### N — Negative Flag

The CPU sets the negative flag when an arithmetic operation, logical operation, or data manipulation produces a negative result.

#### Z — Zero Flag

The CPU sets the zero flag when an arithmetic operation, logical operation, or data manipulation produces a result of \$00.

#### C — Carry/Borrow Flag

The CPU sets the carry/borrow flag when an addition operation produces a carry out of bit 7 of the accumulator or when a subtraction operation requires a borrow. Some logical operations and data manipulation instructions also clear or set the carry/borrow flag (as in bit test and branch instructions and shifts and rotates).

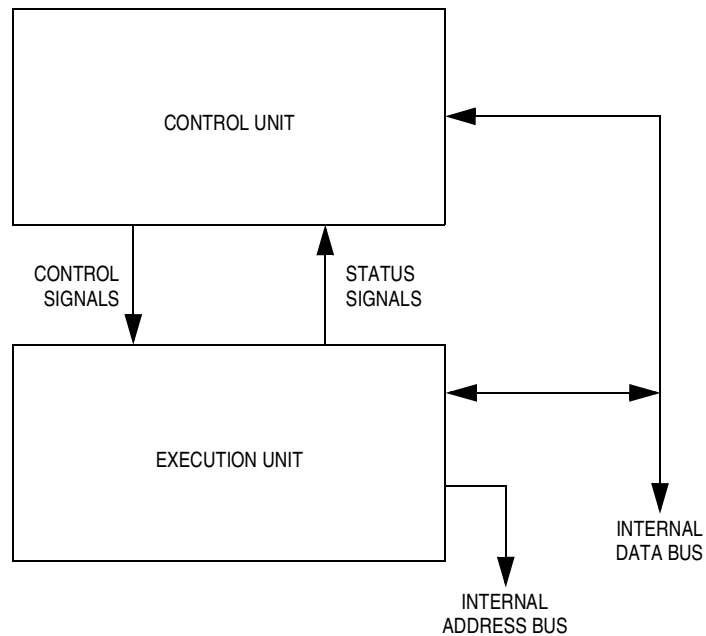
## 2.4 CPU08 Functional Description

This subsection is an overview of the architecture of the M68HC08 CPU with functional descriptions of the major blocks of the CPU.

The CPU08, as shown in [Figure 2-7](#), is divided into two main blocks:

- Control unit
- Execution unit

The control unit contains a finite state machine along with miscellaneous control and timing logic. The outputs of this block drive the execution unit, which contains the arithmetic logic unit (ALU), registers, and bus interface.

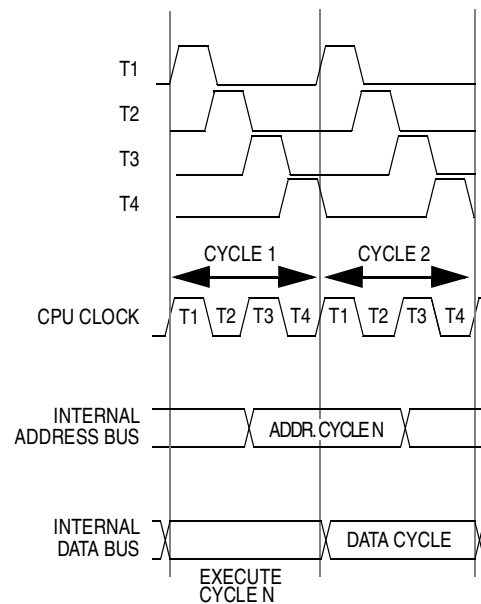


**Figure 2-7. CPU08 Block Diagram**

## 2.4.1 Internal Timing

The CPU08 derives its timing from a 4-phase clock, each phase identified as either T1, T2, T3, or T4. A CPU bus cycle consists of one clock pulse from each phase, as shown in **Figure 2-8**. To simplify subsequent diagrams, the T clocks have been combined into a single signal called the CPU clock. The start of a CPU cycle is defined as the leading edge of T1, though the address associated with this cycle does not drive the address bus until T3. Note that the new address leads the associated data by one-half of a bus cycle.

For example, the data read associated with a new PC value generated in T1/T2 of cycle 1 in **Figure 2-8** would not be read into the control unit until T2 of the next cycle.



**Figure 2-8. Internal Timing Detail**

## 2.4.2 Control Unit

The control unit consists of:

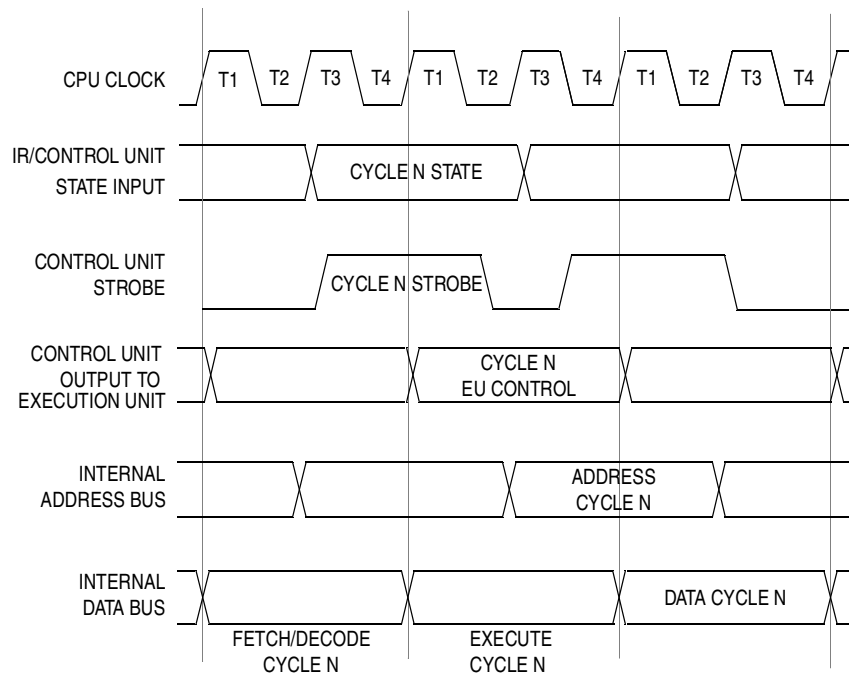
- Sequencer
- Control store
- Random control logic

These blocks make up a finite state machine, which generates all the controls for the execution unit.

The sequencer provides the next state of the machine to the control store based on the contents of the instruction register (IR) and the current state of the machine. The control store is strobed (enabled) when the next state input is stable, producing an output that represents the decoded next state condition for the execution unit (EU). This result, with the help of some random logic, is used to generate the control signals that configure the execution unit. The random logic selects the appropriate signals and adds timing to the outputs of the control store. The control unit fires once per bus cycle but runs almost a full cycle ahead of the execution unit to decode and generate all the controls for the next cycle. The sequential nature of the machine is shown in [Figure 2-9](#).

The sequencer also contains and controls the opcode lookahead register, which is used to prefetch the next sequential instruction. Timing of this operation is discussed in [2.4.4 Instruction Execution](#).





**Figure 2-9. Control Unit Timing**

### 2.4.3 Execution Unit

The execution unit (EU) contains all the registers, the arithmetic logic unit (ALU), and the bus interface. Once per bus cycle a new address is computed by passing selected register values along the internal address buses to the address buffers. Note that the new address leads the associated data by one half of a bus cycle. The execution unit also contains some special function logic for unusual instructions such as DAA, unsigned multiply (MUL), and divide (DIV).

### 2.4.4 Instruction Execution

Each instruction has defined execution boundaries and executes in a finite number of T1-T2-T3-T4 cycles. All instructions are responsible for fetching the next opcode into the opcode lookahead register at some time during execution. The opcode lookahead register is copied into the instruction register during the last cycle of an instruction. This new instruction begins executing during the T1 clock after it has been loaded into the instruction register.

Note that all instructions are also responsible for incrementing the PC after the next instruction prefetch is under way. Therefore, when an instruction finishes (that is, at an instruction boundary), the PC will be pointing to the byte **following** the opcode fetched by the instruction. An example sequence of instructions concerning address and data bus activity with respect to instruction boundaries is shown in [Figure 2-10](#).

A signal from the control unit, OPCODE LOOKAHEAD, indicates the cycle when the next opcode is fetched. Another control signal, LASTBOX, indicates the last cycle of the currently executing instruction. In most cases, OPCODE LOOKAHEAD and LASTBOX are active at the same time. For some instructions, however, the OPCODE LOOKAHEAD signal is asserted earlier in the instruction and the next opcode is prefetched and held in the lookahead register until the end of the currently executing instruction.

In the instruction boundaries example ([Figure 2-10](#)) the OPCODE LOOKAHEAD and LASTBOX are asserted simultaneously during TAX and increment INCX execution, but the load accumulator from memory (LDA) indexed with 8-bit offset instruction prefetches the next opcode before the last cycle. Refer to [Figure 2-11](#). The boldface instructions in [Figure 2-10](#) are illustrated in [Figure 2-11](#).

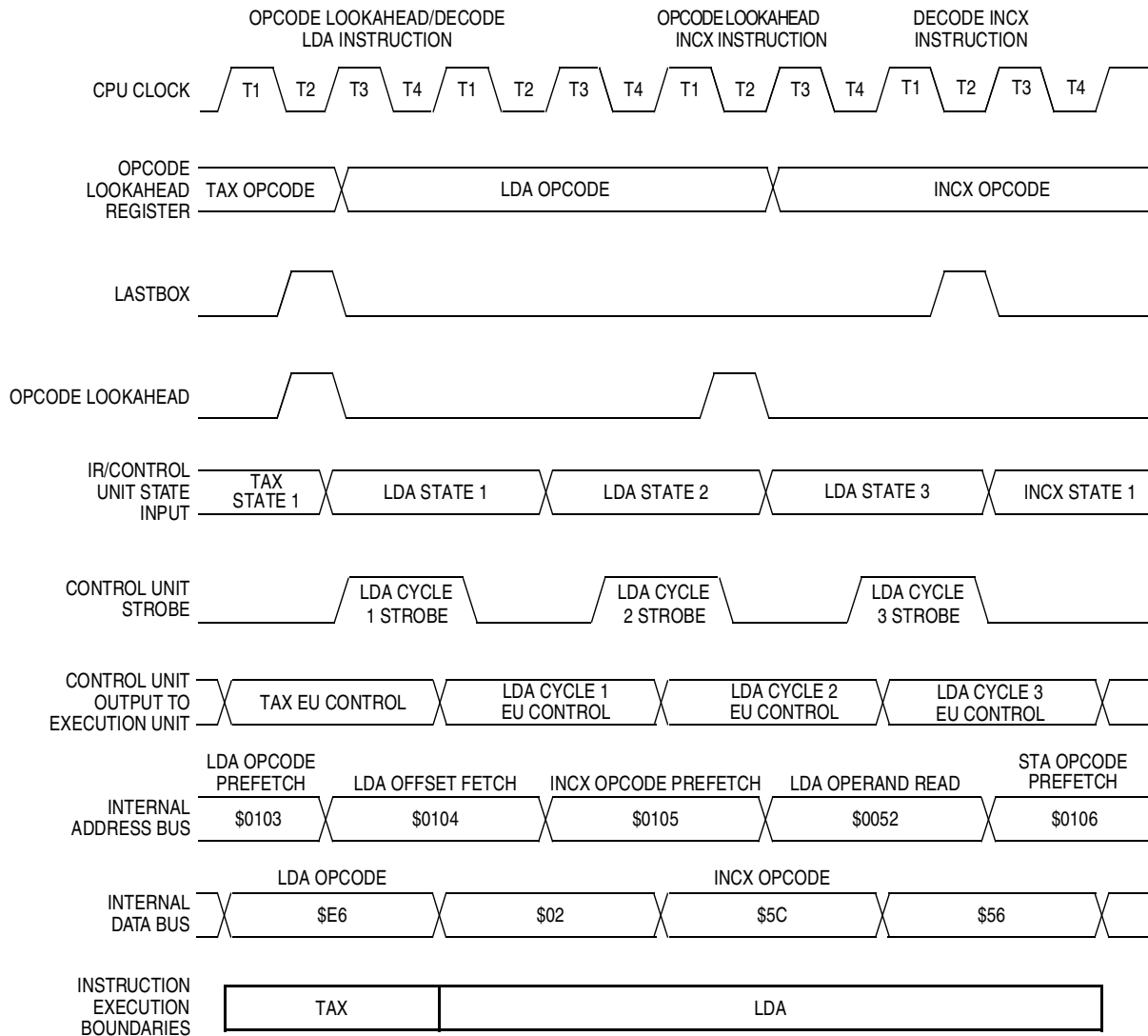
```

                                ORG     $50
                                FCB     $12  $34  $56

                                ORG     $100
0100  A6  50  LDA    # $50      ; A = $50      PC=$0103
0102  97          TAX          ; A -> X        PC=$0104
0103  e6  02  LDA    2,X      ; [X+2] -> A    PC=$0106
0105  5c          INCX        ; X = X+1      PC=$0107
0106  c7  80  00 STA    $8000  ; A -> $8000   PC=$010A

```

**Figure 2-10. Instruction Boundaries**



**Figure 2-11. Instruction Execution Timing Diagram**



## Section 3. Resets and Interrupts

### 3.1 Contents

3.2	Introduction . . . . .	38
3.3	Elements of Reset and Interrupt Processing . . . . .	39
3.3.1	Recognition . . . . .	39
3.3.2	Stacking . . . . .	40
3.3.3	Arbitration . . . . .	41
3.3.4	Masking . . . . .	43
3.3.5	Returning to Calling Program . . . . .	45
3.4	Reset Processing . . . . .	46
3.4.1	Initial Conditions Established . . . . .	47
3.4.2	CPU . . . . .	47
3.4.3	Operating Mode Selection . . . . .	47
3.4.4	Reset Sources . . . . .	48
3.4.5	External Reset . . . . .	49
3.4.6	Active Reset from an Internal Source . . . . .	49
3.5	Interrupt Processing . . . . .	49
3.5.1	Interrupt Sources and Priority . . . . .	51
3.5.2	Interrupts in Stop and Wait Modes . . . . .	52
3.5.3	Nesting of Multiple Interrupts . . . . .	52
3.5.4	Allocating Scratch Space on the Stack . . . . .	53

## 3.2 Introduction

The CPU08 in a microcontroller executes instructions sequentially. In many applications it is necessary to execute sets of instructions in response to requests from various peripheral devices. These requests are often asynchronous to the execution of the main program. Resets and interrupts are both types of CPU08 exceptions. Entry to the appropriate service routine is called exception processing.

Reset is required to initialize the device into a known state, including loading the program counter (PC) with the address of the first instruction. Reset and interrupt operations share the common concept of vector fetching to force a new starting point for further CPU08 operations.

Interrupts provide a way to suspend normal program execution temporarily so that the CPU08 can be freed to service these requests. The CPU08 can process up to 128 separate interrupt sources including a software interrupt (SWI).

On-chip peripheral systems generate maskable interrupts that are recognized only if the global interrupt mask bit (I bit) in the condition code register is clear (reset is non-maskable). Maskable interrupts are prioritized according to a default arrangement. (See [Table 3-2](#) and [3.5.1 Interrupt Sources and Priority](#).) When interrupt conditions occur in an on-chip peripheral system, an interrupt status flag is set to indicate the condition. When the user's program has properly responded to this interrupt request, the status flag must be cleared.

### 3.3 Elements of Reset and Interrupt Processing

Reset and interrupt processing is handled in discrete, though sometimes concurrent, tasks. It is comprised of interrupt recognition, arbitration (evaluating interrupt priority), stacking of the machine state, and fetching of the appropriate vector. Interrupt processing for a reset is comprised of recognition and a fetch of the reset vector only. These tasks, together with interrupt masking and returning from a service routine, are discussed in this subsection.

#### 3.3.1 Recognition

Reset recognition is asynchronous and is recognized when asserted. Internal resets are asynchronous with instruction execution except for illegal opcode and illegal address, which are inherently instruction-synchronized. Exiting the reset state is always synchronous.

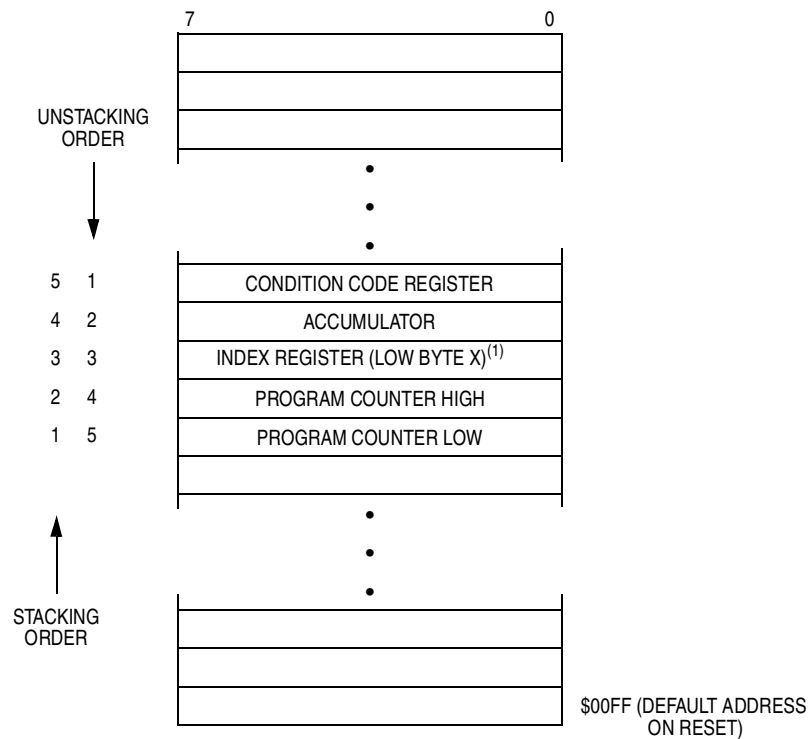
All pending interrupts are recognized by the CPU08 during the last cycle of each instruction. Interrupts that occur during the last cycle will not be recognized by the CPU08 until the last cycle of the following instruction. Instruction execution cannot be suspended to service an interrupt, and so interrupt latency calculations must include the execution time of the longest instruction that could be encountered.

When an interrupt is recognized, an SWI opcode is forced into the instruction register in place of what would have been the next instruction. (When using the CPU08 with the direct memory access (DMA) module, the DMA can suspend instruction operation to service the peripheral.)

Because of the opcode “lookahead” prefetch mechanism, at instruction boundaries the program counter (PC) always points to the address of the next instruction to be executed plus one. The presence of an interrupt is used to modify the SWI flow such that instead of stacking this PC value, the PC is decremented before being stacked. After interrupt servicing is complete, the return-from-interrupt (RTI) instruction will unstack the adjusted PC and use it to prefetch the next instruction again. After SWI interrupt servicing is complete, the RTI instruction then fetches the instruction following the SWI.

## 3.3.2 Stacking

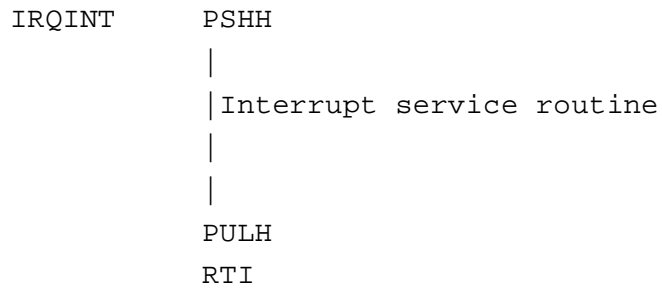
To maintain object code compatibility, the M68HC08 interrupt stack frame is identical to that of the M6805 Family, as shown in **Figure 3-1**. Registers are stacked in the order of PC, X, A, and CCR. They are unstacked in reverse order. Note that the condition code register (CCR) I bit (internal mask) is not set until after the CCR is stacked during cycle 6 of the interrupt stacking procedure. The stack pointer always points to the next available (empty) stack location.



**Figure 3-1. Interrupt Stack Frame**



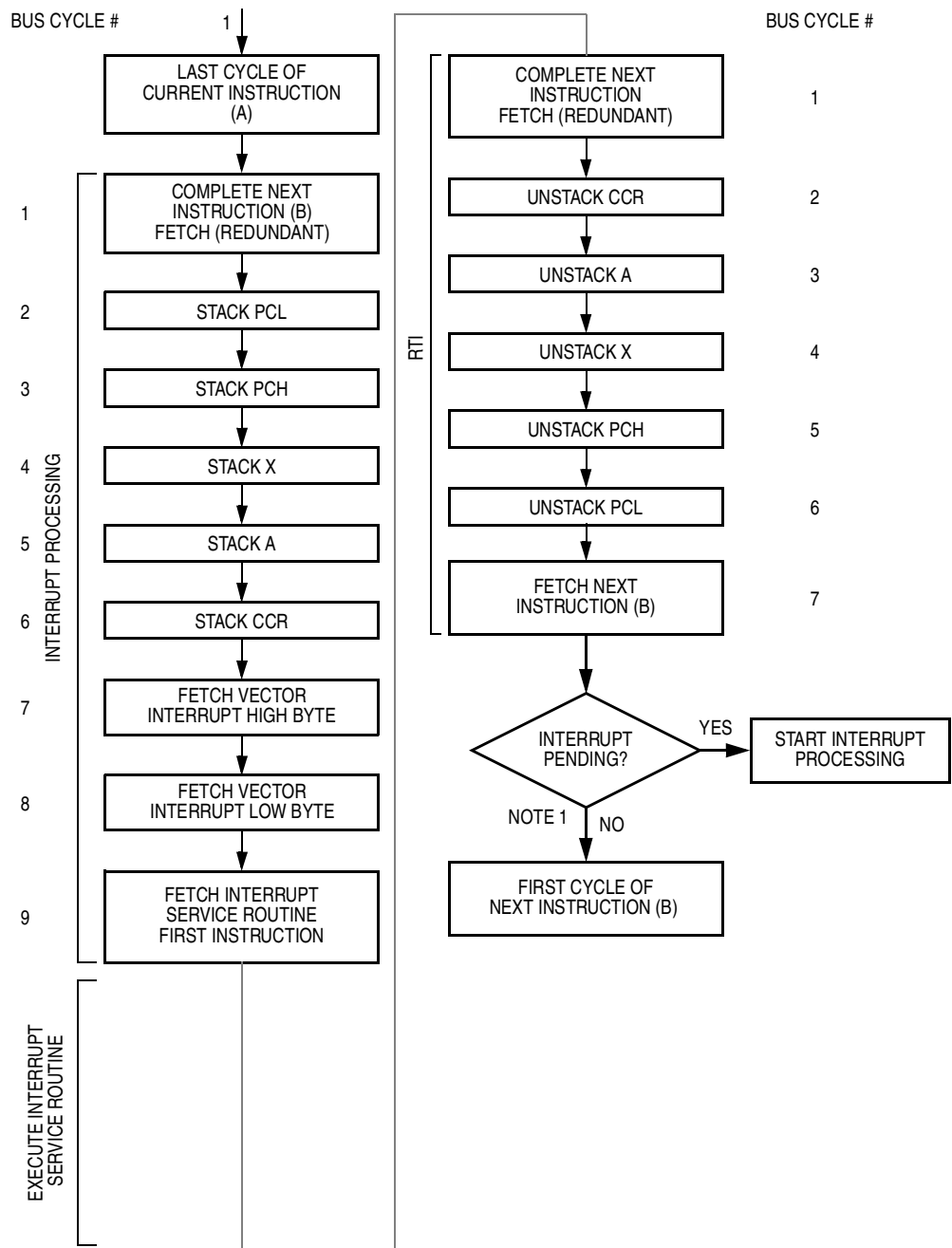
**NOTE:** *To maintain compatibility with the M6805 Family, H (the high byte of the index register) is not stacked during interrupt processing. If the interrupt service routine modifies H or uses the indexed addressing mode, it is the user's responsibility to save and restore it prior to returning. See [Figure 3-2](#).*



**Figure 3-2. H Register Storage**

### 3.3.3 Arbitration

All reset sources always have equal and highest priority and cannot be arbitrated. Interrupts are latched, and arbitration is performed in the system integration module (SIM) at the start of interrupt processing. The arbitration result is a constant that the CPU08 uses to determine which vector to fetch. Once an interrupt is latched by the SIM, no other interrupt may take precedence, regardless of priority, until the latched interrupt is serviced (or the I bit is cleared). See [Figure 3-3](#).

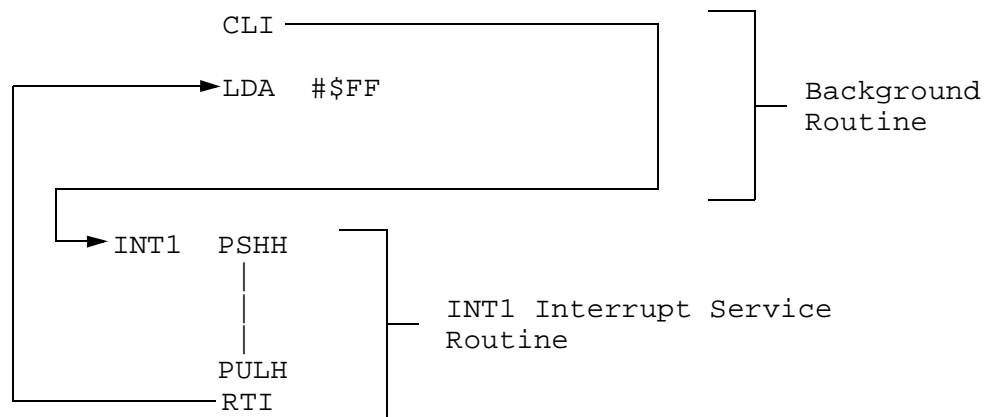


**Figure 3-3. Interrupt Processing Flow and Timing**

### 3.3.4 Masking

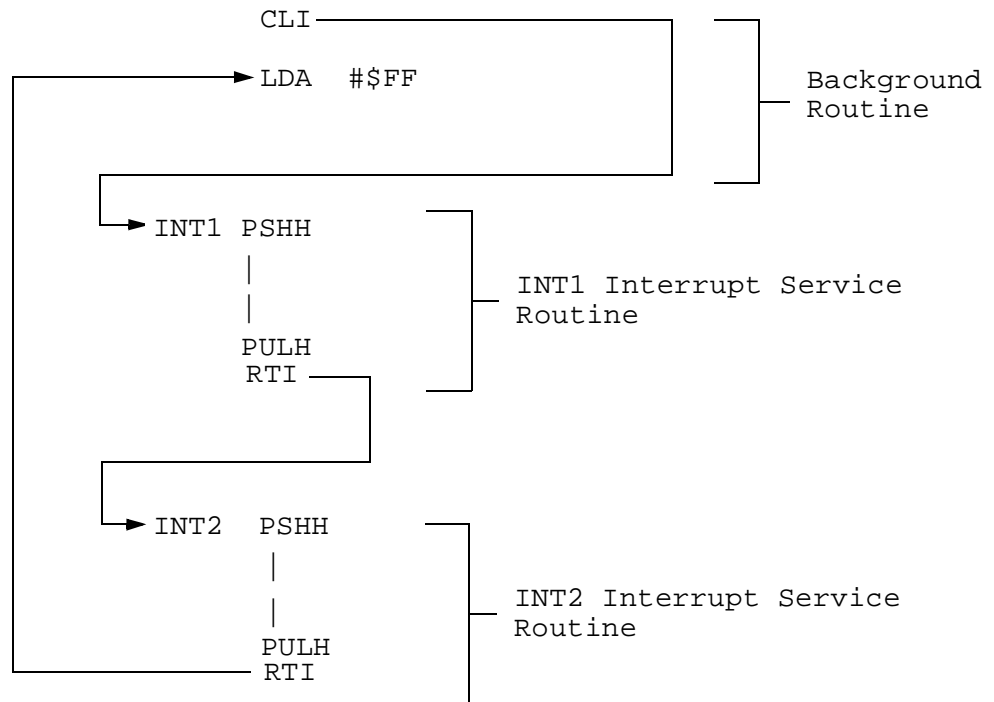
Reset is non-maskable. All other interrupts can be enabled or disabled by the I mask bit in the CCR or by local mask bits in the peripheral control registers. The I bit may also be modified by execution of the set interrupt mask bit (SEI), clear interrupt mask bit (CLI), or transfer accumulator to condition code register (TAP) instructions. The I bit is modified in the first cycle of each instruction (these are all 2-cycle instructions). The I bit is also set during interrupt processing (see [3.3.1 Recognition](#)) and is cleared during the second cycle of the RTI instruction when the CCR is unstacked, provided that the stacked CCR I bit is not modified at the interrupt service routine. (See [3.3.5 Returning to Calling Program](#).)

In all cases where the I bit can be modified, it is modified at least one cycle prior to the last cycle of the instruction or operation, which guarantees that the new I-bit state will be effective prior to execution of the next instruction. For example, if an interrupt is recognized during the CLI instruction, the load accumulator from memory (LDA) instruction will not be executed before the interrupt is serviced. See [Figure 3-4](#).



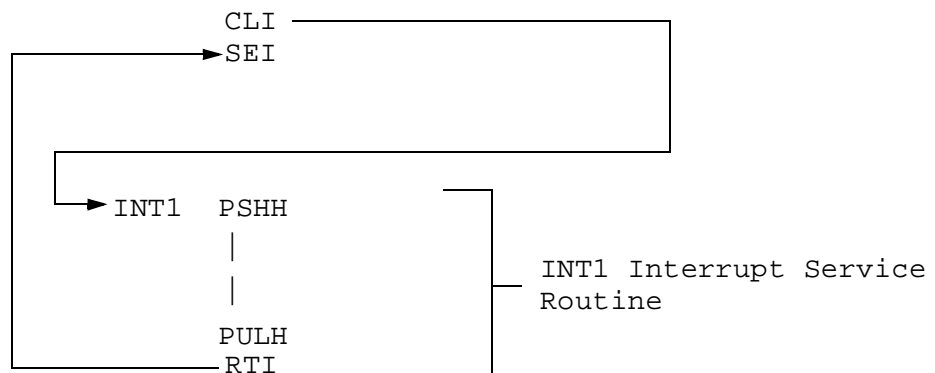
**Figure 3-4. Interrupt Recognition Example 1**

If an interrupt is pending upon exit from the original interrupt service routine, it will also be serviced before the LDA instruction is executed. Note that the LDA opcode is prefetched by both the INT1 and INT2 RTI instructions. However, in the case of the INT1 RTI prefetch, this is a redundant operation. See [Figure 3-5](#).



**Figure 3-5. Interrupt Recognition Example 2**

Similarly, in [Figure 3-6](#), if an interrupt is recognized during the CLI instruction, it will be serviced before the SEI instruction sets the I bit in the CCR.



**Figure 3-6. Interrupt Recognition Example 3**

### 3.3.5 Returning to Calling Program

When an interrupt has been serviced, the RTI instruction terminates interrupt processing and returns to the program that was running at the time of the interrupt. In servicing the interrupt, some or all of the CPU08 registers will have changed. To continue the former program as though uninterrupted, the registers must be restored to the values present at the time the former program was interrupted. The RTI instruction takes care of this by pulling (loading) the saved register values from the stack memory. The last value to be pulled from the stack is the program counter, which causes processing to resume at the point where it was interrupted.

Unstacking the CCR generally clears the I bit, which is cleared during the second cycle of the RTI instruction.

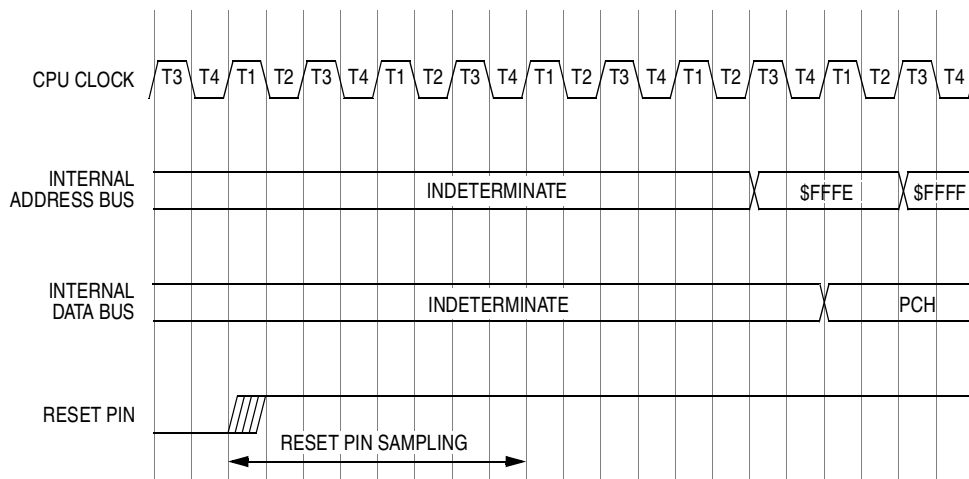
**NOTE:** *Since the return I bit state comes from the stacked CCR, the user, by setting the I bit in the stacked CCR, can block all subsequent interrupts pending or otherwise, regardless of priority, from within an interrupt service routine.*

```
LDA      #$08
ORA      1, SP
STA      1, SP
RTI
```

This capability can be useful in handling a transient situation where the interrupt handler detects that the background program is temporarily unable to cope with the interrupt load and needs some time to recover. At an appropriate juncture, the background program would reinstate interrupts after it has recovered.

## 3.4 Reset Processing

Reset forces the microcontroller unit (MCU) to assume a set of initial conditions and to begin executing instructions from a predetermined starting address. For the M68HC08 Family, reset assertion is asynchronous with instruction execution, and so the initial conditions can be assumed to take effect almost immediately after applying an active low level to the reset pin, regardless of whether the clock has started. Internally, reset is a clocked process, and so reset negation is synchronous with an internal clock, as shown in [Figure 3-7](#), which shows the internal timing for exiting a pin reset.



**Figure 3-7. Exiting Reset**

The reset system is able to actively pull down the reset output if reset-causing conditions are detected by internal systems. This feature can be used to reset external peripherals or other slave MCU devices.

### 3.4.1 Initial Conditions Established

Once the reset condition is recognized, internal registers and control bits are forced to an initial state. These initial states are described throughout this manual. These initial states in turn control on-chip peripheral systems to force them to known startup states. Most of the initial conditions are independent of the operating mode. This subsection summarizes the initial conditions of the CPU08 and input/output (I/O) as they leave reset.

### 3.4.2 CPU

After reset the CPU08 fetches the reset vector from locations \$FFFE and \$FFFF (when in monitor mode, the reset vector is fetched from \$FEFE and \$FEFF), loads the vector into the PC, and begins executing instructions. The stack pointer is loaded with \$00FF. The H register is cleared to provide compatibility for existing M6805 object code. All other CPU08 registers are indeterminate immediately after reset; however, the I interrupt mask bit in the condition code register is set to mask any interrupts, and the STOP and WAIT latches are both cleared.

### 3.4.3 Operating Mode Selection

The CPU08 has two modes of operation useful to the user:

- User mode
- Monitor mode

The monitor mode is the same as user mode except that alternate vectors are used by forcing address bit A8 to 0 instead of 1. The reset vector is therefore fetched from addresses \$FEFE and FEFF instead of FFFE and FFFF. This offset allows the CPU08 to execute code from the internal monitor firmware instead of the user code. (Refer to the appropriate technical data manual for specific information regarding the internal monitor description.)

The mode of operation is latched on the rising edge of the reset pin. The monitor mode is selected by connecting two port lines to  $V_{SS}$  and applying an over-voltage of approximately  $2 \times V_{DD}$  to the  $\overline{IRQ1}$  pin concurrent with the rising edge of reset (see [Table 3-1](#)). Port allocation varies from port to port.

**Table 3-1. Mode Selection**

$\overline{IRQ1}$ Pin	Port x	Port y	Mode
$\leq V_{DD}$	X	X	User
$2 \times V_{DD}$	1	0	Monitor

### 3.4.4 Reset Sources

The system integration module (SIM) has master reset control and may include, depending on device implementation, any of these typical reset sources:

- External reset ( $\overline{RESET}$  pin)
- Power-on reset (POR) circuit
- COP watchdog
- Illegal opcode reset
- Illegal address reset
- Low voltage inhibit (LVI) reset

A reset immediately stops execution of the current instruction. All resets produce the vector  $\$FFFE/\$FFFF$  and assert the internal reset signal. The internal reset causes all registers to return to their default values and all modules to return to their reset state.



### 3.4.5 External Reset

A logic 0 applied to the  $\overline{\text{RESET}}$  pin asserts the internal reset signal, which halts all processing on the chip. The CPU08 and peripherals are reset.

### 3.4.6 Active Reset from an Internal Source

All internal reset sources actively pull down the  $\overline{\text{RESET}}$  pin to allow the resetting of external peripherals. The  $\overline{\text{RESET}}$  pin will be pulled down for 16 bus clock cycles; the internal reset signal will continue to be asserted for an additional 16 cycles after that. If the  $\overline{\text{RESET}}$  pin is still low at the end of the second 16 cycles, then an external reset has occurred. If the pin is high, the appropriate bit will be set to indicate the source of the reset.

The active reset feature allows the part to issue a reset to peripherals and other chips within a system built around an M68HC08 MCU.

## 3.5 Interrupt Processing

The group of instructions executed in response to an interrupt is called an interrupt service routine. These routines are much like subroutines except that they are called through the automatic hardware interrupt mechanism rather than by a subroutine call instruction, and all CPU08 registers, except the H register, are saved on the stack. Refer to the description of the interrupt mask (I) found in [2.3.5 Condition Code Register](#).

An interrupt (provided it is enabled) causes normal program flow to be suspended as soon as the currently executing instruction finishes. The interrupt logic then pushes the contents of all CPU08 registers onto the stack, except the H register, so that the CPU08 contents can be restored after the interrupt is finished. After stacking the CPU08 registers, the vector for the highest priority pending interrupt source is loaded into the program counter and execution continues with the first instruction of the interrupt service routine.

An interrupt is concluded with a return-from-interrupt (RTI) instruction, which causes all CPU08 registers and the return address to be recovered from the stack, so that the interrupted program can resume as if there had been no interruption.

Interrupts can be enabled or disabled by the mask bit (I bit) in the condition code register and by local enable mask bits in the on-chip peripheral control registers. The interrupt mask bits in the CCR provide a means of controlling the nesting of interrupts.

In rare cases it may be useful to allow an interrupt routine to be interrupted (see [3.5.3 Nesting of Multiple Interrupts](#)). However, nesting is discouraged because it greatly complicates a system and rarely improves system performance.

By default, the interrupt structure inhibits interrupts during the interrupt entry sequence by setting the interrupt mask bit(s) in the CCR. As the CCR is recovered from the stack during the return from interrupt, the condition code bits return to the enabled state so that additional interrupts can be serviced.

Upon reset, the I bit is set to inhibit all interrupts. After minimum system initialization, software may clear the I bit by a TAP or CLI instruction, thus enabling interrupts.

### 3.5.1 Interrupt Sources and Priority

The CPU08 can have 128 separate vectors including reset and software interrupt (SWI), which leaves 126 inputs for independent interrupt sources. See [Table 3-2](#).

**NOTE:** *Not all CPU08 versions use all available interrupt vectors.*

**Table 3-2. M68HC08 Vectors**

Address	Reset	Priority
FFFE	Reset	1
FFFC	SWI	2
FFFA	IREQ[0]	3
:	:	:
FF02	IREQ[124]	127
FF00	IREQ[125]	128

When the system integration module (SIM) receives an interrupt request, processing begins at the next instruction boundary. The SIM performs the priority decoding necessary if more than one interrupt source is active at the same time. Also, the SIM encodes the highest priority interrupt request into a constant that the CPU08 uses to generate the corresponding interrupt vector.

**NOTE:** *The interrupt source priority for any specific module may not always be the same in different M68HC08 versions. For details about the priority assigned to interrupt sources in a specific M68HC08 device, refer to the SIM section of the technical data manual written for that device.*

As an instruction, SWI has the highest priority other than reset; once the SWI opcode is fetched, no other interrupt can be honored until the SWI vector has been fetched.

### 3.5.2 Interrupts in Stop and Wait Modes

In wait mode the CPU clocks are disabled, but other module clocks remain active. A module that is active during wait mode can wake the CPU08 by an interrupt if the interrupt is enabled. Processing of the interrupt begins immediately.

In stop mode, the system clocks do not run. The system control module inputs are conditioned so that they can be asynchronous. A particular module can wake the part from stop mode with an interrupt provided that the module has been designed to do so.

### 3.5.3 Nesting of Multiple Interrupts

Under normal circumstances, CPU08 interrupt processing arbitrates multiple pending interrupts, selects the highest, and leaves the rest pending. The I bit in the CCR is also set, preventing nesting of interrupts. While an interrupt is being serviced, it effectively becomes the highest priority task for the system. When servicing is complete, the assigned interrupt priority is re-established.

In certain systems where, for example, a low priority interrupt contains a long interrupt service routine, it may not be desirable to lock out all higher priority interrupts while the low priority interrupt executes. Although not generally advisable, controlled nesting of interrupts can be used to solve problems of this nature.

If nesting of interrupts is desired, the interrupt mask bit(s) must be cleared after entering the interrupt service routine. Care must be taken to specifically mask (disable) the present interrupt with a local enable mask bit or clear the interrupt source flag before clearing the mask bit in the CCR. Failure to do so will cause the same source to immediately interrupt, which will rapidly consume all available stack space.

### 3.5.4 Allocating Scratch Space on the Stack

In some systems, it is useful to allocate local variable or scratch space on the stack for use by the interrupt service routine. Temporary storage can also be obtained using the push (PSH) and pull (PUL) instructions; however, the last-in-first-out (LIFO) structure of the stack makes this impractical for more than one or two bytes. The CPU08 features the 16-bit add immediate value (signed) to stack pointer (AIS) instruction to allocate space. The stack pointer indexing instructions can then be used to access this data space, as demonstrated in this example.

```

IRQINT      PSHH                ;Save H register
            AIS      #-16       ;Allocate 16 bytes of local storage
            STA      3,SP       ;Store a value in the second byte
                                ;of local space

```

```

* Note:      The stack pointer must always point to the next
*            empty stack location.  The location addressed
*            by 0,SP should therefore never be used unless the
*            programmer can guarantee no subroutine calls from
*            within the interrupt service routine.

```

```

.
```

```

.
```

```

.
```

```

LDA      3,SP      ;Read the value at a later time

```

```

.
```

```

.
```

```

AIS      #16       ;Clean up stack
PULH                    ;Restore H register
RTI                    ;Return

```

```

* Note:      Subroutine calls alter the offset from the SP to
*            the local variable data space because of the
*            stacked return address.  If the user wishes to
*            access this data space from subroutines called
*            from within the interrupt service routine, then
*            the offsets should be adjusted by +2 bytes for each
*            level of subroutine nesting.

```



---

**Reference Manual — CPU08**

---

**Section 4. Addressing Modes****4.1 Contents**

4.2	Introduction . . . . .	55
4.3	Addressing Modes . . . . .	56
4.3.1	Inherent . . . . .	57
4.3.2	Immediate . . . . .	59
4.3.3	Direct . . . . .	61
4.3.4	Extended . . . . .	63
4.3.5	Indexed, No Offset . . . . .	65
4.3.6	Indexed, 8-Bit Offset . . . . .	65
4.3.7	Indexed, 16-Bit Offset . . . . .	66
4.3.8	Stack Pointer, 8-Bit Offset . . . . .	68
4.3.9	Stack Pointer, 16-Bit Offset . . . . .	68
4.3.10	Relative . . . . .	71
4.3.11	Memory-to-Memory Immediate to Direct . . . . .	73
4.3.12	Memory-to-Memory Direct to Direct . . . . .	73
4.3.13	Memory-to-Memory Indexed to Direct with Post Increment . . . . .	74
4.3.14	Memory-to-Memory Direct to Indexed with Post Increment . . . . .	76
4.3.15	Indexed with Post Increment . . . . .	77
4.3.16	Indexed, 8-Bit Offset with Post Increment . . . . .	78
4.4	Instruction Set Summary . . . . .	79
4.5	Opcode Map . . . . .	87

**4.2 Introduction**

This section describes the addressing modes of the M68HC08 central processor unit (CPU).

### 4.3 Addressing Modes

The CPU08 uses 16 addressing modes for flexibility in accessing data. These addressing modes define how the CPU finds the data required to execute an instruction.

The 16 addressing modes are:

- Inherent
- Immediate
- Direct
- Extended
- Indexed, no offset
- Indexed, 8-bit offset
- Indexed, 16-bit offset
- Stack pointer, 8-bit offset
- Stack pointer, 16-bit offset
- Relative
- Memory-to-memory (four modes):
  - Immediate to direct
  - Direct to direct
  - Indexed to direct with post increment
  - Direct to indexed with post increment
- Indexed with post increment
- Indexed, 8-bit offset with post increment



### 4.3.1 Inherent

Inherent instructions have no operand fetch associated with the instruction, such as decimal adjust accumulator (DAA), clear index high (CLRHI), and divide (DIV). Some of the inherent instructions act on data in the CPU registers, such as clear accumulator (CLRA), and transfer condition code register to the accumulator (TPA). Inherent instructions require no memory address, and most are one byte long. **Table 4-1** lists the instructions that use inherent addressing.

The assembly language statements shown here are examples of the inherent addressing mode. In the code example and throughout this section, **bold** typeface instructions are examples of the specific addressing mode being discussed; a pound sign (#) before a number indicates an immediate operand. The default base is decimal. Hexadecimal numbers are represented by a dollar sign (\$) preceding the number. Some assemblers use hexadecimal as the default numbering system. Refer to the documentation for the particular assembler to determine the proper syntax.

Machine Code	Label	Operation	Operand	Comments
A657	EX_1	LDA	#\$57	;A = \$57
AB45		ADD	#\$45	;A = \$9C
72		DAA		;A = \$02 w/carry ;bit set = \$102
A614	EX_2	LDA	#20	;LS dividend in A
8C		CLRHI		;Clear MS dividend
AE03		LDX	#3	;Divisor in X
52		DIV		;(H:A)/X→A=06,H=02
A630	EX_3	LDA	#\$30	;A = \$30
87		PSHA		;Push \$30 on stack and ;decrement stack ;pointer by 1

**Table 4-1. Inherent Addressing Instructions**

Instruction	Mnemonic
Arithmetic Shift Left	ASLA, ASLX
Arithmetic Shift Right	ASRA, ASRX
Clear Carry Bit	CLC
Clear Interrupt Mask	CLI
Clear	CLRA, CLRX
Clear H (Index Register High)	CLRH
Complement	COMA, COMX
Decimal Adjust Accumulator	DAA
Decrement Accumulator, Branch if Not Equal (\$00)	DBNZA
Decrement X (Index Register Low), Branch if Not Equal (\$00)	DBNZX
Decrement	DECA, DECX
Divide (Integer 16-Bit by 8-Bit Divide)	DIV
Increment	INCA, INCX
Logical Shift Left	LSLA, LSLX
Logical Shift Right	LSRA, LSRX
Multiply	MUL
Negate	NEGA, NEGX
Nibble Swap Accumulator	NSA
No Operation	NOP
Push Accumulator onto Stack	PSHA
Push H (Index Register High) onto Stack	PSHH
Push X (Index Register Low) onto Stack	PSHX
Pull Accumulator from Stack	PULA
Pull H (Index Register High) from Stack	PULH
Pull X (Index Register Low) from Stack	PULX
Rotate Left through Carry	ROLA, ROLX
Rotate Right through Carry	RORA, RORX
Reset Stack Pointer to \$00FF	RSP
Return from Interrupt	RTI
Return from Subroutine	RTS
Set Carry Bit	SEC
Set Interrupt Mask	SEI

**Table 4-1. Inherent Addressing Instructions (Continued)**

Instruction	Mnemonic
Enable $\overline{\text{IRQ}}$ and Stop Oscillator	STOP
Software Interrupt	SWI
Transfer Accumulator to Condition Code Register	TAP
Transfer Accumulator to X (Index Register Low)	TAX
Transfer Condition Code Register to Accumulator	TPA
Test for Negative or Zero	TSTA, TSTX
Transfer Stack Pointer to Index Register (H:X)	TSX
Transfer X (Index Register Low) to Accumulator	TXA
Transfer Index Register (H:X) to Stack Pointer	TXS
Enable Interrupts and Halt CPU	WAIT

### 4.3.2 Immediate

The operand in immediate instructions is contained in the bytes immediately following the opcode. The byte or bytes that follow the opcode are the value of the statement rather than the address of the value. In this case, the effective address of the instruction is specified by the # sign and implicitly points to the byte following the opcode. The immediate value is limited to either one or two bytes, depending on the size of the register involved in the instruction. [Table 4-2](#) lists the instructions that use immediate addressing.

Immediate instructions associated with the index register (H:X) are 3-byte instructions: one byte for the opcode, two bytes for the immediate data byte.

The example code shown here contains two immediate instructions: AIX (add immediate to H:X) and CPHX (compare H:X with immediate value). H:X is first cleared and then incremented by one until it contains \$FFFF. Once the condition specified by the CPHX becomes true, the program branches to START, and the process is repeated indefinitely.

Machine Code	Label	Operation	Operand	Comments
5F	START	CLR <sub>X</sub>		;X = 0
8C		CLR <sub>H</sub>		;H = 0
AF01	TAG	AIX	#1	;(H:X) = (H:X) + 1
65FFFF		CPHX	#\$FFFF	;Compare (H:X) to ;\$FFFF
26F9		BNE	TAG	;Loop until equal
20F5		BRA	START	;Start over

**Table 4-2. Immediate Addressing Instructions**

Instruction	Mnemonic
Add with Carry Immediate Value to Accumulator	ADC
Add Immediate Value to Accumulator	ADD
Add Immediate Value (Signed) to Stack Pointer	AIS
Add Immediate Value (Signed) to Index Register (H:X)	AIX
Logical AND Immediate Value with Accumulator	AND
Bit Test Immediate Value with Accumulator	BIT
Compare A with Immediate and Branch if Equal	CBEQA
Compare X (Index Register Low) with Immediate and Branch if Equal	CBEQX
Compare Accumulator with Immediate Value	CMP
Compare Index Register (H:X) with Immediate Value	CPHX
Compare X (Index Register Low) with Immediate Value	CPX
Exclusive OR Immediate Value with Accumulator	EOR
Load Accumulator from Immediate Value	LDA
Load Index Register (H:X) with Immediate Value	LDHX
Load X (Index Register Low) from Immediate Value	LDX
Inclusive OR Immediate Value	ORA
Subtract with Carry Immediate Value	SBC
Subtract Immediate Value	SUB

### 4.3.3 Direct

Most direct instructions can access any of the first 256 memory addresses with only two bytes. The first byte is the opcode, and the second is the low byte of the operand address. The high-order byte of the effective address is assumed to be \$00 and is not included as an instruction byte (saving program memory space and execution time). The use of direct addressing mode is therefore limited to operands in the \$0000–\$00FF area of memory (called the direct page or page 0).

Direct addressing instructions take one less byte of program memory space than the equivalent instructions using extended addressing. By eliminating the additional memory access, the execution time is reduced by one cycle. In the course of a long program, this savings can be substantial. Most microcontroller units place some if not all random-access memory (RAM) in the \$0000–\$00FF area; this allows the designer to assign these locations to frequently referenced data variables, thus saving execution time.

BRSET and BRCLR are 3-byte instructions that use direct addressing to access the operand and relative addressing to specify a branch destination.

CPHX, STHX, and LDHX are 2-byte instructions that fetch a 16-bit operand. The most significant byte comes from the direct address; the least significant byte comes from the direct address + 1.

**Table 4-3** lists the instructions that use direct addressing.

This example code contains two direct addressing mode instructions: STHX (store H:X in memory) and CPHX (compare H:X with memory). The first STHX instruction initializes RAM storage location TEMP to zero, and the second STHX instruction loads TEMP with \$5555. The CPHX instruction compares the value in H:X with the value of RAM:(RAM + 1).

In this example, RAM:(RAM + 1) = TEMP = \$50:\$51 = \$5555.

Machine Code	Label	Operation	Operand	Comments
	RAM	EQU	\$50	;RAM equate
	ROM	EQU	\$6E00	;ROM equate
		ORG	\$RAM	;Beginning of RAM
	TEMP	RMB	2	;Reserve 2 bytes
		ORG	\$ROM	;Beginning of ROM
5F	START	CLR <sub>X</sub>		;X = 0
8C		CLR <sub>H</sub>		;H = 0
3550		ST <sub>HX</sub>	TEMP	;H:X=0 > temp
455555		LD <sub>HX</sub>	#\$5555	;Load H:X with \$5555
3550		ST <sub>HX</sub>	TEMP	;Temp=\$5555
7550	BAD_PART	CP <sub>HX</sub>	RAM	;RAM=temp
26FC		BNE	BAD_PART	;RAM=temp will be ;same unless something ;is very wrong!
20F1		BRA	START	;Do it again

**Table 4-3. Direct Addressing Instructions**

Instruction	Mnemonic
Add Memory and Carry to Accumulator	ADC
Add Memory and Accumulator	ADD
Logical AND of Memory and Accumulator	AND
Arithmetic Shift Left Memory	ASL <sup>(1)</sup>
Arithmetic Shift Right Memory	ASR
Clear Bit in Memory	BCLR
Bit Test Memory with Accumulator	BIT
Branch if Bit n in Memory Clear	BRCLR
Branch if Bit n in Memory Set	BRSET
Set Bit in Memory	BSET
Compare Direct with Accumulator and Branch if Equal	CBEQ
Clear Memory	CLR
Compare Accumulator with Memory	CMP
Complement Memory	COM
Compare Index Register (H:X) with Memory	CPHX

**Table 4-3. Direct Addressing Instructions (Continued)**

Instruction	Mnemonic
Compare X (Index Register Low) with Memory	CPX
Decrement Memory and Branch if Not Equal (\$00)	DBNZ
Decrement Memory	DEC
Exclusive OR Memory with Accumulator	EOR
Increment Memory	INC
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load Index Register (H:X) from Memory	LDHX
Load X (Index Register Low) from Memory	LDX
Logical Shift Left Memory	LSL <sup>(1)</sup>
Logical Shift Right Memory	LSR
Negate Memory	NEG
Inclusive OR Accumulator and Memory	ORA
Rotate Memory Left through Carry	ROL
Rotate Memory Right through Carry	ROR
Subtract Memory and Carry from Accumulator	SBC
Store Accumulator in Memory	STA
Store Index Register (H:X) in Memory	STHX
Store X (Index Register Low) in Memory	STX
Subtract Memory from Accumulator	SUB
Test Memory for Negative or Zero	TST

1. ASL = LSL

#### 4.3.4 Extended

Extended instructions can access any address in a 64-Kbyte memory map. All extended instructions are three bytes long. The first byte is the opcode; the second and third bytes are the most significant and least significant bytes of the operand address. This addressing mode is selected when memory above the direct or zero page (\$0000–\$00FF) is accessed.

When using most assemblers, the programmer does not need to specify whether an instruction is direct or extended. The assembler automatically selects the shortest form of the instruction. **Table 4-4** lists the instructions that use the extended addressing mode. An example of the extended addressing mode is shown here.

Machine Code	Label	Operation	Operand	Comments
		ORG	\$50	;Start at \$50
		FCB	\$FF	;\$50 = \$FF
5F		CLR X		
BE50		LDX	\$0050	;Load X direct
		ORG	\$6E00	;Start at \$6E00
		FCB	\$FF	;\$6E00 = \$FF
5F		CLR X		
CE6E00		LDX	\$6E00	;Load X extended

**Table 4-4. Extended Addressing Instructions**

Instruction	Mnemonic
Add Memory and Carry to Accumulator	ADC
Add Memory and Accumulator	ADD
Logical AND of Memory and Accumulator	AND
Bit Test Memory with Accumulator	BIT
Compare Accumulator with Memory	CMP
Compare X (Index Register Low) with Memory	CPX
Exclusive OR Memory with Accumulator	EOR
Jump	JMP
Jump to Subroutine	JSR
Load Accumulator from Memory	LDA
Load X (Index Register Low) from Memory	LDX
Inclusive OR Accumulator with Memory	ORA
Subtract Memory and Carry from Accumulator	SBC
Store Accumulator in Memory	STA
Store X (Index Register Low) in Memory	STX
Subtract Memory from Accumulator	SUB



### 4.3.5 Indexed, No Offset

Indexed instructions with no offset are 1-byte instructions that access data with variable addresses. X contains the low byte of the conditional address of the operand; H contains the high byte. Due to the addition of the H register, this addressing mode is not limited to the first 256 bytes of memory as in the M68HC05.

If none of the M68HC08 instructions that modify H are used (AIX; CBEQ (ix+); LDHX; MOV (dir/ix+); MOV (ix+/dir); DIV; PULH; TSX), then the H value will be \$00, which ensures complete source code compatibility with M68HC05 Family instructions.

Indexed, no offset instructions can move a pointer through a table or hold the address of a frequently used RAM or input/output (I/O) location.

**Table 4-5** lists instructions that use indexed, no offset addressing.

### 4.3.6 Indexed, 8-Bit Offset

Indexed, 8-bit offset instructions are 2-byte instructions that can access data with variable addresses. The CPU adds the unsigned bytes in H:X to the unsigned byte following the opcode. The sum is the effective address of the operand.

If none of the M68HC08 instructions that modify H are used (AIX; CBEQ (ix+); LDHX; MOV (dir/ix+); MOV (ix+/dir); DIV; PULH; TSX), then the H value will be \$00, which ensures complete source code compatibility with the M68HC05 Family instructions.

Indexed, 8-bit offset instructions are useful in selecting the kth element in an n-element table. The table can begin anywhere and can extend as far as the address map allows. The k value would typically be in H:X, and the address of the beginning of the table would be in the byte following the opcode. Using H:X in this way, this addressing mode is limited to the first 256 addresses in memory. Tables can be located anywhere in the address map when H:X is used as the base address, and the byte following is the offset.

**Table 4-5** lists the instructions that use indexed, 8-bit offset addressing.

## 4.3.7 Indexed, 16-Bit Offset

Indexed, 16-bit offset instructions are 3-byte instructions that can access data with variable addresses at any location in memory. The CPU adds the unsigned contents of H:X to the 16-bit unsigned word formed by the two bytes following the opcode. The sum is the effective address of the operand. The first byte after the opcode is the most significant byte of the 16-bit offset; the second byte is the least significant byte of the offset.

As with direct and extended addressing, most assemblers determine the shortest form of indexed addressing. [Table 4-5](#) lists the instructions that use indexed, 16-bit offset addressing.

Indexed, 16-bit offset instructions are useful in selecting the kth element in an n-element table. The table can begin anywhere and can extend as far as the address map allows. The k value would typically be in H:X, and the address of the beginning of the table would be in the bytes following the opcode.

This example uses the JMP (unconditional jump) instruction to show the three different types of indexed addressing.

Machine Code	Label	Operation	Operand	Comments
FC		JMP	,X	<b>;No offset</b> ;Jump to address ;pointed to by H:X
ECFF		JMP	\$FF,X	<b>;8-bit offset</b> ;Jump to address ;pointed to by H:X + \$FF
DC10FF		JMP	\$10FF,X	<b>;16-bit offset</b> ;Jump to address ;pointed to by H:X + \$10FF

**Table 4-5. Indexed Addressing Instructions**

Instruction	Mnemonic	No Offset	8-Bit Offset	16-Bit Offset
Add Memory and Carry to Accumulator	ADC	✓	✓	✓
Add Memory and Accumulator	ADD	✓	✓	✓
Logical AND of Memory and Accumulator	AND	✓	✓	✓
Arithmetic Shift Left Memory	ASL <sup>(1)</sup>	✓	✓	—
Arithmetic Shift Right Memory	ASR	✓	✓	—
Bit Test Memory with Accumulator	BIT	✓	✓	✓
Clear Memory	CLR	✓	✓	—
Compare Accumulator with Memory	CMP	✓	✓	✓
Complement Memory	COM	✓	✓	—
Compare X (Index Register Low) with Memory	CPX	✓	✓	✓
Decrement Memory and Branch if Not Equal (\$00)	DBNZ	✓	✓	—
Decrement Memory	DEC	✓	✓	—
Exclusive OR Memory with Accumulator	EOR	✓	✓	✓
Increment Memory	INC	✓	✓	—
Jump	JMP	✓	✓	✓
Jump to Subroutine	JSR	✓	✓	✓
Load Accumulator from Memory	LDA	✓	✓	✓
Load X (Index Register Low) from Memory	LDX	✓	✓	✓
Logical Shift Left Memory	LSL <sup>(1)</sup>	✓	✓	—
Logical Shift Right Memory	LSR	✓	✓	—
Negate Memory	NEG	✓	✓	—
Inclusive OR Accumulator and Memory	ORA	✓	✓	✓
Rotate Memory Left through Carry	ROL	✓	✓	—
Rotate Memory Right through Carry	ROR	✓	✓	—
Subtract Memory and Carry from Accumulator	SBC	✓	✓	✓
Store Accumulator in Memory	STA	✓	✓	✓
Store X (Index Register Low) in Memory	STX	✓	✓	✓
Subtract Memory from Accumulator	SUB	✓	✓	✓
Test Memory for Negative or Zero	TST	✓	✓	—

1. ASL = LSL

### 4.3.8 Stack Pointer, 8-Bit Offset

Stack pointer, 8-bit offset instructions are 3-byte instructions that address operands in much the same way as indexed 8-bit offset instructions, only they add the 8-bit offset to the value of the stack pointer instead of the index register.

The stack pointer, 8-bit offset addressing mode permits easy access of data on the stack. The CPU adds the unsigned byte in the 16-bit stack pointer (SP) register to the unsigned byte following the opcode. The sum is the effective address of the operand.

If interrupts are disabled, this addressing mode allows the stack pointer to be used as a second “index” register. [Table 4-6](#) lists the instructions that can be used in the stack pointer, 8-bit offset addressing mode.

Stack pointer relative instructions require a pre-byte for access. Consequently, all SP relative instructions take one cycle longer than their index relative counterparts.

### 4.3.9 Stack Pointer, 16-Bit Offset

Stack pointer, 16-bit offset instructions are 4-byte instructions used to access data relative to the stack pointer with variable addresses at any location in memory. The CPU adds the unsigned contents of the 16-bit stack pointer register to the 16-bit unsigned word formed by the two bytes following the opcode. The sum is the effective address of the operand.

As with direct and extended addressing, most assemblers determine the shortest form of stack pointer addressing. Due to the pre-byte, stack pointer relative instructions take one cycle longer than their index relative counterparts. [Table 4-6](#) lists the instructions that can be used in the stack pointer, 16-bit offset addressing mode.

Examples of the 8-bit and 16-bit offset stack pointer addressing modes are shown here. The first example stores the value of \$20 in location \$10,  $SP = \$10 + \$FF = \$10F$  and then decrements that location until equal to zero. The second example loads the accumulator with the contents of memory location \$250,  $SP = \$250 + \$FF = \$34F$ .

Machine Code	Label	Operation	Operand	Comments
450100		LDHX	#\$0100	
94		TXS		;Reset stack pointer ;to \$00FF
A620		LDA	#\$20	;A = \$20
9EE710		STA	\$10,SP	;Location \$10F = \$20
9E6B10FC	LP	DBNZ	\$10,SP,LP	<b>;8-bit offset</b> ;decrement the ;contents of \$10F ;until equal to zero
450100		LDHX	#\$0100	
94		TXS		;Reset stack pointer ;to \$00FF
9ED60250		LDA	\$0250,SP	<b>;16-bit offset</b> ;Load A with contents ;of \$34F

Stack pointer, 16-bit offset instructions are useful in selecting the kth element in an n-element table. The table can begin anywhere and can extend anywhere in memory. With this 4-byte instruction, the k value would typically be in the stack pointer register, and the address of the beginning of the table is located in the two bytes following the 2-byte opcode.

**Table 4-6. Stack Pointer Addressing Instructions**

Instruction	Mnemonic	8-Bit Offset	16-Bit Offset
Add Memory and Carry to Accumulator	ADC	✓	✓
Add Memory and Accumulator	ADD	✓	✓
Logical AND of Memory and Accumulator	AND	✓	✓
Arithmetic Shift Left Memory	ASL <sup>(1)</sup>	✓	—
Arithmetic Shift Right Memory	ASR	✓	—
Bit Test Memory with Accumulator	BIT	✓	✓
Compare Direct with Accumulator and Branch if Equal	CBEQ	✓	—
Clear Memory	CLR	✓	—
Compare Accumulator with Memory	CMP	✓	✓
Complement Memory	COM	✓	—
Compare X (Index Register Low) with Memory	CPX	✓	✓
Decrement Memory and Branch if Not Equal (\$00)	DBNZ	✓	—
Decrement Memory	DEC	✓	—
Exclusive OR Memory with Accumulator	EOR	✓	✓
Increment Memory	INC	✓	—
Load Accumulator from Memory	LDA	✓	✓
Load X (Index Register Low) from Memory	LDX	✓	✓
Logical Shift Left Memory	LSL <sup>(1)</sup>	✓	—
Logical Shift Right Memory	LSR	✓	—
Negate Memory	NEG	✓	—
Inclusive OR Accumulator and Memory	ORA	✓	✓
Rotate Memory Left through Carry	ROL	✓	—
Rotate Memory Right through Carry	ROR	✓	—
Subtract Memory and Carry from Memory	SBC	✓	✓
Store Accumulator in Memory	STA	✓	✓
Store X (Index Register Low) in Memory	STX	✓	✓
Subtract Memory from Accumulator	SUB	✓	✓
Test Memory for Negative or Zero	TST	✓	—

1. ASL = LSL

### 4.3.10 Relative

All conditional branch instructions use relative addressing to evaluate the resultant effective address (EA). The CPU evaluates the conditional branch destination by adding the signed byte following the opcode to the contents of the program counter. If the branch condition is true, the PC is loaded with the EA. If the branch condition is not true, the CPU goes to the next instruction. The offset is a signed, two's complement byte that gives a branching range of  $-128$  to  $+127$  bytes from the address of the next location after the branch instruction.

Four new branch opcodes test the N, Z, and V (overflow) bits to determine the relative signed values of the operands. These new opcodes are BLT, BGT, BLE, and BGE and are designed to be used with signed arithmetic operations.

When using most assemblers, the programmer does not need to calculate the offset, because the assembler determines the proper offset and verifies that it is within the span of the branch.

**Table 4-7** lists the instructions that use relative addressing.

This example contains two relative addressing mode instructions: BLT (branch if less than, signed operation) and BRA (branch always). In this example, the value in the accumulator is compared to the signed value  $-2$ . Because  $\#1$  is greater than  $-2$ , the branch to TAG will not occur.

Machine Code	Label	Operation	Operand	Comments
A601	TAG	LDA	#1	;A = 1
A1FE		CMP	#-2	;Compare with -2
91FA		BLT	TAG	;Branch if value of A ;is less than -2
20FE	HERE	BRA	HERE	;Branch always

**Table 4-7. Relative Addressing Instructions**

Instruction	Mnemonic
Branch if Carry Clear	BCC
Branch if Carry Set	BCS
Branch if Equal	BEQ
Branch if Greater Than or Equal (Signed)	BGE
Branch if Greater Than (Signed)	BGT
Branch if Half-Carry Clear	BHCC
Branch if Half-Carry Set	BHCS
Branch if Higher	BHI
Branch if Higher or Same	BHS (BCC)
Branch if Interrupt Line High	BIH
Branch if Interrupt Line Low	BIL
Branch if Less Than or Equal (Signed)	BLE
Branch if Lower	BLO (BCS)
Branch if Lower or Same	BLS
Branch if Less Than (Signed)	BLT
Branch if Interrupt Mask Clear	BMC
Branch if Minus	BMI
Branch if Interrupt Mask Set	BMS
Branch if Not Equal	BNE
Branch if Plus	BPL
Branch Always	BRA
Branch if Bit n in Memory Clear	BRCLR
Branch if Bit n in Memory Set	BRSET
Branch Never	BRN
Branch to Subroutine	BSR



### 4.3.11 Memory-to-Memory Immediate to Direct

Move immediate to direct (MOV imm/dir) is a 3-byte, 4-cycle addressing mode generally used to initialize variables and registers in the direct page. The operand in the byte immediately following the opcode is stored in the direct page location addressed by the second byte following the opcode. The MOV instruction associated with this addressing mode does not affect the accumulator value. This example shows that by eliminating the accumulator from the data transfer process, the number of execution cycles decreases from 9 to 4 for a similar immediate to direct operation.

	Machine Code	Label	Operation	Operand	Comments
* Data movement with accumulator					
	B750		PSHA		;Save current A ; value
	A622		LDA	#\$22	;A = \$22
	B7F0		STA	\$F0	;Store \$22 into \$F0
	B650		PULA		;Restore A value
					9 cycles
* Data movement without accumulator					
	6E22F0		MOV	#\$22,\$F0	;Location \$F0 ;= \$22

### 4.3.12 Memory-to-Memory Direct to Direct

Move direct to direct (MOV dir/dir) is a 3-byte, 5-cycle addressing mode generally used in register-to-register movements of data from within the direct page. The operand in the direct page location addressed by the byte immediately following the opcode is stored in the direct page location addressed by the second byte following the opcode. The MOV instruction associated with this addressing mode does not affect the accumulator value. As with the previous addressing mode,

eliminating the accumulator from the data transfer process reduces the number of execution cycles from 10 to 5 for similar direct-to-direct operations (see example). This savings can be substantial for a program containing numerous register-to-register data transfers.

Machine Code		Label	Operation	Operand	Comments
* Data movement with accumulator					
B750	(2 cycles)		PSHA		;Save A value
B6F0	(3 cycles)		LDA	\$F0	;Get contents ;of \$F0
B7F1	(3 cycles)		STA	\$F1	;Location \$F1=\$F0
B650	(2 cycles)		PULA		;Restore A value
	10 cycles				
* Data movement without accumulator					
4EF0F1	(5 cycles)		MOV	\$F0,\$F1	;Move contents of ;\$F0 to \$F1

### 4.3.13 Memory-to-Memory Indexed to Direct with Post Increment

Move indexed to direct, post increment (MOV ix+/dir) is a 2-byte, 4-cycle addressing mode generally used to transfer tables addressed by the index register to a register in the direct page. The tables can be located anywhere in the 64-Kbyte map and can be any size. This instruction does not affect the accumulator value. The operand addressed by H:X is stored in the direct page location addressed by the byte following the opcode. H:X is incremented after the move.

This addressing mode is effective for transferring a buffer stored in RAM to a serial transmit register, as shown in the following example. [Table 4-8](#) lists the memory-to-memory move instructions.

**NOTE:** *Move indexed to direct, post increment instructions will increment H if X is incremented past \$FF.*

This example illustrates an interrupt-driven SCI transmit service routine supporting a circular buffer.

Machine Code	Label	Operation	Operand	Comments
	SIZE	EQU	16	;TX circular ;buffer length
	SCSR1	EQU	\$16	;SCI status ;register 1
	SCDR	EQU	\$18	;SCI transmit ;data register
		ORG	\$50	
	PTR_OUT	RMB	2	;Circular buffer ;data out pointer
	PTR_IN	RMB	2	;Circular buffer ;data in pointer
	TX_B	RMB	SIZE	;Circular buffer
		*		
		*		* SCI transmit data register empty interrupt
		*		* service routine
		*		
		ORG	\$6E00	
55 50	TX_INT	LDHX	PTR_OUT	;Load pointer
B6 16		LDA	SCSR1	;Dummy read of ;SCSR1 as part of ;the TDRE reset
<b>7E 18</b>		<b>MOV</b>	<b>X+, SCDR</b>	;Move new byte to ;SCI data reg. ;Clear TDRE. Post ;increment H:X.
65 00 64		CPHX	#TX_B + SIZE	;Gone past end of ;circular buffer?
23 03		BLS	NOLOOP	;If not, continue
45 00 54		LDHX	#TX_B	;Else reset to ;start of buffer
35 50	NOLOOP	STHX	PTR_OUT	;Save new ;pointer value
80		RTI		;Return

## 4.3.14 Memory-to-Memory Direct to Indexed with Post Increment

Move direct to indexed, post increment (MOV dir/ix+) is a 2-byte, 4-cycle addressing mode generally used to fill tables from registers in the direct page. The tables can be located anywhere in the 64-Kbyte map and can be any size. The instruction associated with this addressing mode does not affect the accumulator value. The operand in the direct page location addressed by the byte immediately following the opcode is stored in the location addressed by H:X. H:X is incremented after the move.

An example of this addressing mode would be in filling a serial receive buffer located in RAM from the receive data register. [Table 4-8](#) lists the memory-to-memory move instructions.

**NOTE:** *Move direct to indexed, post increment instructions will increment H if X is incremented past \$FF.*

This example illustrates an interrupt-driven SCI receive service routine supporting a circular buffer.

Machine Code	Label	Operation	Operand	Comments
	SIZE	EQU	16	;RX circular ;buffer length
	SCSR1	EQU	\$16	;SCI status reg.1
	SCDR	EQU	\$18	;SCI receive ;data reg.
		ORG	\$70	
	PTR_OUT	RMB	2	;Circular buffer ;data out pointer
	PTR_IN	RMB	2	;Circular buffer ;data in pointer
	RX_B	RMB	SIZE	;Circular buffer
	*			
	*			* SCI receive data register full interrupt
	*			* service routine
	*			*

Machine Code	Label	Operation	Operand	Comments
		ORG	\$6E00	
55 72	RX_INT	LDHX	PTR_IN	;Load pointer
B6 16		LDA	SCSR1	;Dummy read of ;SCSR1 as part of ;the RDRF reset
<b>5E 18</b>		<b>MOV</b>	<b>SCDR ,X+</b>	;Move new byte from ;SCI data reg. ;Clear RDRF. Post ;increment H:X.
65 00 64		CPHX	#RX_B + SIZE	;Gone past end of ;circular buffer?
23 03		BLS	NOLOOP	;If not continue
45 00 54		LDHX	#RX_B	;Else reset to ;start of buffer
35 52	NOLOOP	STHX	PTR_IN	;Save new ;pointer value
80		RTI		;Return

**Table 4-8. Memory-to-Memory Move Instructions**

Instruction	Mnemonic
Move Immediate Operand to Direct Memory Location	MOV
Move Direct Memory Operand to Another Direct Memory Location	MOV
Move Indexed Operand to Direct Memory Location	MOV
Move Direct Memory Operand to Indexed Memory Location	MOV

#### 4.3.15 Indexed with Post Increment

Indexed, no offset with post increment instructions are 2-byte instructions that address operands, then increment H:X. X contains the low byte of the conditional address of the operand; H contains the high byte. The sum is the conditional address of the operand. This addressing mode is generally used for table searches. [Table 4-9](#) lists the indexed with post increment instructions.

**NOTE:** *Indexed with post increment instructions will increment H if X is incremented past \$FF.*

## 4.3.16 Indexed, 8-Bit Offset with Post Increment

Indexed, 8-bit offset with post increment instructions are 3-byte instructions that access operands with variable addresses, then increment H:X. X contains the low byte of the conditional address of the operand; H contains the high byte. The sum is the conditional address of the operand. As with indexed, no offset, this addressing mode is generally used for table searches. [Table 4-9](#) lists the indexed with post increment instructions.

**NOTE:** *Indexed, 8-bit offset with post increment instructions will increment H if X is incremented past \$FF.*

This example uses the CBEQ (compare and branch if equal) instruction to show the two different indexed with post increment addressing modes.

Machine Code	Label	Operation	Operand	Comments
A6FF		LDA	#\$FF	;A = \$FF
B710		STA	\$10	;LOC \$10 = \$FF
4E1060		MOV	\$10,\$60	;LOC \$60 = \$FF
5F		CLR X		;Zero X
* Compare contents of A with contents of location pointed to by * H:X and branch to TAG when equal				
<b>7102</b>	<b>LOOP</b>	<b>CBEQ</b>	<b>X+,TAG</b>	<b>;No offset</b>
20FC		BRA	LOOP	;Check next location
5F	TAG	CLR X		;Zero X
* Compare contents of A with contents of location pointed to by * H:X + \$50 and branch to TG1 when equal				
<b>615002</b>	<b>LOOP2</b>	<b>CBEQ</b>	<b>\$50,X+,TG1</b>	<b>;8-bit offset</b>
20FB		BRA	LOOP2	;Check next location
20FE	TG1	BRA	TG1	;Finished

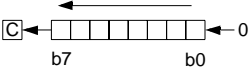
**Table 4-9. Indexed and Indexed, 8-Bit Offset with Post Increment Instructions**

Instruction	Mnemonic
Compare and Branch if Equal, Indexed (H:X)	CBEQ
Compare and Branch if Equal, Indexed (H:X), 8-Bit Offset	CBEQ
Move Indexed Operand to Direct Memory Location	MOV
Move Direct Memory Operand to Indexed Memory Location	MOV

## 4.4 Instruction Set Summary

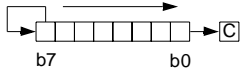
**Table 4-10** provides a summary of the M68HC08 instruction set in all possible addressing modes. The table shows operand construction and the execution time in internal bus clock cycles of each instruction.

**Table 4-10. Instruction Set Summary (Sheet 1 of 9)**

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
ADC #opr8i ADC opr8a ADC opr16a ADC oprx16,X ADC oprx8,X ADC ,X ADC oprx16,SP ADC oprx8,SP	Add with Carry	$A \leftarrow (A) + (M) + (C)$	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A9 B9 C9 D9 E9 F9 9ED9 9EE9	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
ADD #opr8i ADD opr8a ADD opr16a ADD oprx16,X ADD oprx8,X ADD ,X ADD oprx16,SP ADD oprx8,SP	Add without Carry	$A \leftarrow (A) + (M)$	↕	↕	–	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	AB BB CB DB EB FB 9EDB 9EEB	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
AIS #opr8i	Add Immediate Value (Signed) to Stack Pointer	$SP \leftarrow (SP) + (M)$ M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	A7	ii	2
AIX #opr8i	Add Immediate Value (Signed) to Index Register (H:X)	$H:X \leftarrow (H:X) + (M)$ M is sign extended to a 16-bit value	–	–	–	–	–	–	IMM	AF	ii	2
AND #opr8i AND opr8a AND opr16a AND oprx16,X AND oprx8,X AND ,X AND oprx16,SP AND oprx8,SP	Logical AND	$A \leftarrow (A) \& (M)$	0	–	–	↕	↕	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A4 B4 C4 D4 E4 F4 9ED4 9EE4	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
ASL opr8a ASLA ASLX ASL oprx8,X ASL ,X ASL oprx8,SP	Arithmetic Shift Left (Same as LSL)		↕	–	–	↕	↕	↕	DIR INH INH IX1 IX SP1	38 48 58 68 78 9E68	dd ff ff ff ff	4 1 1 4 3 5

# Addressing Modes

Table 4-10. Instruction Set Summary (Sheet 2 of 9)

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
ASR <i>opr8a</i> ASRA ASRX ASR <i>opr8,X</i> ASR <i>,X</i> ASR <i>opr8,SP</i>	Arithmetic Shift Right		↓	-	-	↓	↓	↓	DIR INH IX1 IX SP1	37 47 57 67 77 9E67	dd  ff  ff	4 1 1 4 3 5
BCC <i>rel</i>	Branch if Carry Bit Clear	Branch if (C) = 0	-	-	-	-	-	-	REL	24	rr	3
BCLR <i>n,opr8a</i>	Clear Bit n in Memory	$M_n \leftarrow 0$	-	-	-	-	-	-	DIR (b0) DIR (b1) DIR (b2) DIR (b3) DIR (b4) DIR (b5) DIR (b6) DIR (b7)	11 13 15 17 19 1B 1D 1F	dd dd dd dd dd dd dd dd	4 4 4 4 4 4 4 4
BCS <i>rel</i>	Branch if Carry Bit Set (Same as BLO)	Branch if (C) = 1	-	-	-	-	-	-	REL	25	rr	3
BEQ <i>rel</i>	Branch if Equal	Branch if (Z) = 1	-	-	-	-	-	-	REL	27	rr	3
BGE <i>rel</i>	Branch if Greater Than or Equal To (Signed Operands)	Branch if $(N \oplus V) = 0$	-	-	-	-	-	-	REL	90	rr	3
BGT <i>rel</i>	Branch if Greater Than (Signed Operands)	Branch if $(Z)   (N \oplus V) = 0$	-	-	-	-	-	-	REL	92	rr	3
BHCC <i>rel</i>	Branch if Half Carry Bit Clear	Branch if (H) = 0	-	-	-	-	-	-	REL	28	rr	3
BHCS <i>rel</i>	Branch if Half Carry Bit Set	Branch if (H) = 1	-	-	-	-	-	-	REL	29	rr	3
BHI <i>rel</i>	Branch if Higher	Branch if $(C)   (Z) = 0$	-	-	-	-	-	-	REL	22	rr	3
BHS <i>rel</i>	Branch if Higher or Same (Same as BCC)	Branch if (C) = 0	-	-	-	-	-	-	REL	24	rr	3
BIH <i>rel</i>	Branch if IRQ Pin High	Branch if IRQ pin = 1	-	-	-	-	-	-	REL	2F	rr	3
BIL <i>rel</i>	Branch if IRQ Pin Low	Branch if IRQ pin = 0	-	-	-	-	-	-	REL	2E	rr	3
BIT <i>#opr8i</i> BIT <i>opr8a</i> BIT <i>opr16a</i> BIT <i>opr16,X</i> BIT <i>opr8,X</i> BIT <i>,X</i> BIT <i>opr16,SP</i> BIT <i>opr8,SP</i>	Bit Test	(A) & (M) (CCR Updated but Operands Not Changed)	0	-	-	↓	↓	-	IMM DIR EXT IX2 IX1 IX SP2 SP1	A5 B5 C5 D5 E5 F5 9ED5 9EE5	ii dd hh ll ee ff ff  ee ff ff	2 3 4 4 3 2 5 4
BLE <i>rel</i>	Branch if Less Than or Equal To (Signed Operands)	Branch if $(Z)   (N \oplus V) = 1$	-	-	-	-	-	-	REL	93	rr	3
BLO <i>rel</i>	Branch if Lower (Same as BCS)	Branch if (C) = 1	-	-	-	-	-	-	REL	25	rr	3



**Table 4-10. Instruction Set Summary (Sheet 3 of 9)**

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
BLS <i>rel</i>	Branch if Lower or Same	Branch if (C)   (Z) = 1	-	-	-	-	-	-	REL	23	rr	3
BLT <i>rel</i>	Branch if Less Than (Signed Operands)	Branch if (N ⊕ V) = 1	-	-	-	-	-	-	REL	91	rr	3
BMC <i>rel</i>	Branch if Interrupt Mask Clear	Branch if (I) = 0	-	-	-	-	-	-	REL	2C	rr	3
BMI <i>rel</i>	Branch if Minus	Branch if (N) = 1	-	-	-	-	-	-	REL	2B	rr	3
BMS <i>rel</i>	Branch if Interrupt Mask Set	Branch if (I) = 1	-	-	-	-	-	-	REL	2D	rr	3
BNE <i>rel</i>	Branch if Not Equal	Branch if (Z) = 0	-	-	-	-	-	-	REL	26	rr	3
BPL <i>rel</i>	Branch if Plus	Branch if (N) = 0	-	-	-	-	-	-	REL	2A	rr	3
BRA <i>rel</i>	Branch Always	No Test	-	-	-	-	-	-	REL	20	rr	3
BRCLR <i>n,opr8a,rel</i>	Branch if Bit <i>n</i> in Memory Clear	Branch if (Mn) = 0	-	-	-	-	-	↑	DIR (b0)	01	dd rr	5
									DIR (b1)	03	dd rr	5
									DIR (b2)	05	dd rr	5
									DIR (b3)	07	dd rr	5
									DIR (b4)	09	dd rr	5
									DIR (b5)	0B	dd rr	5
									DIR (b6)	0D	dd rr	5
DIR (b7)	0F	dd rr	5									
BRN <i>rel</i>	Branch Never	Uses 3 Bus Cycles	-	-	-	-	-	-	REL	21	rr	3
BRSET <i>n,opr8a,rel</i>	Branch if Bit <i>n</i> in Memory Set	Branch if (Mn) = 1	-	-	-	-	-	↑	DIR (b0)	00	dd rr	5
									DIR (b1)	02	dd rr	5
									DIR (b2)	04	dd rr	5
									DIR (b3)	06	dd rr	5
									DIR (b4)	08	dd rr	5
									DIR (b5)	0A	dd rr	5
									DIR (b6)	0C	dd rr	5
DIR (b7)	0E	dd rr	5									
BSET <i>n,opr8a</i>	Set Bit <i>n</i> in Memory	Mn ← 1	-	-	-	-	-	-	DIR (b0)	10	dd	4
									DIR (b1)	12	dd	4
									DIR (b2)	14	dd	4
									DIR (b3)	16	dd	4
									DIR (b4)	18	dd	4
									DIR (b5)	1A	dd	4
									DIR (b6)	1C	dd	4
DIR (b7)	1E	dd	4									
BSR <i>rel</i>	Branch to Subroutine	PC ← (PC) + \$0002 push (PCL); SP ← (SP) - \$0001 push (PCH); SP ← (SP) - \$0001 PC ← (PC) + <i>rel</i>	-	-	-	-	-	-	REL	AD	rr	4
CBEQ <i>opr8a,rel</i> CBEQA <i>#opr8i,rel</i> CBEQX <i>#opr8i,rel</i> CBEQ <i>opr8,X+,rel</i> CBEQ <i>,X+,rel</i> CBEQ <i>opr8,SP,rel</i>	Compare and Branch if Equal	Branch if (A) = (M) Branch if (A) = (M) Branch if (X) = (M) Branch if (A) = (M) Branch if (A) = (M) Branch if (A) = (M)	-	-	-	-	-	-	DIR	31	dd rr	5
									IMM	41	ii rr	4
									IMM	51	ii rr	4
									IX1+	61	ff rr	5
									IX+	71	rr	4
									SP1	9E61	ff rr	6

**Table 4-10. Instruction Set Summary (Sheet 4 of 9)**

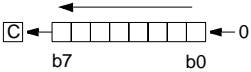
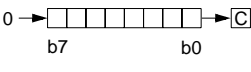
Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
CLC	Clear Carry Bit	$C \leftarrow 0$	-	-	-	-	-	0	INH	98		1
CLI	Clear Interrupt Mask Bit	$I \leftarrow 0$	-	-	0	-	-	-	INH	9A		2
CLR <i>opr8a</i> CLRA CLR X CLR H CLR <i>opr8,X</i> CLR <i>,X</i> CLR <i>opr8,SP</i>	Clear	$M \leftarrow \$00$ $A \leftarrow \$00$ $X \leftarrow \$00$ $H \leftarrow \$00$ $M \leftarrow \$00$ $M \leftarrow \$00$ $M \leftarrow \$00$	0	-	-	0	1	-	DIR INH INH INH IX1 IX SP1	3F 4F 5F 8C 6F 7F 9E6F	dd ff ff	3 1 1 1 3 2 4
CMP <i>#opr8i</i> CMP <i>opr8a</i> CMP <i>opr16a</i> CMP <i>opr16,X</i> CMP <i>opr8,X</i> CMP <i>,X</i> CMP <i>opr16,SP</i> CMP <i>opr8,SP</i>	Compare Accumulator with Memory	$(A) - (M)$ (CCR Updated But Operands Not Changed)	↕	-	-	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A1 B1 C1 D1 E1 F1 9ED1 9EE1	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
COM <i>opr8a</i> COMA COM X COM <i>opr8,X</i> COM <i>,X</i> COM <i>opr8,SP</i>	Complement (One's Complement)	$M \leftarrow (\overline{M}) = \$FF - (M)$ $A \leftarrow (\overline{A}) = \$FF - (A)$ $X \leftarrow (\overline{X}) = \$FF - (X)$ $M \leftarrow (\overline{M}) = \$FF - (M)$ $M \leftarrow (\overline{M}) = \$FF - (M)$ $M \leftarrow (\overline{M}) = \$FF - (M)$	0	-	-	↕	↕	1	DIR INH INH IX1 IX SP1	33 43 53 63 73 9E63	dd ff ff ff	4 1 1 4 3 5
CPHX <i>#opr</i> CPHX <i>opr</i>	Compare Index Register (H:X) with Memory	$(H:X) - (M:M + \$0001)$ (CCR Updated But Operands Not Changed)	↕	-	-	↕	↕	↕	IMM DIR	65 75	jj ii+1 dd	3 4
CPX <i>#opr8i</i> CPX <i>opr8a</i> CPX <i>opr16a</i> CPX <i>opr16,X</i> CPX <i>opr8,X</i> CPX <i>,X</i> CPX <i>opr16,SP</i> CPX <i>opr8,SP</i>	Compare X (Index Register Low) with Memory	$(X) - (M)$ (CCR Updated But Operands Not Changed)	↕	-	-	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A3 B3 C3 D3 E3 F3 9ED3 9EE3	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	$(A)_{10}$	U	-	-	↕	↕	↕	INH	72		2
DBNZ <i>opr8a,rel</i> DBNZ <i>rel</i> DBNZ X <i>rel</i> DBNZ <i>opr8,X,rel</i> DBNZ <i>,X,rel</i> DBNZ <i>opr8,SP,rel</i>	Decrement and Branch if Not Zero	Decrement A, X, or M Branch if (result) $\neq 0$ DBNZX Affects X Not H	-	-	-	-	-	-	DIR INH INH IX1 IX SP1	3B 4B 5B 6B 7B 9E6B	dd rr rr rr ff rr rr ff rr	5 3 3 5 4 6

Table 4-10. Instruction Set Summary (Sheet 5 of 9)

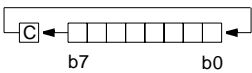
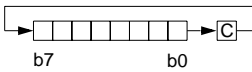
Source Form	Operation	Description	Effect on CCR					Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z				
DEC <i>opr8a</i> DECA DECX DEC <i>opr8,X</i> DEC <i>,X</i> DEC <i>opr8,SP</i>	Decrement	M ← (M) – \$01 A ← (A) – \$01 X ← (X) – \$01 M ← (M) – \$01 M ← (M) – \$01 M ← (M) – \$01	↓	–	–	↓	↓	–	DIR INH INH IX1 IX SP1	3A dd 4A 5A 6A ff 7A 9E6A ff	4 1 1 4 3 5
DIV	Divide	A ← (H:A)÷(X) H ← Remainder	–	–	–	–	↓	↓	INH	52	7
EOR <i>#opr8i</i> EOR <i>opr8a</i> EOR <i>opr16a</i> EOR <i>opr8,X</i> EOR <i>,X</i> EOR <i>opr8,SP</i>	Exclusive OR Memory with Accumulator	A ← (A ⊕ M)	0	–	–	↓	↓	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A8 ii B8 dd C8 hh ll D8 ee ff E8 ff F8 9ED8 ee ff 9EE8 ff	2 3 4 4 3 2 5 4
INC <i>opr8a</i> INCA INCX INC <i>opr8,X</i> INC <i>,X</i> INC <i>opr8,SP</i>	Increment	M ← (M) + \$01 A ← (A) + \$01 X ← (X) + \$01 M ← (M) + \$01 M ← (M) + \$01 M ← (M) + \$01	↓	–	–	↓	↓	–	DIR INH INH IX1 IX SP1	3C dd 4C 5C 6C ff 7C 9E6C ff	4 1 1 4 3 5
JMP <i>opr8a</i> JMP <i>opr16a</i> JMP <i>opr8,X</i> JMP <i>,X</i>	Jump	PC ← Jump Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BC dd CC hh ll DC ee ff EC ff FC	2 3 4 3 3
JSR <i>opr8a</i> JSR <i>opr16a</i> JSR <i>opr8,X</i> JSR <i>,X</i>	Jump to Subroutine	PC ← (PC) + <i>n</i> ( <i>n</i> = 1, 2, or 3) Push (PCL); SP ← (SP) – \$0001 Push (PCH); SP ← (SP) – \$0001 PC ← Unconditional Address	–	–	–	–	–	–	DIR EXT IX2 IX1 IX	BD dd CD hh ll DD ee ff ED ff FD	4 5 6 5 4
LDA <i>#opr8i</i> LDA <i>opr8a</i> LDA <i>opr16a</i> LDA <i>opr8,X</i> LDA <i>,X</i> LDA <i>opr8,SP</i>	Load Accumulator from Memory	A ← (M)	0	–	–	↓	↓	–	IMM DIR EXT IX2 IX1 IX SP2 SP1	A6 ii B6 dd C6 hh ll D6 ee ff E6 ff F6 9ED6 ee ff 9EE6 ff	2 3 4 4 3 2 5 4
LDHX <i>#opr</i> LDHX <i>opr</i>	Load Index Register (H:X) from Memory	H:X ← (M:M + \$0001)	0	–	–	↓	↓	–	IMM DIR	45 ii jj 55 dd	3 4

# Addressing Modes

**Table 4-10. Instruction Set Summary (Sheet 6 of 9)**

Source Form	Operation	Description	Effect on CCR					Address Mode	Opcode	Operand	Cycles	
			V	H	I	N	Z					C
LDX #opr8i LDX opr8a LDX opr16a LDX oprx16,X LDX oprx8,X LDX ,X LDX oprx16,SP LDX oprx8,SP	Load X (Index Register Low) from Memory	$X \leftarrow (M)$	0	-	-	↕	↕	-	IMM DIR EXT IX2 IX1 IX SP2 SP1	AE ii BE dd CE hh ll DE ee ff EE ff FE 9EDE ee ff 9EEE ff	2 3 4 4 3 2 5 4	
LSL opr8a LSLA LSLX LSL oprx8,X LSL ,X LSL oprx8,SP	Logical Shift Left (Same as ASL)		↕	-	-	↕	↕	↕	DIR INH INH IX1 IX SP1	38 dd 48 58 68 ff 78 9E68 ff	4 1 1 4 3 5	
LSR opr8a LSRA LSRX LSR oprx8,X LSR ,X LSR oprx8,SP	Logical Shift Right		↕	-	-	0	↕	↕	DIR INH INH IX1 IX SP1	34 dd 44 54 64 ff 74 9E64 ff	4 1 1 4 3 5	
MOV opr8a,opr8a MOV opr8a,X+ MOV #opr8i,opr8a MOV ,X+,opr8a	Move	$(M)_{\text{destination}} \leftarrow (M)_{\text{source}}$  $H:X \leftarrow (H:X) + \$0001$ in IX+/DIR and DIR/IX+ Modes	0	-	-	↕	↕	-	DIR/DIR DIR/IX+ IMM/DIR IX+/DIR	4E dd dd 5E dd 6E ii dd 7E dd	5 4 4 4	
MUL	Unsigned multiply	$X:A \leftarrow (X) \times (A)$	-	0	-	-	-	0	INH	42		5
NEG opr8a NEGA NEGX NEG oprx8,X NEG ,X NEG oprx8,SP	Negate (Two's Complement)	$M \leftarrow -(M) = \$00 - (M)$ $A \leftarrow -(A) = \$00 - (A)$ $X \leftarrow -(X) = \$00 - (X)$ $M \leftarrow -(M) = \$00 - (M)$ $M \leftarrow -(M) = \$00 - (M)$ $M \leftarrow -(M) = \$00 - (M)$	↕	-	-	↕	↕	↕	DIR INH INH IX1 IX SP1	30 dd 40 50 60 ff 70 9E60 ff	4 1 1 4 3 5	
NOP	No Operation	Uses 1 Bus Cycle	-	-	-	-	-	-	INH	9D		1
NSA	Nibble Swap Accumulator	$A \leftarrow (A[3:0]:A[7:4])$	-	-	-	-	-	-	INH	62		3
ORA #opr8i ORA opr8a ORA opr16a ORA oprx16,X ORA oprx8,X ORA ,X ORA oprx16,SP ORA oprx8,SP	Inclusive OR Accumulator and Memory	$A \leftarrow (A)   (M)$	0	-	-	↕	↕	-	IMM DIR EXT IX2 IX1 IX SP2 SP1	AA ii BA dd CA hh ll DA ee ff EA ff FA 9EDA ee ff 9EEA ff	2 3 4 4 3 2 5 4	
PSHA	Push Accumulator onto Stack	Push (A); $SP \leftarrow (SP) - \$0001$	-	-	-	-	-	-	INH	87		2
PSHH	Push H (Index Register High) onto Stack	Push (H); $SP \leftarrow (SP) - \$0001$	-	-	-	-	-	-	INH	8B		2

**Table 4-10. Instruction Set Summary (Sheet 7 of 9)**

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
PSHX	Push X (Index Register Low) onto Stack	Push (X); $SP \leftarrow (SP) - \$0001$	-	-	-	-	-	-	INH	89		2
PULA	Pull Accumulator from Stack	$SP \leftarrow (SP + \$0001)$ ; Pull (A)	-	-	-	-	-	-	INH	86		2
PULH	Pull H (Index Register High) from Stack	$SP \leftarrow (SP + \$0001)$ ; Pull (H)	-	-	-	-	-	-	INH	8A		2
PULX	Pull X (Index Register Low) from Stack	$SP \leftarrow (SP + \$0001)$ ; Pull (X)	-	-	-	-	-	-	INH	88		2
ROL <i>opr8a</i> ROLA ROLX ROL <i>opr8,X</i> ROL <i>,X</i> ROL <i>opr8,SP</i>	Rotate Left through Carry		↕	-	-	↕	↕	↕	DIR INH INH IX1 IX SP1	39 49 59 69 79 9E69	dd ff ff	4 1 1 4 3 5
ROR <i>opr8a</i> RORA RORX ROR <i>opr8,X</i> ROR <i>,X</i> ROR <i>opr8,SP</i>	Rotate Right through Carry		↕	-	-	↕	↕	↕	DIR INH INH IX1 IX SP1	36 46 56 66 76 9E66	dd ff ff	4 1 1 4 3 5
RSP	Reset Stack Pointer	$SP \leftarrow \$FF$ (High Byte Not Affected)	-	-	-	-	-	-	INH	9C		1
RTI	Return from Interrupt	$SP \leftarrow (SP) + \$0001$ ; Pull (CCR) $SP \leftarrow (SP) + \$0001$ ; Pull (A) $SP \leftarrow (SP) + \$0001$ ; Pull (X) $SP \leftarrow (SP) + \$0001$ ; Pull (PCH) $SP \leftarrow (SP) + \$0001$ ; Pull (PCL)	↕	↕	↕	↕	↕	↕	INH	80		7
RTS	Return from Subroutine	$SP \leftarrow SP + \$0001$ ; Pull (PCH) $SP \leftarrow SP + \$0001$ ; Pull (PCL)	-	-	-	-	-	-	INH	81		4
SBC <i>#opr8i</i> SBC <i>opr8a</i> SBC <i>opr16a</i> SBC <i>opr16,X</i> SBC <i>opr8,X</i> SBC <i>,X</i> SBC <i>opr16,SP</i> SBC <i>opr8,SP</i>	Subtract with Carry	$A \leftarrow (A) - (M) - (C)$	↕	-	-	↕	↕	↕	IMM DIR EXT IX2 IX1 IX SP2 SP1	A2 B2 C2 D2 E2 F2 9ED2 9EE2	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
SEC	Set Carry Bit	$C \leftarrow 1$	-	-	-	-	-	1	INH	99		1
SEI	Set Interrupt Mask Bit	$I \leftarrow 1$	-	-	1	-	-	-	INH	9B		2
STA <i>opr8a</i> STA <i>opr16a</i> STA <i>opr16,X</i> STA <i>opr8,X</i> STA <i>,X</i> STA <i>opr16,SP</i> STA <i>opr8,SP</i>	Store Accumulator in Memory	$M \leftarrow (A)$	0	-	-	↕	↕	-	DIR EXT IX2 IX1 IX SP2 SP1	B7 C7 D7 E7 F7 9ED7 9EE7	dd hh ll ee ff ff ff ee ff ff	3 4 4 3 2 5 4

# Addressing Modes

**Table 4-10. Instruction Set Summary (Sheet 8 of 9)**

Source Form	Operation	Description	Effect on CCR						Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z	C				
STHX <i>opr</i>	Store H:X (Index Reg.)	$(M:M + \$0001) \leftarrow (H:X)$	0	-	-	↑	↑	-	DIR	35	dd	4
STOP	Enable Interrupts: Stop Processing Refer to MCU Documentation	$I \text{ bit} \leftarrow 0$ ; Stop Processing	-	-	0	-	-	-	INH	8E		1
STX <i>opr8a</i> STX <i>opr16a</i> STX <i>opr16,X</i> STX <i>opr8,X</i> STX <i>,X</i> STX <i>opr16,SP</i> STX <i>opr8,SP</i>	Store X (Low 8 Bits of Index Register) in Memory	$M \leftarrow (X)$	0	-	-	↑	↑	-	DIR EXT IX2 IX1 IX SP2 SP1	BF CF DF EF FF 9EDF 9EEF	dd hh ll ee ff ff ff ee ff ff	3 4 4 3 2 5 4
SUB <i>#opr8i</i> SUB <i>opr8a</i> SUB <i>opr16a</i> SUB <i>opr16,X</i> SUB <i>opr8,X</i> SUB <i>,X</i> SUB <i>opr16,SP</i> SUB <i>opr8,SP</i>	Subtract	$A \leftarrow (A) - (M)$	↑	-	-	↑	↑	↑	IMM DIR EXT IX2 IX1 IX SP2 SP1	A0 B0 C0 D0 E0 F0 9ED0 9EE0	ii dd hh ll ee ff ff ff ee ff ff	2 3 4 4 3 2 5 4
SWI	Software Interrupt	$PC \leftarrow (PC) + \$0001$ Push (PCL); $SP \leftarrow (SP) - \$0001$ Push (PCH); $SP \leftarrow (SP) - \$0001$ Push (X); $SP \leftarrow (SP) - \$0001$ Push (A); $SP \leftarrow (SP) - \$0001$ Push (CCR); $SP \leftarrow (SP) - \$0001$ $I \leftarrow 1$ ; PCH $\leftarrow$ Interrupt Vector High Byte PCL $\leftarrow$ Interrupt Vector Low Byte	-	-	1	-	-	-	INH	83		9
TAP	Transfer Accumulator to CCR	$CCR \leftarrow (A)$	↑	↑	↑	↑	↑	↑	INH	84		2
TAX	Transfer Accumulator to X (Index Register Low)	$X \leftarrow (A)$	-	-	-	-	-	-	INH	97		1
TPA	Transfer CCR to Accumulator	$A \leftarrow (CCR)$	-	-	-	-	-	-	INH	85		1
TST <i>opr8a</i> TSTA TSTX TST <i>opr8,X</i> TST <i>,X</i> TST <i>opr8,SP</i>	Test for Negative or Zero	$(M) - \$00$ $(A) - \$00$ $(X) - \$00$ $(M) - \$00$ $(M) - \$00$ $(M) - \$00$	0	-	-	↑	↑	-	DIR INH INH IX1 IX SP1	3D 4D 5D 6D 7D 9E6D	dd ff ff ff ff	3 1 1 3 2 4
TSX	Transfer SP to Index Reg.	$H:X \leftarrow (SP) + \$0001$	-	-	-	-	-	-	INH	95		2
TXA	Transfer X (Index Reg. Low) to Accumulator	$A \leftarrow (X)$	-	-	-	-	-	-	INH	9F		1

**Table 4-10. Instruction Set Summary (Sheet 9 of 9)**

Source Form	Operation	Description	Effect on CCR					Address Mode	Opcode	Operand	Cycles
			V	H	I	N	Z				
TXS	Transfer Index Reg. to SP	SP ← (H:X) – \$0001	–	–	–	–	–	INH	94		2
WAIT	Enable Interrupts; Wait for Interrupt	I bit ← 0; Halt CPU	–	–	0	–	–	INH	8F		1

- |       |   |            |   |
|-------|---|------------|---|
| A     | Accumulator   | <i>n</i>   | Any bit                                     |
| C     | Carry/borrow bit  | <i>opr</i> | Operand (one or two bytes)                  |
| CCR   | Condition code register   | PC         | Program counter                             |
| dd    | Direct address of operand   | PCH        | Program counter high byte                   |
| dd rr | Direct address of operand and relative offset of branch instruction | PCL        | Program counter low byte                    |
| DD    | Direct to direct addressing mode                                    | REL        | Relative addressing mode                    |
| DIR   | Direct addressing mode  | <i>rel</i> | Relative program counter offset byte        |
| DIX+  | Direct to indexed with post increment addressing mode               | rr         | Relative program counter offset byte        |
| ee ff | High and low bytes of offset in indexed, 16-bit offset addressing   | SP1        | Stack pointer, 8-bit offset addressing mode |
| EXT   | Extended addressing mode  | SP2        | Stack pointer 16-bit offset addressing mode |
| ff    | Offset byte in indexed, 8-bit offset addressing                     | SP         | Stack pointer                               |
| H     | Half-carry bit  | U          | Undefined                                   |
| H     | Index register high byte  | V          | Overflow bit                                |
| hh ll | High and low bytes of operand address in extended addressing        | X          | Index register low byte                     |
| I     | Interrupt mask  | Z          | Zero bit                                    |
| ii    | Immediate operand byte  | &          | Logical AND                                 |
| IMD   | Immediate source to direct destination addressing mode              |            | Logical OR                                  |
| IMM   | Immediate addressing mode   | ⊕          | Logical EXCLUSIVE OR                        |
| INH   | Inherent addressing mode  | ( )        | Contents of                                 |
| IX    | Indexed, no offset addressing mode                                  | –( )       | Negation (two's complement)                 |
| IX+   | Indexed, no offset, post increment addressing mode                  | #          | Immediate value                             |
| IX+D  | Indexed with post increment to direct addressing mode               | «          | Sign extend                                 |
| IX1   | Indexed, 8-bit offset addressing mode                               | ←          | Loaded with                                 |
| IX1+  | Indexed, 8-bit offset, post increment addressing mode               | ?          | If  |
| IX2   | Indexed, 16-bit offset addressing mode                              | :          | Concatenated with                           |
| M     | Memory location   | ↓          | Set or cleared                              |
| N     | Negative bit  | —          | Not affected                                |

## 4.5 Opcode Map

The opcode map is provided in [Table 4-11](#).

**Table 4-11. Opcode Map**

HIGH LOW	Bit-Manipulation		Branch	Read-Modify-Write						Control			Register/Memory						
	DIR	DIR	REL	DIR	INH	INH	IX1	SP1	IX	INH	INH	IMM	DIR	EXT	IX2	SP2	IX1	SP1	IX
	0	1	2	3	4	5	6	9E6	7	8	9	A	B	C	D	9ED	E	9EE	F
0	BRSET0 <sup>5</sup> <sub>3</sub> DIR	BSET0 <sup>4</sup> <sub>2</sub> DIR	BRA <sup>3</sup> <sub>2</sub> REL	NEG <sup>4</sup> <sub>2</sub> DIR	NEGA <sup>1</sup> <sub>1</sub> INH	NEGX <sup>1</sup> <sub>1</sub> INH	NEG <sup>4</sup> <sub>2</sub> IX1	NEG <sup>5</sup> <sub>3</sub> SP1	NEG <sup>3</sup> <sub>1</sub> IX	RTI <sup>7</sup> <sub>1</sub> INH	BGE <sup>3</sup> <sub>2</sub> REL	SUB <sup>2</sup> <sub>2</sub> IMM	SUB <sup>3</sup> <sub>2</sub> DIR	SUB <sup>4</sup> <sub>3</sub> EXT	SUB <sup>4</sup> <sub>3</sub> IX2	SUB <sup>5</sup> <sub>4</sub> SP2	SUB <sup>3</sup> <sub>2</sub> IX1	SUB <sup>4</sup> <sub>3</sub> SP1	SUB <sup>2</sup> <sub>1</sub> IX
1	BRCLR0 <sup>5</sup> <sub>3</sub> DIR	BCLR0 <sup>4</sup> <sub>2</sub> DIR	BRN <sup>3</sup> <sub>2</sub> REL	CBEQ <sup>5</sup> <sub>3</sub> DIR	CBEQA <sup>4</sup> <sub>3</sub> IMM	CBEQX <sup>4</sup> <sub>3</sub> IMM	CBEQ <sup>5</sup> <sub>3</sub> IX1+	CBEQ <sup>6</sup> <sub>4</sub> SP1	CBEQ <sup>4</sup> <sub>2</sub> IX+	RTS <sup>4</sup> <sub>1</sub> INH	BLT <sup>3</sup> <sub>2</sub> REL	CMP <sup>2</sup> <sub>2</sub> IMM	CMP <sup>3</sup> <sub>2</sub> DIR	CMP <sup>4</sup> <sub>3</sub> EXT	CMP <sup>4</sup> <sub>3</sub> IX2	CMP <sup>5</sup> <sub>4</sub> SP2	CMP <sup>3</sup> <sub>2</sub> IX1	CMP <sup>4</sup> <sub>3</sub> SP1	CMP <sup>2</sup> <sub>1</sub> IX
2	BRSET1 <sup>5</sup> <sub>3</sub> DIR	BSET1 <sup>4</sup> <sub>2</sub> DIR	BHI <sup>3</sup> <sub>2</sub> REL		MUL <sup>5</sup> <sub>1</sub> INH	DIV <sup>7</sup> <sub>1</sub> INH	NSA <sup>3</sup> <sub>1</sub> INH		DAA <sup>2</sup> <sub>1</sub> INH	BGT <sup>3</sup> <sub>2</sub> REL	SBC <sup>2</sup> <sub>2</sub> IMM	SBC <sup>3</sup> <sub>2</sub> DIR	SBC <sup>4</sup> <sub>3</sub> EXT	SBC <sup>4</sup> <sub>3</sub> IX2	SBC <sup>5</sup> <sub>4</sub> SP2	SBC <sup>3</sup> <sub>2</sub> IX1	SBC <sup>4</sup> <sub>3</sub> SP1	SBC <sup>2</sup> <sub>1</sub> IX	
3	BRCLR1 <sup>5</sup> <sub>3</sub> DIR	BCLR1 <sup>4</sup> <sub>2</sub> DIR	BLS <sup>3</sup> <sub>2</sub> REL	COM <sup>4</sup> <sub>2</sub> DIR	COMA <sup>1</sup> <sub>1</sub> INH	COMX <sup>1</sup> <sub>1</sub> INH	COM <sup>4</sup> <sub>2</sub> IX1	COM <sup>5</sup> <sub>3</sub> SP1	COM <sup>3</sup> <sub>1</sub> IX	SWI <sup>9</sup> <sub>1</sub> INH	BLE <sup>3</sup> <sub>2</sub> REL	CPX <sup>2</sup> <sub>2</sub> IMM	CPX <sup>3</sup> <sub>2</sub> DIR	CPX <sup>4</sup> <sub>3</sub> EXT	CPX <sup>4</sup> <sub>3</sub> IX2	CPX <sup>5</sup> <sub>4</sub> SP2	CPX <sup>3</sup> <sub>2</sub> IX1	CPX <sup>4</sup> <sub>3</sub> SP1	CPX <sup>2</sup> <sub>1</sub> IX
4	BRSET2 <sup>5</sup> <sub>3</sub> DIR	BSET2 <sup>4</sup> <sub>2</sub> DIR	BCC <sup>3</sup> <sub>2</sub> REL	LSR <sup>4</sup> <sub>2</sub> DIR	LSRA <sup>1</sup> <sub>1</sub> INH	LSRX <sup>1</sup> <sub>1</sub> INH	LSR <sup>4</sup> <sub>2</sub> IX1	LSR <sup>5</sup> <sub>3</sub> SP1	LSR <sup>3</sup> <sub>1</sub> IX	TAP <sup>2</sup> <sub>1</sub> INH	TXS <sup>2</sup> <sub>1</sub> INH	AND <sup>2</sup> <sub>2</sub> IMM	AND <sup>3</sup> <sub>2</sub> DIR	AND <sup>4</sup> <sub>3</sub> EXT	AND <sup>4</sup> <sub>3</sub> IX2	AND <sup>5</sup> <sub>4</sub> SP2	AND <sup>3</sup> <sub>2</sub> IX1	AND <sup>4</sup> <sub>3</sub> SP1	AND <sup>2</sup> <sub>1</sub> IX
5	BRCLR2 <sup>5</sup> <sub>3</sub> DIR	BCLR2 <sup>4</sup> <sub>2</sub> DIR	BCS <sup>3</sup> <sub>2</sub> REL	STHX <sup>4</sup> <sub>2</sub> DIR	LDHX <sup>3</sup> <sub>3</sub> IMM	LDHX <sup>3</sup> <sub>3</sub> IMM	CPHX <sup>3</sup> <sub>3</sub> IMM		CPHX <sup>3</sup> <sub>2</sub> DIR	TPA <sup>4</sup> <sub>1</sub> INH	TSX <sup>3</sup> <sub>1</sub> INH	BIT <sup>2</sup> <sub>2</sub> IMM	BIT <sup>3</sup> <sub>2</sub> DIR	BIT <sup>4</sup> <sub>3</sub> EXT	BIT <sup>4</sup> <sub>3</sub> IX2	BIT <sup>5</sup> <sub>4</sub> SP2	BIT <sup>3</sup> <sub>2</sub> IX1	BIT <sup>4</sup> <sub>3</sub> SP1	BIT <sup>2</sup> <sub>1</sub> IX
6	BRSET3 <sup>5</sup> <sub>3</sub> DIR	BSET3 <sup>4</sup> <sub>2</sub> DIR	BNE <sup>3</sup> <sub>2</sub> REL	ROR <sup>4</sup> <sub>2</sub> DIR	RORA <sup>1</sup> <sub>1</sub> INH	RORX <sup>1</sup> <sub>1</sub> INH	ROR <sup>4</sup> <sub>2</sub> IX1	ROR <sup>5</sup> <sub>3</sub> SP1	ROR <sup>3</sup> <sub>1</sub> IX	PULA <sup>2</sup> <sub>1</sub> INH		LDA <sup>2</sup> <sub>2</sub> IMM	LDA <sup>3</sup> <sub>2</sub> DIR	LDA <sup>4</sup> <sub>3</sub> EXT	LDA <sup>4</sup> <sub>3</sub> IX2	LDA <sup>5</sup> <sub>4</sub> SP2	LDA <sup>3</sup> <sub>2</sub> IX1	LDA <sup>4</sup> <sub>3</sub> SP1	LDA <sup>2</sup> <sub>1</sub> IX
7	BRCLR3 <sup>5</sup> <sub>3</sub> DIR	BCLR3 <sup>4</sup> <sub>2</sub> DIR	BEQ <sup>3</sup> <sub>2</sub> REL	ASR <sup>4</sup> <sub>2</sub> DIR	ASRA <sup>1</sup> <sub>1</sub> INH	ASRX <sup>1</sup> <sub>1</sub> INH	ASR <sup>4</sup> <sub>2</sub> IX1	ASR <sup>5</sup> <sub>3</sub> SP1	ASR <sup>3</sup> <sub>1</sub> IX	PSHA <sup>2</sup> <sub>1</sub> INH	TAX <sup>1</sup> <sub>1</sub> INH	AIS <sup>2</sup> <sub>2</sub> IMM	STA <sup>3</sup> <sub>2</sub> DIR	STA <sup>4</sup> <sub>3</sub> EXT	STA <sup>4</sup> <sub>3</sub> IX2	STA <sup>5</sup> <sub>4</sub> SP2	STA <sup>3</sup> <sub>2</sub> IX1	STA <sup>4</sup> <sub>3</sub> SP1	STA <sup>2</sup> <sub>1</sub> IX
8	BRSET4 <sup>5</sup> <sub>3</sub> DIR	BSET4 <sup>4</sup> <sub>2</sub> DIR	BHCC <sup>3</sup> <sub>2</sub> REL	LSL <sup>4</sup> <sub>2</sub> DIR	LSLA <sup>1</sup> <sub>1</sub> INH	LSLX <sup>1</sup> <sub>1</sub> INH	LSL <sup>4</sup> <sub>2</sub> IX1	LSL <sup>5</sup> <sub>3</sub> SP1	LSL <sup>3</sup> <sub>1</sub> IX	PULX <sup>2</sup> <sub>1</sub> INH	CLC <sup>1</sup> <sub>1</sub> INH	EOR <sup>2</sup> <sub>2</sub> IMM	EOR <sup>3</sup> <sub>2</sub> DIR	EOR <sup>4</sup> <sub>3</sub> EXT	EOR <sup>4</sup> <sub>3</sub> IX2	EOR <sup>5</sup> <sub>4</sub> SP2	EOR <sup>3</sup> <sub>2</sub> IX1	EOR <sup>4</sup> <sub>3</sub> SP1	EOR <sup>2</sup> <sub>1</sub> IX
9	BRCLR4 <sup>5</sup> <sub>3</sub> DIR	BCLR4 <sup>4</sup> <sub>2</sub> DIR	BHCS <sup>3</sup> <sub>2</sub> REL	ROL <sup>4</sup> <sub>2</sub> DIR	ROLA <sup>1</sup> <sub>1</sub> INH	ROLX <sup>1</sup> <sub>1</sub> INH	ROL <sup>4</sup> <sub>2</sub> IX1	ROL <sup>5</sup> <sub>3</sub> SP1	ROL <sup>3</sup> <sub>1</sub> IX	PSHX <sup>2</sup> <sub>1</sub> INH	SEC <sup>1</sup> <sub>1</sub> INH	ADC <sup>2</sup> <sub>2</sub> IMM	ADC <sup>3</sup> <sub>2</sub> DIR	ADC <sup>4</sup> <sub>3</sub> EXT	ADC <sup>4</sup> <sub>3</sub> IX2	ADC <sup>5</sup> <sub>4</sub> SP2	ADC <sup>3</sup> <sub>2</sub> IX1	ADC <sup>4</sup> <sub>3</sub> SP1	ADC <sup>2</sup> <sub>1</sub> IX
A	BRSET5 <sup>5</sup> <sub>3</sub> DIR	BSET5 <sup>4</sup> <sub>2</sub> DIR	BPL <sup>3</sup> <sub>2</sub> REL	DEC <sup>4</sup> <sub>2</sub> DIR	DECA <sup>1</sup> <sub>1</sub> INH	DECX <sup>1</sup> <sub>1</sub> INH	DEC <sup>4</sup> <sub>2</sub> IX1	DEC <sup>5</sup> <sub>3</sub> SP1	DEC <sup>3</sup> <sub>1</sub> IX	PULH <sup>2</sup> <sub>1</sub> INH	CLI <sup>1</sup> <sub>1</sub> INH	ORA <sup>2</sup> <sub>2</sub> IMM	ORA <sup>3</sup> <sub>2</sub> DIR	ORA <sup>4</sup> <sub>3</sub> EXT	ORA <sup>4</sup> <sub>3</sub> IX2	ORA <sup>5</sup> <sub>4</sub> SP2	ORA <sup>3</sup> <sub>2</sub> IX1	ORA <sup>4</sup> <sub>3</sub> SP1	ORA <sup>2</sup> <sub>1</sub> IX
B	BRCLR5 <sup>5</sup> <sub>3</sub> DIR	BCLR5 <sup>4</sup> <sub>2</sub> DIR	BMI <sup>3</sup> <sub>2</sub> REL	DBNZ <sup>5</sup> <sub>3</sub> DIR	DBNZA <sup>3</sup> <sub>2</sub> INH	DBNZX <sup>3</sup> <sub>2</sub> INH	DBNZ <sup>5</sup> <sub>3</sub> IX1	DBNZ <sup>6</sup> <sub>4</sub> SP1	DBNZ <sup>4</sup> <sub>2</sub> IX	PSHH <sup>2</sup> <sub>1</sub> INH	SEI <sup>2</sup> <sub>1</sub> INH	ADD <sup>2</sup> <sub>2</sub> IMM	ADD <sup>3</sup> <sub>2</sub> DIR	ADD <sup>4</sup> <sub>3</sub> EXT	ADD <sup>4</sup> <sub>3</sub> IX2	ADD <sup>5</sup> <sub>4</sub> SP2	ADD <sup>3</sup> <sub>2</sub> IX1	ADD <sup>4</sup> <sub>3</sub> SP1	ADD <sup>2</sup> <sub>1</sub> IX
C	BRSET6 <sup>5</sup> <sub>3</sub> DIR	BSET6 <sup>4</sup> <sub>2</sub> DIR	BMC <sup>3</sup> <sub>2</sub> REL	INC <sup>4</sup> <sub>2</sub> DIR	INCA <sup>1</sup> <sub>1</sub> INH	INCX <sup>1</sup> <sub>1</sub> INH	INC <sup>4</sup> <sub>2</sub> IX1	INC <sup>5</sup> <sub>3</sub> SP1	INC <sup>3</sup> <sub>1</sub> IX	CLR <sup>1</sup> <sub>1</sub> INH	RSP <sup>1</sup> <sub>1</sub> INH		JMP <sup>2</sup> <sub>2</sub> DIR	JMP <sup>3</sup> <sub>2</sub> EXT	JMP <sup>4</sup> <sub>3</sub> IX2		JMP <sup>3</sup> <sub>2</sub> IX1		JMP <sup>2</sup> <sub>1</sub> IX
D	BRCLR6 <sup>5</sup> <sub>3</sub> DIR	BCLR6 <sup>4</sup> <sub>2</sub> DIR	BMS <sup>3</sup> <sub>2</sub> REL	TST <sup>3</sup> <sub>2</sub> DIR	TSTA <sup>1</sup> <sub>1</sub> INH	TSTX <sup>1</sup> <sub>1</sub> INH	TST <sup>3</sup> <sub>2</sub> IX1	TST <sup>4</sup> <sub>3</sub> SP1	TST <sup>2</sup> <sub>1</sub> IX		NOP <sup>1</sup> <sub>1</sub> INH	BSR <sup>4</sup> <sub>2</sub> REL	JSR <sup>4</sup> <sub>3</sub> DIR	JSR <sup>5</sup> <sub>3</sub> EXT	JSR <sup>6</sup> <sub>4</sub> IX2		JSR <sup>5</sup> <sub>4</sub> IX1		JSR <sup>4</sup> <sub>3</sub> IX
E	BRSET7 <sup>5</sup> <sub>3</sub> DIR	BSET7 <sup>4</sup> <sub>2</sub> DIR	BIL <sup>3</sup> <sub>2</sub> REL		MOV <sup>5</sup> <sub>3</sub> DD	MOV <sup>4</sup> <sub>3</sub> DIX+	MOV <sup>4</sup> <sub>3</sub> IMD		MOV <sup>4</sup> <sub>2</sub> IX+D	STOP <sup>1</sup> <sub>1</sub> INH	*	LDX <sup>2</sup> <sub>2</sub> IMM	LDX <sup>3</sup> <sub>2</sub> DIR	LDX <sup>4</sup> <sub>3</sub> EXT	LDX <sup>4</sup> <sub>3</sub> IX2	LDX <sup>5</sup> <sub>4</sub> SP2	LDX <sup>3</sup> <sub>2</sub> IX1	LDX <sup>4</sup> <sub>3</sub> SP1	LDX <sup>2</sup> <sub>1</sub> IX
F	BRCLR7 <sup>5</sup> <sub>3</sub> DIR	BCLR7 <sup>4</sup> <sub>2</sub> DIR	BIH <sup>3</sup> <sub>2</sub> REL	CLR <sup>3</sup> <sub>2</sub> DIR	CLRA <sup>1</sup> <sub>1</sub> INH	CLR <sup>1</sup> <sub>1</sub> INH	CLR <sup>3</sup> <sub>2</sub> IX1	CLR <sup>4</sup> <sub>3</sub> SP1	CLR <sup>2</sup> <sub>1</sub> IX	WAIT <sup>1</sup> <sub>1</sub> INH	TXA <sup>1</sup> <sub>1</sub> INH	AIX <sup>2</sup> <sub>2</sub> IMM	STX <sup>3</sup> <sub>2</sub> DIR	STX <sup>4</sup> <sub>3</sub> EXT	STX <sup>4</sup> <sub>3</sub> IX2	STX <sup>5</sup> <sub>4</sub> SP2	STX <sup>3</sup> <sub>2</sub> IX1	STX <sup>4</sup> <sub>3</sub> SP1	STX <sup>2</sup> <sub>1</sub> IX

INH Inherent  
 IMM Immediate  
 DIR Direct  
 EXT Extended  
 DD Direct-Direct  
 IX+D Indexed-Direct

REL Relative  
 IX Indexed, No Offset  
 IX1 Indexed, 8-Bit Offset  
 IX2 Indexed, 16-Bit Offset  
 IMM Immediate-Direct  
 DIX+ Direct-Indexed

SP1 Stack Pointer, 8-Bit Offset  
 SP2 Stack Pointer, 16-Bit Offset  
 IX+ Indexed, No Offset with Post Increment  
 IX1+ Indexed, 1-Byte Offset with Post Increment

\*Pre-byte for stack pointer indexed instructions

High Byte of Opcode in Hexadecimal **F**

Low Byte of Opcode in Hexadecimal **0**

**SUB**<sup>2</sup><sub>1</sub> IX

HC08 Cycles  
 Opcode Mnemonic  
 Number of Bytes / Addressing Mode



## Section 5. Instruction Set

## 5.1 Contents

5.2	Introduction . . . . .	92
5.3	Nomenclature . . . . .	92
5.4	Convention Definitions . . . . .	96
5.5	Instruction Set. . . . .	96
	ADC     Add with Carry . . . . .	97
	ADD     Add without Carry . . . . .	98
	AIS     Add Immediate Value (Signed) to Stack Pointer . . . . .	99
	AIX     Add Immediate Value (Signed) to Index Register . . . . .	100
	AND     Logical AND . . . . .	101
	ASL     Arithmetic Shift Left . . . . .	102
	ASR     Arithmetic Shift Right . . . . .	103
	BCC     Branch if Carry Bit Clear . . . . .	104
	BCLR <i>n</i> Clear Bit <i>n</i> in Memory . . . . .	105
	BCS     Branch if Carry Bit Set . . . . .	106
	BEQ     Branch if Equal . . . . .	107
	BGE     Branch if Greater Than or Equal To . . . . .	108
	BGT     Branch if Greater Than . . . . .	109
	BHCC    Branch if Half Carry Bit Clear . . . . .	110
	BHCS    Branch if Half Carry Bit Set . . . . .	111
	BHI     Branch if Higher. . . . .	112
	BHS     Branch if Higher or Same . . . . .	113
	BIH     Branch if $\overline{\text{IRQ}}$ Pin High . . . . .	114
	BIL     Branch if $\overline{\text{IRQ}}$ Pin Low . . . . .	115
	BIT     Bit Test . . . . .	116
	BLE     Branch if Less Than or Equal To. . . . .	117
	BLO     Branch if Lower . . . . .	118

BLS	Branch if Lower or Same . . . . .	119
BLT	Branch if Less Than . . . . .	120
BMC	Branch if Interrupt Mask Clear. . . . .	121
BMI	Branch if Minus . . . . .	122
BMS	Branch if Interrupt Mask Set . . . . .	123
BNE	Branch if Not Equal . . . . .	124
BPL	Branch if Plus . . . . .	125
BRA	Branch Always. . . . .	126
BRA	Branch Always. . . . .	127
BRCLR <i>n</i>	Branch if Bit <i>n</i> in Memory Clear. . . . .	128
BRN	Branch Never . . . . .	129
BRSET <i>n</i>	Branch if Bit <i>n</i> in Memory Set . . . . .	130
BSET <i>n</i>	Set Bit <i>n</i> in Memory . . . . .	131
BSR	Branch to Subroutine. . . . .	132
CBEQ	Compare and Branch if Equal . . . . .	133
CLC	Clear Carry Bit. . . . .	134
CLI	Clear Interrupt Mask Bit. . . . .	135
CLR	Clear . . . . .	136
CMP	Compare Accumulator with Memory . . . . .	137
COM	Complement (One's Complement) . . . . .	138
CPHX	Compare Index Register with Memory . . . . .	139
CPX	Compare X (Index Register Low) with Memory . . . . .	140
DAA	Decimal Adjust Accumulator . . . . .	141
DAA	Decimal Adjust Accumulator (Continued) . . . . .	142
DBNZ	Decrement and Branch if Not Zero . . . . .	143
DEC	Decrement. . . . .	144
DIV	Divide . . . . .	145
EOR	Exclusive-OR Memory with Accumulator . . . . .	146
INC	Increment . . . . .	147
JMP	Jump . . . . .	148
JSR	Jump to Subroutine . . . . .	149
LDA	Load Accumulator from Memory . . . . .	150
LDHX	Load Index Register from Memory . . . . .	151
LDX	Load X (Index Register Low) from Memory. . . . .	152
LSL	Logical Shift Left . . . . .	153
LSR	Logical Shift Right . . . . .	154
MOV	Move . . . . .	155

MUL	Unsigned Multiply . . . . .	156
NEG	Negate (Two's Complement). . . . .	157
NOP	No Operation . . . . .	158
NSA	Nibble Swap Accumulator . . . . .	159
ORA	Inclusive-OR Accumulator and Memory . . . . .	160
PSHA	Push Accumulator onto Stack . . . . .	161
PSHH	Push H (Index Register High) onto Stack . . . . .	162
PSHX	Push X (Index Register Low) onto Stack . . . . .	163
PULA	Pull Accumulator from Stack . . . . .	164
PULH	Pull H (Index Register High) from Stack . . . . .	165
PULX	Pull X (Index Register Low) from Stack. . . . .	166
ROL	Rotate Left through Carry . . . . .	167
ROR	Rotate Right through Carry . . . . .	168
RSP	Reset Stack Pointer. . . . .	169
RTI	Return from Interrupt. . . . .	170
RTS	Return from Subroutine . . . . .	171
SBC	Subtract with Carry . . . . .	172
SEC	Set Carry Bit . . . . .	173
SEI	Set Interrupt Mask Bit . . . . .	174
STA	Store Accumulator in Memory. . . . .	175
STHX	Store <u>Index</u> Register . . . . .	176
STOP	Enable $\overline{IRQ}$ Pin, Stop Oscillator . . . . .	177
STX	Store X (Index Register Low) in Memory . . . . .	178
SUB	Subtract. . . . .	179
SWI	Software Interrupt . . . . .	180
TAP	Transfer Accumulator to Processor Status Byte . . . . .	181
TAX	Transfer Accumulator to X (Index Register Low) . . . . .	182
TPA	Transfer Processor Status Byte to Accumulator . . . . .	183
TST	Test for Negative or Zero . . . . .	184
TSX	Transfer Stack Pointer to Index Register . . . . .	185
TXA	Transfer X (Index Register Low) to Accumulator . . . . .	186
TXS	Transfer Index Register to Stack Pointer . . . . .	187
WAIT	Enable Interrupts; Stop Processor . . . . .	188

## 5.2 Introduction

This section contains detailed information for all HC08 Family instructions. The instructions are arranged in alphabetical order with the instruction mnemonic set in larger type for easy reference.

## 5.3 Nomenclature

This nomenclature is used in the instruction descriptions throughout this section.

### Operators

( )	=	Contents of register or memory location shown inside parentheses
←	=	Is loaded with (read: “gets”)
&	=	Boolean AND
	=	Boolean OR
⊕	=	Boolean exclusive-OR
×	=	Multiply
÷	=	Divide
:	=	Concatenate
+	=	Add
−	=	Negate (two’s complement)
«	=	Sign extend

### CPU registers

A	=	Accumulator
CCR	=	Condition code register
H	=	Index register, higher order (most significant) eight bits
X	=	Index register, lower order (least significant) eight bits
PC	=	Program counter
PCH	=	Program counter, higher order (most significant) eight bits
PCL	=	Program counter, lower order (least significant) eight bits
SP	=	Stack pointer

### Memory and addressing

- $M$  = A memory location or absolute data, depending on addressing mode
- $M:M + \$0001$  = A 16-bit value in two consecutive memory locations. The higher-order (most significant) eight bits are located at the address of  $M$ , and the lower-order (least significant) eight bits are located at the next higher sequential address.
- rel* = The relative offset, which is the two's complement number stored in the last byte of machine code corresponding to a branch instruction

### Condition code register (CCR) bits

- $V$  = Two's complement overflow indicator, bit 7
- $H$  = Half carry, bit 4
- $I$  = Interrupt mask, bit 3
- $N$  = Negative indicator, bit 2
- $Z$  = Zero indicator, bit 1
- $C$  = Carry/borrow, bit 0 (carry out of bit 7)

### Bit status BEFORE execution of an instruction ( $n = 7, 6, 5, \dots 0$ )

For 2-byte operations such as LDHX, STHX, and CPHX,  $n = 15$  refers to bit 15 of the 2-byte word or bit 7 of the most significant (first) byte.

- $Mn$  = Bit  $n$  of memory location used in operation
- $An$  = Bit  $n$  of accumulator
- $Hn$  = Bit  $n$  of index register H
- $Xn$  = Bit  $n$  of index register X
- $bn$  = Bit  $n$  of the source operand ( $M$ ,  $A$ , or  $X$ )

### Bit status AFTER execution of an instruction

For 2-byte operations such as LDHX, STHX, and CPHX,  $n = 15$  refers to bit 15 of the 2-byte word or bit 7 of the most significant (first) byte.

- $Rn$  = Bit  $n$  of the result of an operation ( $n = 7, 6, 5, \dots 0$ )

## CCR activity figure notation

- = Bit not affected
- 0 = Bit forced to 0
- 1 = Bit forced to 1
- ↕ = Bit set or cleared according to results of operation
- U = Undefined after the operation

## Machine coding notation

- dd = Low-order eight bits of a direct address \$0000–\$00FF (high byte assumed to be \$00)
- ee = Upper eight bits of 16-bit offset
- ff = Lower eight bits of 16-bit offset or 8-bit offset
- ii = One byte of immediate data
- jj = High-order byte of a 16-bit immediate data value
- kk = Low-order byte of a 16-bit immediate data value
- hh = High-order byte of 16-bit extended address
- ll = Low-order byte of 16-bit extended address
- rr = Relative offset

## Source forms

The instruction detail 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.

Typically, assemblers are 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, and plus signs (+) are literal characters.

The definition of a legal label or expression varies from assembler to assembler. Assemblers also vary in the way CPU registers are specified. Refer to assembler documentation for detailed information.

Recommended register designators are a, A, h, H, x, X, sp, and SP.

- n* — Any label or expression that evaluates to a single integer in the range 0–7
- 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 eight bits of an address in the direct page of the 64-Kbyte address space (\$00xx).
- opr16a* — Any label or expression that evaluates to a 16-bit value. The instruction treats this value as an address in the 64-Kbyte address space.
- opr8* — Any label or expression that evaluates to an unsigned 8-bit value; used for indexed addressing
- opr16* — Any label or expression that evaluates to a 16-bit value. Since the MC68HC08S has a 16-bit address bus, this can be either a signed or an unsigned value.
- rel* — 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.

## Address modes

INH	=	Inherent (no operands)
IMM	=	8-bit or 16-bit immediate
DIR	=	8-bit direct
EXT	=	16-bit extended
IX	=	16-bit indexed no offset
IX+	=	16-bit indexed no offset, post increment (CBEQ and MOV only)
IX1	=	16-bit indexed with 8-bit offset from H:X
IX1+	=	16-bit indexed with 8-bit offset, post increment (CBEQ only)
IX2	=	16-bit indexed with 16-bit offset from H:X
REL	=	8-bit relative offset
SP1	=	Stack pointer relative with 8-bit offset
SP2	=	Stack pointer relative with 16-bit offset

## 5.4 Convention Definitions

**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.

**A specific bit** is referred to by mnemonic and bit number. A7 is bit 7 of accumulator A. **A range of bits** is referred to by mnemonic and the bit numbers that define the range. A [7:4] are bits 7 to 4 of the accumulator.

**Parentheses** indicate the contents of a register or memory location, rather than the register or memory location itself. (A) is the contents of the accumulator. In Boolean expressions, parentheses have the traditional mathematical meaning.

## 5.5 Instruction Set

The following pages summarize each instruction, including operation and description, condition codes and Boolean formulae, and a table with source forms, addressing modes, machine code, and cycles.



# ADC

## Add with Carry

# ADC

### Operation

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

### Description

Adds the contents of the C bit to the sum of the contents of A and M and places the result in A. This operation is useful for addition of operands that are larger than eight bits.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↑	1	1	↑	—	↓	↓	↓

$$V: A7 \& M7 \& \overline{R7} \mid \overline{A7} \& \overline{M7} \& R7$$

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

$$H: A3 \& M3 \mid M3 \& \overline{R3} \mid \overline{R3} \& A3$$

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

$$N: R7$$

Set if MSB of result is 1; cleared otherwise

$$Z: \overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$$

Set if result is \$00; cleared otherwise

$$C: A7 \& M7 \mid M7 \& \overline{R7} \mid \overline{R7} \& A7$$

Set if there was a carry from the most significant bit (MSB) of the result; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ADC <i>#opr8i</i>	IMM	A9	ii	2
ADC <i>opr8a</i>	DIR	B9	dd	3
ADC <i>opr16a</i>	EXT	C9	hh ll	4
ADC <i>opr16,X</i>	IX2	D9	ee ff	4
ADC <i>opr8,X</i>	IX1	E9	ff	3
ADC <i>,X</i>	IX	F9		2
ADC <i>opr16,SP</i>	SP2	9ED9	ee ff	5
ADC <i>opr8,SP</i>	SP1	9EE9	ff	4

## ADD

### Add without Carry

## ADD

#### Operation

$A \leftarrow (A) + (M)$

#### Description

Adds the contents of M to the contents of A and places the result in A

#### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↑	1	1	↑	—	↑	↑	↑

V:  $A7 \& M7 \& \overline{R7} \mid \overline{A7} \& \overline{M7} \& R7$

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

H:  $A3 \& M3 \mid M3 \& \overline{R3} \mid \overline{R3} \& A3$

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

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C:  $A7 \& M7 \mid M7 \& \overline{R7} \mid \overline{R7} \& A7$

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

#### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ADD #opr8i	IMM	AB	ii	2
ADD opr8a	DIR	BB	dd	3
ADD opr16a	EXT	CB	hh ll	4
ADD oprx16,X	IX2	DB	ee ff	4
ADD oprx8,X	IX1	EB	ff	3
ADD ,X	IX	FB		2
ADD oprx16,SP	SP2	9EDB	ee ff	5
ADD oprx8,SP	SP1	9EEB	ff	4

# AIS

## Add Immediate Value (Signed) to Stack Pointer

# AIS

### Operation

$SP \leftarrow (SP) + (16 \ll M)$

### Description

Adds the immediate operand to the stack pointer (SP). The immediate value is an 8-bit two's complement signed operand. The 8-bit operand is sign-extended to 16 bits prior to the addition. The AIS instruction can be used to create and remove a stack frame buffer that is used to store temporary variables.

This instruction does not affect any condition code bits so status information can be passed to or from a subroutine or C function and allocation or deallocation of space for local variables will not disturb that status information.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycle, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
AIS #opr8i	IMM	A7	ii	2

# AIX

## Add Immediate Value (Signed) to Index Register

# AIX

**Operation**

$$H:X \leftarrow (H:X) + (16 \ll M)$$

**Description**

Adds an immediate operand to the 16-bit index register, formed by the concatenation of the H and X registers. The immediate operand is an 8-bit two's complement signed offset. The 8-bit operand is sign-extended to 16 bits prior to the addition.

This instruction does not affect any condition code bits so index register pointer calculations do not disturb the surrounding code which may rely on the state of CCR status bits.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
AIX #opr8i	IMM	AF	ii	2

# AND

## Logical AND

# AND

**Operation**  $A \leftarrow (A) \& (M)$

**Description** Performs the logical AND between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical AND of the corresponding bits of M and of A before the operation.

### Condition Codes and Boolean Formulae

V	H	I	N	Z	C
0	1	1	—	—	—

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

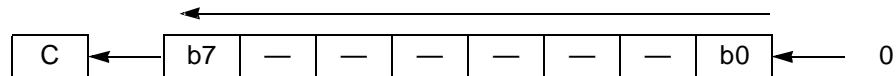
Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
AND #opr8i	IMM	A4	ii	2
AND opr8a	DIR	B4	dd	3
AND opr16a	EXT	C4	hh ll	4
AND oprx16,X	IX2	D4	ee ff	4
AND oprx8,X	IX1	E4	ff	3
AND ,X	IX	F4		2
AND oprx16,SP	SP2	9ED4	ee ff	5
AND oprx8,SP	SP1	9EE4	ff	4

# ASL

## Arithmetic Shift Left (Same as LSL)

# ASL

### Operation



### Description

Shifts all bits of A, X, or M one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A, X, or M. This is mathematically equivalent to multiplication by two. The V bit indicates whether the sign of the result has changed.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V:  $R7 \oplus b7$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: b7

Set if, before the shift, the MSB of A, X, or M was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

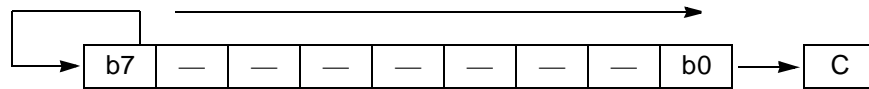
Source Form	Addr Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ASL <i>opr8a</i>	DIR	38	dd	4
ASLA	INH (A)	48		1
ASLX	INH (X)	58		1
ASL <i>opr8,X</i>	IX1	68	ff	4
ASL <i>,X</i>	IX	78		3
ASL <i>opr8,SP</i>	SP1	9E68	ff	5

# ASR

## Arithmetic Shift Right

# ASR

### Operation



### Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is held constant. Bit 0 is loaded into the C bit of the CCR. This operation effectively divides a two's complement value by 2 without changing its sign. The carry bit can be used to round the result.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V:  $R7 \oplus b0$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: b0

Set if, before the shift, the LSB of A, X, or M was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ASR <i>opr8a</i>	DIR	37	dd	4
ASRA	INH (A)	47		1
ASRX	INH (X)	57		1
ASR <i>opr8,X</i>	IX1	67	ff	4
ASR <i>,X</i>	IX	77		3
ASR <i>opr8,SP</i>	SP1	9E67	ff	5

# BCC

## Branch if Carry Bit Clear (Same as BHS)

# BCC

### Operation

If (C) = 0,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch

### Description

Tests state of C bit in CCR and causes a branch if C is clear. BCC can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BCC <i>rel</i>	REL	24	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.



# BCLR *n*

## Clear Bit *n* in Memory

# BCLR *n*

**Operation**  $Mn \leftarrow 0$

**Description** Clear bit *n* ( $n = 7, 6, 5, \dots 0$ ) in location M. All other bits in M are unaffected. In other words, M can be any random-access memory (RAM) or input/output (I/O) register address in the \$0000 to \$00FF area of memory. (Direct addressing mode is used to specify the address of the operand.) This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

**Condition Codes and Boolean Formulae** None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BCLR 0, <i>opr8a</i>	DIR (b0)	11	dd	4
BCLR 1, <i>opr8a</i>	DIR (b1)	13	dd	4
BCLR 2, <i>opr8a</i>	DIR (b2)	15	dd	4
BCLR 3, <i>opr8a</i>	DIR (b3)	17	dd	4
BCLR 4, <i>opr8a</i>	DIR (b4)	19	dd	4
BCLR 5, <i>opr8a</i>	DIR (b5)	1B	dd	4
BCLR 6, <i>opr8a</i>	DIR (b6)	1D	dd	4
BCLR 7, <i>opr8a</i>	DIR (b7)	1F	dd	4

# BCS

## Branch if Carry Bit Set (Same as BLO)

# BCS

### Operation

If (C) = 1,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch

### Description

Tests the state of the C bit in the CCR and causes a branch if C is set. BCS can be used after shift or rotate instructions or to check for overflow after operations on unsigned numbers. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BCS <i>rel</i>	REL	25	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BEQ

## Branch if Equal

# BEQ

### Operation

If  $(Z) = 1$ ,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch; may be used with signed or unsigned operations

### Description

Tests the state of the Z bit in the CCR and causes a branch if Z is set. Compare instructions perform a subtraction with two operands and produce an internal result without changing the original operands. If the two operands were equal, the internal result of the subtraction for the compare will be zero so the Z bit will be equal to one and the BEQ will cause a branch.

This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BEQ <i>rel</i>	REL	27	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BGE

## Branch if Greater Than or Equal To

# BGE

### Operation

If  $(N \oplus V) = 0$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BGE instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch occurs if and only if the two's complement number in the A, X, or H:X register was greater than or equal to the two's complement number in memory.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BGE <i>rel</i>	REL	90	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BGT

## Branch if Greater Than

# BGT

### Operation

If  $(Z) \mid (N \oplus V) = 0$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BGT instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two's complement number in the A, X, or H:X register was greater than the two's complement number in memory.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BGT <i>rel</i>	REL	92	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BHCC

## Branch if Half Carry Bit Clear

# BHCC

### Operation

If (H) = 0,  $PC \leftarrow (PC) + \$0002 + rel$

### Description

Tests the state of the H bit in the CCR and causes a branch if H is clear. This instruction is used in algorithms involving BCD numbers that were originally written for the M68HC05 or M68HC08 devices. The DAA instruction in the HC08 simplifies operations on BCD numbers so BHCC and BHCS should not be needed in new programs. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BHCC <i>rel</i>	REL	28	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BHCS

## Branch if Half Carry Bit Set

# BHCS

### Operation

If (H) = 1,  $PC \leftarrow (PC) + \$0002 + rel$

### Description

Tests the state of the H bit in the CCR and causes a branch if H is set. This instruction is used in algorithms involving BCD numbers that were originally written for the M68HC05 or M68HC08 devices. The DAA instruction in the HC08 simplifies operations on BCD numbers so BHCC and BHCS should not be needed in new programs. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V		H	I	N	Z	C
—	1	1	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BHCS <i>rel</i>	REL	29	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BHI

## Branch if Higher

# BHI

### Operation

If  $(C) \mid (Z) = 0$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

Causes a branch if both C and Z are cleared. If the BHI instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was greater than unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the [BRA](#) instruction for details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BHI <i>rel</i>	REL	22	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.



# BHS

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

# BHS

### Operation

If (C) = 0,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BHS instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was greater than or equal to the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BHS <i>rel</i>	REL	24	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BIH

## Branch if $\overline{\text{IRQ}}$ Pin High

# BIH

**Operation**

If  $\overline{\text{IRQ}}$  pin = 1,  $\text{PC} \leftarrow (\text{PC}) + \$0002 + \text{rel}$

**Description**

Tests the state of the external interrupt pin and causes a branch if the pin is high. See the **BRA** instruction for further details of the execution of the branch.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BIH <i>rel</i>	REL	2F	rr	3

See the **BRA** instruction for a summary of all branches and their complements.

# BIL

## Branch if $\overline{\text{IRQ}}$ Pin Low

# BIL

### Operation

If  $\overline{\text{IRQ}}$  pin = 0,  $\text{PC} \leftarrow (\text{PC}) + \$0002 + \text{rel}$

### Description

Tests the state of the external interrupt pin and causes a branch if the pin is low. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BIL <i>rel</i>	REL	2E	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BIT

## Bit Test

# BIT

**Operation** (A) & (M)

**Description** Performs the logical AND comparison of the contents of A and the contents of M and modifies the condition codes accordingly. Neither the contents of A nor M are altered. (Each bit of the result of the AND would be the logical AND of the corresponding bits of A and M.)

This instruction is typically used to see if a particular bit, or any of several bits, in a byte are 1s. A mask value is prepared with 1s in any bit positions that are to be checked. This mask may be in accumulator A or memory and the unknown value to be checked will be in memory or the accumulator A, respectively. After the BIT instruction, a BNE instruction will branch if any bits in the tested location that correspond to 1s in the mask were 1s.

**Condition Codes and Boolean Formulae**

V		H	I	N	Z	C
0	1	1	—	—	↓	—

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BIT #opr8i	IMM	A5	ii	2
BIT opr8a	DIR	B5	dd	3
BIT opr16a	EXT	C5	hh ll	4
BIT oprx16,X	IX2	D5	ee ff	4
BIT oprx8,X	IX1	E5	ff	3
BIT ,X	IX	F5		2
BIT oprx16,SP	SP2	9ED5	ee ff	5
BIT oprx8,SP	SP1	9EE5	ff	4

# BLE

## Branch if Less Than or Equal To

# BLE

### Operation

If  $(Z) \mid (N \oplus V) = 1$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BLE instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two's complement in the A, X, or H:X register was less than or equal to the two's complement number in memory.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BLE <i>rel</i>	REL	93	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BLO

## Branch if Lower

# BLO

### Operation

If (C) = 1,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BLO instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if the unsigned binary number in the A, X, or H:X register was less than the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BLO <i>rel</i>	REL	25	<i>rr</i>	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BLS

## Branch if Lower or Same

# BLS

### Operation

If  $(C) \mid (Z) = 1$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

Causes a branch if (C is set) or (Z is set). If the BLS instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the unsigned binary number in the A, X, or H:X register was less than or equal to the unsigned binary number in memory. Generally not useful after CLR, COM, DEC, INC, LDA, LDHX, LDX, STA, STHX, STX, or TST because these instructions do not affect the carry bit in the CCR. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycle, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BLS <i>rel</i>	REL	23	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BLT

## Branch if Less Than (Signed Operands)

# BLT

### Operation

If  $(N \oplus V) = 1$ ,  $PC \leftarrow (PC) + \$0002 + rel$

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

### Description

If the BLT instruction is executed immediately after execution of a CMP, CPHX, CPX, SBC, or SUB instruction, the branch will occur if and only if the two's complement number in the A, X, or H:X register was less than the two's complement number in memory. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BLT <i>rel</i>	REL	91	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.



# BMC

## Branch if Interrupt Mask Clear

# BMC

### Operation

If  $(I) = 0$ ,  $PC \leftarrow (PC) + \$0002 + rel$

### Description

Tests the state of the I bit in the CCR and causes a branch if I is clear (if interrupts are enabled). See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BMC <i>rel</i>	REL	2C	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BMI

## Branch if Minus

# BMI

**Operation**

If (N) = 1,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch; may be used with signed or unsigned operations

**Description**

Tests the state of the N bit in the CCR and causes a branch if N is set.

Simply loading or storing A, X, or H:X will cause the N condition code bit to be set or cleared to match the most significant bit of the value loaded or stored. The BMI instruction can be used after such a load or store without having to do a separate test or compare instruction before the conditional branch. See the [BRA](#) instruction for further details of the execution of the branch.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BMI <i>rel</i>	REL	2B	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BMS

## Branch if Interrupt Mask Set

# BMS

### Operation

If (I) = 1,  $PC \leftarrow (PC) + \$0002 + rel$

### Description

Tests the state of the I bit in the CCR and causes a branch if I is set (if interrupts are disabled). See [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BMS <i>rel</i>	REL	2D	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BNE

## Branch if Not Equal

# BNE

### Operation

If  $(Z) = 0$ ,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch, may be used with signed or unsigned operations

### Description

Tests the state of the Z bit in the CCR and causes a branch if Z is clear

Following a compare or subtract instruction, the branch will occur if the arguments were not equal. This instruction can also be used after a load or store without having to do a separate test or compare on the loaded value. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BNE <i>rel</i>	REL	26	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BPL

## Branch if Plus

# BPL

### Operation

If (N) = 0,  $PC \leftarrow (PC) + \$0002 + rel$

Simple branch

### Description

Tests the state of the N bit in the CCR and causes a branch if N is clear

Simply loading or storing A, X, or H:X will cause the N condition code bit to be set or cleared to match the most significant bit of the value loaded or stored. The BPL instruction can be used after such a load or store without having to do a separate test or compare instruction before the conditional branch. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BPL <i>rel</i>	REL	2A	rr	3

See the [BRA](#) instruction for a summary of all branches and their complements.

# BRA

## Branch Always

# BRA

**Operation**

$PC \leftarrow (PC) + \$0002 + rel$

**Description**

Performs an unconditional branch to the address given in the foregoing formula. In this formula, *rel* is the two's-complement relative offset in the last byte of machine code for the instruction and (PC) is the address of the opcode for the branch instruction.

A source program specifies the destination of a branch instruction by its absolute address, either as a numerical value or as a symbol or expression which can be numerically evaluated by the assembler. The assembler calculates the 8-bit relative offset *rel* from this absolute address and the current value of the location counter.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BRA <i>rel</i>	REL	20	rr	3

The table on the facing page is a summary of all branch instructions.

*The BRA description continues next page.*

# BRA

## Branch Always (Continued)

# BRA

### Branch Instruction Summary

**Table 5-1** is a summary of all branch instructions.

**Table 5-1. Branch Instruction Summary**

Branch				Complementary Branch			Type
Test	Boolean	Mnemonic	Opcode	Test	Mnemonic	Opcode	
r>m	(Z)   (N⊕V)=0	BGT	92	r≤m	BLE	93	Signed
r≥m	(N⊕V)=0	BGE	90	r<m	BLT	91	Signed
r=m	(Z)=1	BEQ	27	r≠m	BNE	26	Signed
r≤m	(Z)   (N⊕V)=1	BLE	93	r>m	BGT	92	Signed
r<m	(N⊕V)=1	BLT	91	r≥m	BGE	90	Signed
r>m	(C)   (Z)=0	BHI	22	r≤m	BLS	23	Unsigned
r≥m	(C)=0	BHS/BCC	24	r<m	BLO/BCS	25	Unsigned
r=m	(Z)=1	BEQ	27	r≠m	BNE	26	Unsigned
r≤m	(C)   (Z)=1	BLS	23	r>m	BHI	22	Unsigned
r<m	(C)=1	BLO/BCS	25	r≥m	BHS/BCC	24	Unsigned
Carry	(C)=1	BCS	25	No carry	BCC	24	Simple
result=0	(Z)=1	BEQ	27	result≠0	BNE	26	Simple
Negative	(N)=1	BMI	2B	Plus	BPL	2A	Simple
l mask	(l)=1	BMS	2D	l mask=0	BMC	2C	Simple
H-Bit	(H)=1	BHCS	29	H=0	BHCC	28	Simple
$\overline{\text{IRQ}}$ high	—	BIH	2F	—	BIL	2E	Simple
Always	—	BRA	20	Never	BRN	21	Uncond.

r = register: A, X, or H:X (for CPHX instruction)    m = memory operand

During program execution, if the tested condition is true, the two's complement offset is sign-extended to a 16-bit value which is added to the current program counter. This causes program execution to continue at the address specified as the branch destination. If the tested condition is not true, the program simply continues to the next instruction after the branch.

# BRCLR *n*

Branch if Bit *n* in Memory Clear

# BRCLR *n*

**Operation**

If bit *n* of M = 0, PC ← (PC) + \$0003 + *rel*

**Description**

Tests bit *n* (*n* = 7, 6, 5, ... 0) of location M and branches if the bit is clear. M can be any RAM or I/O register address in the \$0000 to \$00FF area of memory because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRCLR *n* provides an easy method for performing serial-to-parallel conversions.

**Condition Codes and Boolean Formulae**

V			H	I	N	Z	C
—	1	1	—	—	—	—	↓

C: Set if  $M_n = 1$ ; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code			HC08 Cycles
		Opcode	Operand(s)		
BRCLR 0, <i>opr8a,rel</i>	DIR (b0)	01	dd	rr	5
BRCLR 1, <i>opr8a,rel</i>	DIR (b1)	03	dd	rr	5
BRCLR 2, <i>opr8a,rel</i>	DIR (b2)	05	dd	rr	5
BRCLR 3, <i>opr8a,rel</i>	DIR (b3)	07	dd	rr	5
BRCLR 4, <i>opr8a,rel</i>	DIR (b4)	09	dd	rr	5
BRCLR 5, <i>opr8a,rel</i>	DIR (b5)	0B	dd	rr	5
BRCLR 6, <i>opr8a,rel</i>	DIR (b6)	0D	dd	rr	5
BRCLR 7, <i>opr8a,rel</i>	DIR (b7)	0F	dd	rr	5



# BRN

## Branch Never

# BRN

### Operation

$PC \leftarrow (PC) + \$0002$

### Description

Never branches. In effect, this instruction can be considered a 2-byte no operation (NOP) requiring three cycles for execution. Its inclusion in the instruction set provides a complement for the **BRA** instruction. The BRN instruction is useful during program debugging to negate the effect of another branch instruction without disturbing the offset byte.

This instruction can be useful in instruction-based timing delays. Instruction-based timing delays are usually discouraged because such code is not portable to systems with different clock speeds.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BRN <i>rel</i>	REL	21	rr	3

See the **BRA** instruction for a summary of all branches and their complements.

# BRSET *n*

Branch if Bit *n* in Memory Set

# BRSET *n*

**Operation**

If bit *n* of M = 1, PC ← (PC) + \$0003 + *rel*

**Description**

Tests bit *n* (*n* = 7, 6, 5, ... 0) of location M and branches if the bit is set. M can be any RAM or I/O register address in the \$0000 to \$00FF area of memory because direct addressing mode is used to specify the address of the operand.

The C bit is set to the state of the tested bit. When used with an appropriate rotate instruction, BRSET *n* provides an easy method for performing serial-to-parallel conversions.

**Condition Codes and Boolean Formulae**

V		H		I	N	Z	C
—	1	1	—	—	—	—	↓

C: Set if  $M_n = 1$ ; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code			HC08 Cycles
		Opcode	Operand(s)		
BRSET 0, <i>opr8a,rel</i>	DIR (b0)	00	dd	rr	5
BRSET 1, <i>opr8a,rel</i>	DIR (b1)	02	dd	rr	5
BRSET 2, <i>opr8a,rel</i>	DIR (b2)	04	dd	rr	5
BRSET 3, <i>opr8a,rel</i>	DIR (b3)	06	dd	rr	5
BRSET 4, <i>opr8a,rel</i>	DIR (b4)	08	dd	rr	5
BRSET 5, <i>opr8a,rel</i>	DIR (b5)	0A	dd	rr	5
BRSET 6, <i>opr8a,rel</i>	DIR (b6)	0C	dd	rr	5
BRSET 7, <i>opr8a,rel</i>	DIR (b7)	0E	dd	rr	5

# BSET *n*

## Set Bit *n* in Memory

# BSET *n*

**Operation**  $Mn \leftarrow 1$

**Description** Set bit *n* ( $n = 7, 6, 5, \dots, 0$ ) in location M. All other bits in M are unaffected. M can be any RAM or I/O register address in the \$0000 to \$00FF area of memory because direct addressing mode is used to specify the address of the operand. This instruction reads the specified 8-bit location, modifies the specified bit, and then writes the modified 8-bit value back to the memory location.

**Condition Codes and Boolean Formulae** None affected

V	H	I	N	Z	C
—	1	1	—	—	—

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BSET 0, <i>opr8a</i>	DIR (b0)	10	dd	4
BSET 1, <i>opr8a</i>	DIR (b1)	12	dd	4
BSET 2, <i>opr8a</i>	DIR (b2)	14	dd	4
BSET 3, <i>opr8a</i>	DIR (b3)	16	dd	4
BSET 4, <i>opr8a</i>	DIR (b4)	18	dd	4
BSET 5, <i>opr8a</i>	DIR (b5)	1A	dd	4
BSET 6, <i>opr8a</i>	DIR (b6)	1C	dd	4
BSET 7, <i>opr8a</i>	DIR (b7)	1E	dd	4

# BSR

## Branch to Subroutine

# BSR

### Operation

$PC \leftarrow (PC) + \$0002$	Advance PC to return address
Push (PCL); $SP \leftarrow (SP) - \$0001$	Push low half of return address
Push (PCH); $SP \leftarrow (SP) - \$0001$	Push high half of return address
$PC \leftarrow (PC) + rel$	Load PC with start address of requested subroutine

### Description

The program counter is incremented by 2 from the opcode address (so it points to the opcode of the next instruction which will be the return address). The least significant byte of the contents of the program counter (low-order return address) is pushed onto the stack. The stack pointer is then decremented by 1. The most significant byte of the contents of the program counter (high-order return address) is pushed onto the stack. The stack pointer is then decremented by 1. A branch then occurs to the location specified by the branch offset. See the [BRA](#) instruction for further details of the execution of the branch.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
BSR <i>rel</i>	REL	AD	rr	4

# CBEQ

## Compare and Branch if Equal

# CBEQ

### Operation

For DIR or IMM modes: if (A) = (M), PC ← (PC) + \$0003 + *rel*  
**Or** for IX+ mode: if (A) = (M); PC ← (PC) + \$0002 + *rel*  
**Or** for SP1 mode: if (A) = (M); PC ← (PC) + \$0004 + *rel*  
**Or** for CBEQX: if (X) = (M); PC ← (PC) + \$0003 + *rel*

### Description

CBEQ compares the operand with the accumulator (or index register for CBEQX instruction) against the contents of a memory location and causes a branch if the register (A or X) is equal to the memory contents. The CBEQ instruction combines CMP and BEQ for faster table lookup routines and condition codes are not changed.

The IX+ variation of the CBEQ instruction compares the operand addressed by H:X to A and causes a branch if the operands are equal. H:X is then incremented regardless of whether a branch is taken. The IX1+ variation of CBEQ operates the same way except that an 8-bit offset is added to H:X to form the effective address of the operand.

### Condition Codes and Boolean Formulae

None affected

V		H	I	N	Z	C
—	1	1	—	—	—	—

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code			HC08 Cycles
		Opcode	Operand(s)		
CBEQ <i>opr8a,rel</i>	DIR	31	dd	rr	5
CBEQA <i>#opr8i,rel</i>	IMM	41	ii	rr	4
CBEQX <i>#opr8i,rel</i>	IMM	51	ii	rr	4
CBEQ <i>opr8,X+,rel</i>	IX1+	61	ff	rr	5
CBEQ <i>,X+,rel</i>	IX+	71	rr		4
CBEQ <i>opr8,SP,rel</i>	SP1	9E61	ff	rr	6

# CLC

## Clear Carry Bit

# CLC

**Operation**

C bit ← 0

**Description**

Clears the C bit in the CCR. CLC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit. The C bit can also be used to pass status information between a subroutine and the calling program.

**Condition Codes and Boolean Formulae**

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	0

C: 0  
Cleared

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CLC	INH	98		1

# CLI

## Clear Interrupt Mask Bit

# CLI

### Operation

I bit ← 0

### Description

Clears the interrupt mask bit in the CCR. When the I bit is clear, interrupts are enabled. The I bit actually changes to zero at the end of the cycle where the CLI instruction executes. This is too late to recognize an interrupt that arrived before or during the CLI instruction so if interrupts were previously disabled, the next instruction after a CLI will always be executed even if there was an interrupt pending prior to execution of the CLI instruction.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0  
Cleared

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CLI	INH	9A		2

# CLR

Clear

# CLR

## Operation

$A \leftarrow \$00$   
**Or**  $M \leftarrow \$00$   
**Or**  $X \leftarrow \$00$   
**Or**  $H \leftarrow \$00$

## Description

The contents of memory (M), A, X, or H are replaced with zeros.

## Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	0	1	—

V: 0  
Cleared

N: 0  
Cleared

Z: 1  
Set

## Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CLR <i>opr8a</i>	DIR	3F	dd	3
CLRA	INH (A)	4F		1
CLR X	INH (X)	5F		1
CLR H	INH (H)	8C		1
CLR <i>opr8,X</i>	IX1	6F	ff	3
CLR <i>,X</i>	IX	7F		2
CLR <i>opr8,SP</i>	SP1	9E6F	ff	4



# CMP

## Compare Accumulator with Memory

# CMP

### Operation

(A) – (M)

### Description

Compares the contents of A to the contents of M and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both A and M are unchanged.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
↑	1	1	—	—	↓	↓

V:  $A7 \& \overline{M7} \& \overline{R7} \mid \overline{A7} \& M7 \& R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C:  $\overline{A7} \& M7 \mid M7 \& R7 \mid R7 \& \overline{A7}$

Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CMP #opr8i	IMM	A1	ii	2
CMP opr8a	DIR	B1	dd	3
CMP opr16a	EXT	C1	hh ll	4
CMP oprx16,X	IX2	D1	ee ff	4
CMP oprx8,X	IX1	E1	ff	3
CMP ,X	IX	F1		2
CMP oprx16,SP	SP2	9ED1	ee ff	5
CMP oprx8,SP	SP1	9EE1	ff	4

# COM

## Complement (One's Complement)

# COM

### Operation

$A \leftarrow \bar{A} = \$FF - (A)$   
**Or**  $X \leftarrow \bar{X} = \$FF - (X)$   
**Or**  $M \leftarrow \bar{M} = \$FF - (M)$

### Description

Replaces the contents of A, X, or M with the one's complement. Each bit of A, X, or M is replaced with the complement of that bit.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↓	↓	1

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\bar{R7} \& \bar{R6} \& \bar{R5} \& \bar{R4} \& \bar{R3} \& \bar{R2} \& \bar{R1} \& \bar{R0}$   
Set if result is \$00; cleared otherwise

C: 1  
Set

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
COM <i>opr8a</i>	DIR	33	dd	4
COMA	INH (A)	43		1
COMX	INH (X)	53		1
COM <i>opr8,X</i>	IX1	63	ff	4
COM <i>,X</i>	IX	73		3
COM <i>opr8,SP</i>	SP1	9E63	ff	5

# CPHX

## Compare Index Register with Memory

# CPHX

### Operation

(H:X) – (M:M + \$0001)

### Description

CPHX compares index register (H:X) with the 16-bit value in memory and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both H:X and M:M + \$0001 are unchanged.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↑	1	1	—	—	↓	↓	↓

V:  $\overline{H7 \& M15 \& R15} \mid \overline{H7 \& M15 \& R15}$

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

N: R15

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R15 \& R14 \& R13 \& R12 \& R11 \& R10 \& R9 \& R8}$   
 $\& R7 \& R6 \& R5 \& R4 \& R3 \& R2 \& R1 \& R0$

Set if the result is \$0000; cleared otherwise

C:  $\overline{H7 \& M15} \mid M15 \& R15 \mid R15 \& \overline{H7}$

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

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CPHX #opr	IMM	65	jj kk+1	3
CPHX opr	DIR	75	dd	4

# CPX

## Compare X (Index Register Low) with Memory

# CPX

**Operation** (X) – (M)

**Description** Compares the contents of X to the contents of M and sets the condition codes, which may then be used for arithmetic (signed or unsigned) and logical conditional branching. The contents of both X and M are unchanged.

**Condition Codes and Boolean Formulae**

V		H	I	N	Z	C
↑	1	1	—	—	↓	↓

V:  $X7 \& \overline{M7} \& \overline{R7} \mid \overline{X7} \& M7 \& R7$   
 Set if a two's complement overflow resulted from the operation; cleared otherwise

N: R7  
 Set if MSB of result of the subtraction is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
 Set if result is \$00; cleared otherwise

C:  $\overline{X7} \& M7 \mid M7 \& R7 \mid R7 \& \overline{X7}$   
 Set if the unsigned value of the contents of memory is larger than the unsigned value in the index register; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
CPX #opr8i	IMM	A3	ii	2
CPX opr8a	DIR	B3	dd	3
CPX opr16a	EXT	C3	hh ll	4
CPX oprx16,X	IX2	D3	ee ff	4
CPX oprx8,X	IX1	E3	ff	3
CPX ,X	IX	F3		2
CPX oprx16,SP	SP2	9ED3	ee ff	5
CPX oprx8,SP	SP1	9EE3	ff	4

# DAA

## Decimal Adjust Accumulator

# DAA

### Operation

$(A)_{10}$

### Description

Adjusts the contents of the accumulator and the state of the CCR carry bit after an ADD or ADC operation involving binary-coded decimal (BCD) values, so that there is a correct BCD sum and an accurate carry indication. The state of the CCR half carry bit affects operation. Refer to [Table 5-2](#) for details of operation.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
U	1	1	—	—	↓	↓	↓

V: U

Undefined

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: Set if the decimal adjusted result is greater than 99 (decimal); refer to [Table 5-2](#)

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
DAA	INH	72		2

*The DAA description continues next page.*

## DAA

## Decimal Adjust Accumulator (Continued)

## DAA

**Table 5-2** shows DAA operation for all legal combinations of input operands. Columns 1–4 represent the results of 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 in this table are hexadecimal.

Table 5-2. DAA Function Summary

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

# DBNZ

## Decrement and Branch if Not Zero

# DBNZ

### Operation

$A \leftarrow (A) - \$01$

**Or**  $M \leftarrow (M) - \$01$

**Or**  $X \leftarrow (X) - \$01$

For DIR or IX1 modes:  $PC \leftarrow (PC) + \$0003 + rel$  if (result)  $\neq 0$

**Or** for INH or IX modes:  $PC \leftarrow (PC) + \$0002 + rel$  if (result)  $\neq 0$

**Or** for SP1 mode:  $PC \leftarrow (PC) + \$0004 + rel$  if (result)  $\neq 0$

### Description

Subtract 1 from the contents of A, M, or X; then branch using the relative offset if the result of the subtraction is not \$00. DBNZX only affects the low order eight bits of the H:X index register pair; the high-order byte (H) is not affected.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
DBNZ <i>opr8a,rel</i>	DIR	3B	dd rr	5
DBNZ <i>rel</i>	INH	4B	rr	3
DBNZX <i>rel</i>	INH	5B	rr	3
DBNZ <i>opr8,X,rel</i>	IX1	6B	ff rr	5
DBNZ <i>,X,rel</i>	IX	7B	rr	4
DBNZ <i>opr8,SP,rel</i>	SP1	9E6B	ff rr	6

# DEC

## Decrement

# DEC

### Operation

$A \leftarrow (A) - \$01$   
**Or**  $X \leftarrow (X) - \$01$   
**Or**  $M \leftarrow (M) - \$01$

### Description

Subtract 1 from the contents of A, X, or M. The V, N, and Z bits in the CCR are set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BLS, BLO, BHS, and BHI branch instructions are not useful following a DEC instruction.

DECX only affects the low-order byte of index register pair (H:X). To decrement the full 16-bit index register pair (H:X), use AIX # -1.

### Condition Codes and Boolean Formulae

V	H	I	N	Z	C
↑	1	1	—	—	—

V:  $\overline{R7} \& A7$

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), (X), or (M) was \$80 before the operation.

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
DEC <i>opr8a</i>	DIR	3A	dd	4
DECA	INH (A)	4A		1
DECX	INH (X)	5A		1
DEC <i>opr8,X</i>	IX1	6A	ff	4
DEC <i>,X</i>	IX	7A		3
DEC <i>opr8,SP</i>	SP1	9E6A	ff	5

DEX is recognized by assemblers as being equivalent to DECX.



# DIV

## Divide

# DIV

### Operation

$A \leftarrow (H:A) \div (X); H \leftarrow \text{Remainder}$

### Description

Divides a 16-bit unsigned dividend contained in the concatenated registers H and A by an 8-bit divisor contained in X. The quotient is placed in A, and the remainder is placed in H. The divisor is left unchanged.

An overflow (quotient > \$FF) or divide-by-0 sets the C bit, and the quotient and remainder are indeterminate.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	↓	↓

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result (quotient) is \$00; cleared otherwise

C: Set if a divide-by-0 was attempted or if an overflow occurred; cleared otherwise

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
DIV	INH	52		7

# EOR

## Exclusive-OR Memory with Accumulator

# EOR

### Operation

$$A \leftarrow (A \oplus M)$$

### Description

Performs the logical exclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical exclusive-OR of the corresponding bits of M and A before the operation.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
0	1	1	—	—	↓	—

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
EOR # <i>opr8i</i>	IMM	A8	ii	2
EOR <i>opr8a</i>	DIR	B8	dd	3
EOR <i>opr16a</i>	EXT	C8	hh ll	4
EOR <i>opr16,X</i>	IX2	D8	ee ff	4
EOR <i>opr8,X</i>	IX1	E8	ff	3
EOR ,X	IX	F8		2
EOR <i>opr16,SP</i>	SP2	9ED8	ee ff	5
EOR <i>opr8,SP</i>	SP1	9EE8	ff	4

# INC

## Increment

# INC

### Operation

$A \leftarrow (A) + \$01$   
**Or**  $X \leftarrow (X) + \$01$   
**Or**  $M \leftarrow (M) + \$01$

### Description

Add 1 to the contents of A, X, or M. The V, N, and Z bits in the CCR are set or cleared according to the results of this operation. The C bit in the CCR is not affected; therefore, the BLS, BLO, BHS, and BHI branch instructions are not useful following an INC instruction.

INCX only affects the low-order byte of index register pair (H:X). To increment the full 16-bit index register pair (H:X), use AIX #1.

### Condition Codes and Boolean Formulae

V	H	I	N	Z	C
↑	1	1	—	—	↓

V:  $\overline{A7 \& R7}$

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), (X), or (M) was \$7F before the operation.

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7 \& R6 \& R5 \& R4 \& R3 \& R2 \& R1 \& R0}$

Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
INC <i>opr8a</i>	DIR	3C	dd	4
INCA	INH (A)	4C		1
INCX	INH (X)	5C		1
INC <i>opr8,X</i>	IX1	6C	ff	4
INC <i>,X</i>	IX	7C		3
INC <i>opr8,SP</i>	SP1	9E6C	ff	5

INX is recognized by assemblers as being equivalent to INCX.

# JMP

## Jump

# JMP

**Operation** PC ← effective address

**Description** A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for extended, direct, or indexed addressing.

**Condition Codes and Boolean Formulae** None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
JMP <i>opr8a</i>	DIR	BC	dd	2
JMP <i>opr16a</i>	EXT	CC	hh ll	3
JMP <i>opr16,X</i>	IX2	DC	ee ff	4
JMP <i>opr8,X</i>	IX1	EC	ff	3
JMP <i>,X</i>	IX	FC		3

# JSR

## Jump to Subroutine

# JSR

### Operation

$PC \leftarrow (PC) + n$ ;

$n = 1, 2,$  or  $3$  depending on address mode

Push (PCL);  $SP \leftarrow (SP) - \$0001$     Push low half of return address

Push (PCH);  $SP \leftarrow (SP) - \$0001$     Push high half of return address

$PC \leftarrow$  effective address                    Load PC with start address of  
requested subroutine

### Description

The program counter is incremented by  $n$  so that it points to the opcode of the next instruction that follows the JSR instruction ( $n = 1, 2,$  or  $3$  depending on the addressing mode). The PC is then pushed onto the stack, eight bits at a time, least significant byte first. The stack pointer points to the next empty location on the stack. A jump occurs to the instruction stored at the effective address. The effective address is obtained according to the rules for extended, direct, or indexed addressing.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
JSR <i>opr8a</i>	DIR	BD	dd	4
JSR <i>opr16a</i>	EXT	CD	hh ll	5
JSR <i>opr16,X</i>	IX2	DD	ee ff	6
JSR <i>opr8,X</i>	IX1	ED	ff	5
JSR <i>,X</i>	IX	FD		4

# LDA

## Load Accumulator from Memory

# LDA

**Operation**

$A \leftarrow (M)$

**Description**

Loads the contents of the specified memory location into A. The N and Z condition codes are set or cleared according to the loaded data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

**Condition Codes and Boolean Formulae**

V		H	I	N	Z	C
0	1	1	—	—	↓	↓

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
LDA #opr8i	IMM	A6	ii	2
LDA opr8a	DIR	B6	dd	3
LDA opr16a	EXT	C6	hh ll	4
LDA oprx16,X	IX2	D6	ee ff	4
LDA oprx8,X	IX1	E6	ff	3
LDA ,X	IX	F6		2
LDA oprx16,SP	SP2	9ED6	ee ff	5
LDA oprx8,SP	SP1	9EE6	ff	4

# LDHX

## Load Index Register from Memory

# LDHX

### Operation

$H:X \leftarrow (M:M + \$0001)$

### Description

Loads the contents of the specified memory location into the index register (H:X). The N and Z condition codes are set according to the data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
0	1	1	—	—	↓	↓

V: 0  
Cleared

N: R15  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R15 \& R14 \& R13 \& R12 \& R11 \& R10 \& R9 \& R8 \& R7 \& R6 \& R5 \& R4 \& R3 \& R2 \& R1 \& R0}$   
Set if the result is \$0000; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
LDHX #opr	IMM	45	jj kk	3
LDHX opr	DIR	55	dd	4

# LDX

## Load X (Index Register Low) from Memory

# LDX

**Operation**  $X \leftarrow (M)$

**Description** Loads the contents of the specified memory location into X. The N and Z condition codes are set or cleared according to the loaded data; V is cleared. This allows conditional branching after the load without having to perform a separate test or compare.

**Condition Codes and Boolean Formulae**

V		H	I	N	Z	C
0	1	1	—	—	↓	↓

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
LDX #opr8i	IMM	AE	ii	2
LDX opr8a	DIR	BE	dd	3
LDX opr16a	EXT	CE	hh ll	4
LDX oprx16,X	IX2	DE	ee ff	4
LDX oprx8,X	IX1	EE	ff	3
LDX ,X	IX	FE		2
LDX oprx16,SP	SP2	9EDE	ee ff	5
LDX oprx8,SP	SP1	9EEE	ff	4

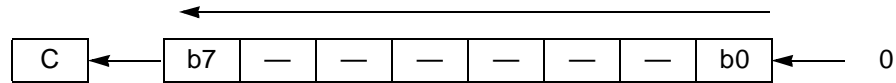


# LSL

## Logical Shift Left (Same as ASL)

# LSL

### Operation



### Description

Shifts all bits of the A, X, or M one place to the left. Bit 0 is loaded with a 0. The C bit in the CCR is loaded from the most significant bit of A, X, or M.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↓	1	1	—	—	↓	↓	↓

V:  $R7 \oplus b7$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: b7

Set if, before the shift, the MSB of A, X, or M was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

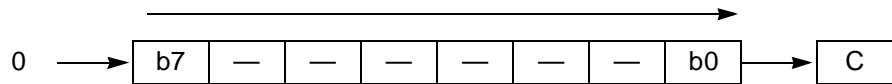
Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
LSL <i>opr8a</i>	DIR	38	dd	4
LSLA	INH (A)	48		1
LSLX	INH (X)	58		1
LSL <i>opr8,X</i>	IX1	68	ff	4
LSL <i>,X</i>	IX	78		3
LSL <i>opr8,SP</i>	SP1	9E68	ff	5

# LSR

## Logical Shift Right

# LSR

### Operation



### Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is loaded with a 0. Bit 0 is shifted into the C bit.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
↑	1	1	—	—	0	↓

V:  $0 \oplus b0 = b0$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise. Since  $N = 0$ , this simplifies to the value of bit 0 before the shift.

N: 0  
Cleared

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

C: b0  
Set if, before the shift, the LSB of A, X, or M, was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
LSR <i>opr8a</i>	DIR	34	dd	4
LSRA	INH (A)	44		1
LSRX	INH (X)	54		1
LSR <i>opr8,X</i>	IX1	64	ff	4
LSR <i>,X</i>	IX	74		3
LSR <i>opr8,SP</i>	SP1	9E64	ff	5

# MOV

## Move

# MOV

**Operation**  $(M)_{\text{Destination}} \leftarrow (M)_{\text{Source}}$

**Description** Moves a byte of data from a source address to a destination address. Data is examined as it is moved, and condition codes are set. Source data is not changed. The accumulator is not affected.

The four addressing modes for the MOV instruction are:

1. IMM/DIR moves an immediate byte to a direct memory location.
2. DIR/DIR moves a direct location byte to another direct location.
3. IX+/DIR moves a byte from a location addressed by H:X to a direct location. H:X is incremented after the move.
4. DIR/IX+ moves a byte from a direct location to one addressed by H:X. H:X is incremented after the move.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↓	↓	—

V: 0  
Cleared

N: R7  
Set if MSB of result is set; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
MOV <i>opr8a,opr8a</i>	DIR/DIR	4E	dd dd	5
MOV <i>opr8a,X+</i>	DIR/IX+	5E	dd	4
MOV <i>#opr8i,opr8a</i>	IMM/DIR	6E	ii dd	4
MOV <i>,X+,opr8a</i>	IX+/DIR	7E	dd	4

# MUL

## Unsigned Multiply

# MUL

**Operation**  $X:A \leftarrow (X) \times (A)$

**Description** Multiplies the 8-bit value in X (index register low) by the 8-bit value in the accumulator to obtain a 16-bit unsigned result in the concatenated index register and accumulator. After the operation, X contains the upper eight bits of the 16-bit result and A contains the lower eight bits of the result.

**Condition Codes and Boolean Formulae**

V			H		I		N		Z		C
—	1	1	0	—	—	—	—	—	—	—	0

H: 0  
Cleared

C: 0  
Cleared

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
MUL	INH	42		5

# NEG

## Negate (Two's Complement)

# NEG

### Operation

$A \leftarrow - (A)$

**Or**  $X \leftarrow - (X)$

**Or**  $M \leftarrow - (M);$

this is equivalent to subtracting A, X, or M from \$00

### Description

Replaces the contents of A, X, or M with its two's complement. Note that the value \$80 is left unchanged.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↓	1	1	—	—	↓	↓	↓

V: M7&R7

Set if a two's complement overflow resulted from the operation; cleared otherwise. Overflow will occur only if the operand is \$80 before the operation.

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C:  $R7|R6|R5|R4|R3|R2|R1|R0$

Set if there is a borrow in the implied subtraction from 0; cleared otherwise. The C bit will be set in all cases except when the contents of A, X, or M was \$00 prior to the NEG operation.

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
NEG <i>opr8a</i>	DIR	30	dd	4
NEGA	INH (A)	40		1
NEGX	INH (X)	50		1
NEG <i>opr8,X</i>	IX1	60	ff	4
NEG <i>,X</i>	IX	70		3
NEG <i>opr8,SP</i>	SP1	9E60	ff	5

# NOP

## No Operation

# NOP

**Operation** Uses one bus cycle

**Description** This is a single-byte instruction that does nothing except to consume one CPU clock cycle while the program counter is advanced to the next instruction. No register or memory contents are affected by this instruction.

**Condition Codes and Boolean Formulae** None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
NOP	INH	9D		1

# NSA

## Nibble Swap Accumulator

# NSA

### Operation

$A \leftarrow (A[3:0]:A[7:4])$

### Description

Swaps upper and lower nibbles (4 bits) of the accumulator. The NSA instruction is used for more efficient storage and use of binary-coded decimal operands.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
NSA	INH	62		3

# ORA

## Inclusive-OR Accumulator and Memory

# ORA

### Operation

$$A \leftarrow (A) | (M)$$

### Description

Performs the logical inclusive-OR between the contents of A and the contents of M and places the result in A. Each bit of A after the operation will be the logical inclusive-OR of the corresponding bits of M and A before the operation.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
0	1	1	—	—	↓	—

V: 0  
Cleared

N: R7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ORA #opr8i	IMM	AA	ii	2
ORA opr8a	DIR	BA	dd	3
ORA opr16a	EXT	CA	hh ll	4
ORA oprx16,X	IX2	DA	ee ff	4
ORA oprx8,X	IX1	EA	ff	3
ORA ,X	IX	FA		2
ORA oprx16,SP	SP2	9EDA	ee ff	5
ORA oprx8,SP	SP1	9EEA	ff	4



# PSHA

## Push Accumulator onto Stack

# PSHA

### Operation

Push (A);  $SP \leftarrow (SP) - \$0001$

### Description

The contents of A are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point to the next available location in the stack. The contents of A remain unchanged.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PSHA	INH	87		2

# PSHH

## Push H (Index Register High) onto Stack

# PSHH

**Operation** Push (H);  $SP \leftarrow (SP) - \$0001$

**Description** The contents of H are pushed onto the stack at the address contained in the stack pointer. The stack pointer is then decremented to point to the next available location in the stack. The contents of H remain unchanged.

**Condition Codes and Boolean Formulae** None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PSHH	INH	8B		2

# PSHX

## Push X (Index Register Low) onto Stack

# PSHX

### Operation

Push (X);  $SP \leftarrow (SP) - \$0001$

### Description

The contents of X are pushed onto the stack at the address contained in the stack pointer (SP). SP is then decremented to point to the next available location in the stack. The contents of X remain unchanged.

### Condition Codes and Boolean Formulae

None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PSHX	INH	89		2

# PULA

## Pull Accumulator from Stack

# PULA

**Operation**  $SP \leftarrow (SP + \$0001); \text{pull } (A)$

**Description** The stack pointer (SP) is incremented to address the last operand on the stack. The accumulator is then loaded with the contents of the address pointed to by SP.

**Condition Codes and Boolean Formulae** None affected

V			H		I		N		Z		C
—	1	1	—	—	—	—	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PULA	INH	86		2

# PULH

## Pull H (Index Register High) from Stack

# PULH

### Operation

$SP \leftarrow (SP + \$0001)$ ; pull (H)

### Description

The stack pointer (SP) is incremented to address the last operand on the stack. H is then loaded with the contents of the address pointed to by SP.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PULH	INH	8A		2

# PULX

## Pull X (Index Register Low) from Stack

# PULX

**Operation**

$SP \leftarrow (SP + \$0001)$ ; pull (X)

**Description**

The stack pointer (SP) is incremented to address the last operand on the stack. X is then loaded with the contents of the address pointed to by SP.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

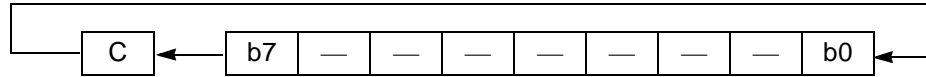
Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
PULX	INH	88		2

# ROL

## Rotate Left through Carry

# ROL

### Operation



### Description

Shifts all bits of A, X, or M one place to the left. Bit 0 is loaded from the C bit. The C bit is loaded from the most significant bit of A, X, or M. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes. For example, to shift a 24-bit value left one bit, the sequence (ASL LOW, ROL MID, ROL HIGH) could be used, where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↓	1	1	—	—	↓	↓	↓

V:  $R7 \oplus b7$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: b7

Set if, before the rotate, the MSB of A, X, or M was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

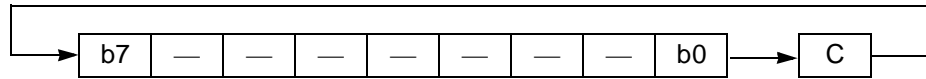
Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ROL <i>opr8a</i>	DIR	39	dd	4
ROLA	INH (A)	49		1
ROLX	INH (X)	59		1
ROL <i>opr8,X</i>	IX1	69	ff	4
ROL <i>,X</i>	IX	79		3
ROL <i>opr8,SP</i>	SP1	9E69	ff	5

# ROR

## Rotate Right through Carry

# ROR

### Operation



### Description

Shifts all bits of A, X, or M one place to the right. Bit 7 is loaded from the C bit. Bit 0 is shifted into the C bit. The rotate instructions include the carry bit to allow extension of the shift and rotate instructions to multiple bytes. For example, to shift a 24-bit value right one bit, the sequence (LSR HIGH, ROR MID, ROR LOW) could be used, where LOW, MID, and HIGH refer to the low-order, middle, and high-order bytes of the 24-bit value, respectively.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↕	1	1	—	—	↕	↕	↕

V:  $R7 \oplus b0$

Set if the exclusive-OR of the resulting N and C flags is 1; cleared otherwise

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C: b0

Set if, before the shift, the LSB of A, X, or M was set; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
ROR <i>opr8a</i>	DIR	36	dd	4
RORA	INH (A)	46		1
RORX	INH (X)	56		1
ROR <i>opr8,X</i>	IX1	66	ff	4
ROR <i>,X</i>	IX	76		3
ROR <i>opr8,SP</i>	SP1	9E66	ff	5



# RSP

## Reset Stack Pointer

# RSP

**Operation** SP ← \$FF

**Description** In most M68HC05 MCUs, RAM only goes to \$00FF. In most HC08s, however, RAM extends beyond \$00FF. Therefore, do not locate the stack in direct address space which is more valuable for commonly accessed variables. In new HC08 programs, it is more appropriate to initialize the stack pointer to the address of the last location (highest address) in the on-chip RAM, shortly after reset. This code segment demonstrates a typical method for initializing SP.

```
LDHX  #ram_end+1    ; Point at next addr past RAM
TXS                                ; SP <-(H:X)-1
```

**Condition Codes and Boolean Formulae** None affected

V	H	I	N	Z	C
—	1	1	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
RSP	INH	9C		1

# RTI

## Return from Interrupt

# RTI

### Operation

$SP \leftarrow SP + \$0001$ ; pull (CCR)    Restore CCR from stack  
 $SP \leftarrow SP + \$0001$ ; pull (A)       Restore A from stack  
 $SP \leftarrow SP + \$0001$ ; pull (X)       Restore X from stack  
 $SP \leftarrow SP + \$0001$ ; pull (PCH)    Restore PCH from stack  
 $SP \leftarrow SP + \$0001$ ; pull (PCL)    Restore PCL from stack

### Description

The condition codes, the accumulator, X (index register low), and the program counter are restored to the state previously saved on the stack. The I bit will be cleared if the corresponding bit stored on the stack is 0, the normal case.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↓	1	1	↓	↓	↓	↓	↓

Set or cleared according to the byte pulled from the stack into CCR.

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
RTI	INH	80		7

# RTS

## Return from Subroutine

# RTS

### Operation

SP ← SP + \$0001; pull (PCH) Restore PCH from stack  
 SP ← SP + \$0001; pull (PCL) Restore PCL from stack

### Description

The stack pointer is incremented by 1. The contents of the byte of memory that is pointed to by the stack pointer are loaded into the high-order byte of the program counter. The stack pointer is again incremented by 1. The contents of the byte of memory that are pointed to by the stack pointer are loaded into the low-order eight bits of the program counter. Program execution resumes at the address that was just restored from the stack.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
RTS	INH	81		4

# SBC

## Subtract with Carry

# SBC

**Operation**

$$A \leftarrow (A) - (M) - (C)$$

**Description**

Subtracts the contents of M and the contents of the C bit of the CCR from the contents of A and places the result in A. This is useful for multi-precision subtract algorithms involving operands with more than eight bits.

**Condition Codes and Boolean Formulae**

V		H	I	N	Z	C
↑	1	1	—	—	↑	↑

V:  $A7 \& \overline{M7} \& \overline{R7} \mid \overline{A7} \& M7 \& R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

N: R7

Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

C:  $\overline{A7} \& M7 \mid M7 \& R7 \mid R7 \& \overline{A7}$

Set if the unsigned value of the contents of memory plus the previous carry are larger than the unsigned value of the accumulator; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
SBC #opr8i	IMM	A2	ii	2
SBC opr8a	DIR	B2	dd	3
SBC opr16a	EXT	C2	hh ll	4
SBC oprx16,X	IX2	D2	ee ff	4
SBC oprx8,X	IX1	E2	ff	3
SBC ,X	IX	F2		2
SBC oprx16,SP	SP2	9ED2	ee ff	5
SBC oprx8,SP	SP1	9EE2	ff	4

# SEC

## Set Carry Bit

# SEC

### Operation

C bit  $\leftarrow$  1

### Description

Sets the C bit in the condition code register (CCR). SEC may be used to set up the C bit prior to a shift or rotate instruction that involves the C bit.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	—	—	—	1

C: 1  
Set

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
SEC	INH	99		1

# SEI

## Set Interrupt Mask Bit

# SEI

**Operation**

I bit ← 1

**Description**

Sets the interrupt mask bit in the condition code register (CCR). The microprocessor is inhibited from responding to interrupts while the I bit is set. The I bit actually changes at the end of the cycle where SEI executed. This is too late to stop an interrupt that arrived during execution of the SEI instruction so it is possible that an interrupt request could be serviced after the SEI instruction before the next instruction after SEI is executed. The global I-bit interrupt mask takes effect before the next instruction can be completed.

**Condition Codes and Boolean Formulae**

V			H	I	N	Z	C
—	1	1	—	1	—	—	—

I: 1  
Set

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
SEI	INH	9B		2

# STA

## Store Accumulator in Memory

# STA

### Operation

$M \leftarrow (A)$

### Description

Stores the contents of A in memory. The contents of A remain unchanged. The N condition code is set if the most significant bit of A is set, the Z bit is set if A was \$00, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↓	↓	—

V: 0  
Cleared

N: A7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{A7 \& A6 \& A5 \& A4 \& A3 \& A2 \& A1 \& A0}$   
Set if result is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
STA <i>opr8a</i>	DIR	B7	dd	3
STA <i>opr16a</i>	EXT	C7	hh ll	4
STA <i>opr16,X</i>	IX2	D7	ee ff	4
STA <i>opr8,X</i>	IX1	E7	ff	3
STA <i>,X</i>	IX	F7		2
STA <i>opr16,SP</i>	SP2	9ED7	ee ff	5
STA <i>opr8,SP</i>	SP1	9EE7	ff	4

# STHX

## Store Index Register

# STHX

### Operation

$(M:M + \$0001) \leftarrow (H:X)$

### Description

Stores the contents of H in memory location M and then the contents of X into the next memory location (M + \$0001). The N condition code bit is set if the most significant bit of H was set, the Z bit is set if the value of H:X was \$0000, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↓	↓	—

V: 0  
Cleared

N: R15  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{R15} \& \overline{R14} \& \overline{R13} \& \overline{R12} \& \overline{R11} \& \overline{R10} \& \overline{R9} \& \overline{R8} \& \overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$   
Set if the result is \$0000; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
STHX <i>opr</i>	DIR	35	dd	4



# STOP

## Enable $\overline{\text{IRQ}}$ Pin, Stop Oscillator

# STOP

### Operation

I bit  $\leftarrow$  0; stop oscillator

### Description

Reduces power consumption by eliminating all dynamic power dissipation. (See module documentation for module reactions to STOP instruction.) The external interrupt pin is enabled and the I bit in the condition code register (CCR) is cleared to enable the external interrupt. Finally, the oscillator is inhibited to put the MCU into the STOP condition.

When either the  $\overline{\text{RESET}}$  pin or  $\overline{\text{IRQ}}$  pin goes low, the oscillator is enabled. A delay of 4095 processor clock cycles is imposed allowing the oscillator to stabilize. The reset vector or interrupt request vector is fetched and the associated service routine is executed.

External interrupts are enabled after a STOP command.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0  
Cleared

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
STOP	INH	8E		1

# STX

## Store X (Index Register Low) in Memory

# STX

### Operation

$M \leftarrow (X)$

### Description

Stores the contents of X in memory. The contents of X remain unchanged. The N condition code is set if the most significant bit of X was set, the Z bit is set if X was \$00, and V is cleared. This allows conditional branching after the store without having to do a separate test or compare.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
0	1	1	—	—	↓	↓	—

V: 0  
Cleared

N: X7  
Set if MSB of result is 1; cleared otherwise

Z:  $\overline{X7 \& X6 \& X5 \& X4 \& X3 \& X2 \& X1 \& X0}$   
Set if X is \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
STX <i>opr8a</i>	DIR	BF	dd	3
STX <i>opr16a</i>	EXT	CF	hh ll	4
STX <i>opr16,X</i>	IX2	DF	ee ff	4
STX <i>opr8,X</i>	IX1	EF	ff	3
STX <i>,X</i>	IX	FF		2
STX <i>opr16,SP</i>	SP2	9EDF	ee ff	5
STX <i>opr8,SP</i>	SP1	9EEF	ff	4

# SUB

## Subtract

# SUB

**Operation**  $A \leftarrow (A) - (M)$

**Description** Subtracts the contents of M from A and places the result in A

**Condition Codes and Boolean Formulae**

V			H	I	N	Z	C
↑	1	1	—	—	↑	↑	↑

**V:**  $A7 \& \overline{M7} \& \overline{R7} \mid \overline{A7} \& M7 \& R7$

Set if a two's complement overflow resulted from the operation; cleared otherwise. Literally read, an overflow condition occurs if a positive number is subtracted from a negative number with a positive result, or, if a negative number is subtracted from a positive number with a negative result.

**N:** R7

Set if MSB of result is 1; cleared otherwise

**Z:**  $\overline{R7} \& \overline{R6} \& \overline{R5} \& \overline{R4} \& \overline{R3} \& \overline{R2} \& \overline{R1} \& \overline{R0}$

Set if result is \$00; cleared otherwise

**C:**  $\overline{A7} \& M7 \mid M7 \& R7 \mid R7 \& \overline{A7}$

Set if the unsigned value of the contents of memory is larger than the unsigned value of the accumulator; cleared otherwise

**Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
SUB #opr8i	IMM	A0	ii	2
SUB opr8a	DIR	B0	dd	3
SUB opr16a	EXT	C0	hh ll	4
SUB oprx16,X	IX2	D0	ee ff	4
SUB oprx8,X	IX1	E0	ff	3
SUB X	IX	F0		2
SUB oprx16,SP	SP2	9ED0	ee ff	5
SUB oprx8,SP	SP1	9EE0	ff	4

# SWI

## Software Interrupt

# SWI

### Operation

$PC \leftarrow (PC) + \$0001$	Increment PC to return address
Push (PCL); $SP \leftarrow (SP) - \$0001$	Push low half of return address
Push (PCH); $SP \leftarrow (SP) - \$0001$	Push high half of return address
Push (X); $SP \leftarrow (SP) - \$0001$	Push index register on stack
Push (A); $SP \leftarrow (SP) - \$0001$	Push A on stack
Push (CCR); $SP \leftarrow (SP) - \$0001$	Push CCR on stack
I bit $\leftarrow 1$	Mask further interrupts
$PCH \leftarrow (\$FFFC)$	Vector fetch (high byte)
$PCL \leftarrow (\$FFFD)$	Vector fetch (low byte)

### Description

The program counter (PC) is incremented by 1 to point at the instruction after the SWI. The PC, index register, and accumulator are pushed onto the stack. The condition code register (CCR) bits are then pushed onto the stack, with bits V, H, I, N, Z, and C going into bit positions 7 and 4–0. Bit positions 6 and 5 contain 1s. The stack pointer is decremented by 1 after each byte of data is stored on the stack. The interrupt mask bit is then set. The program counter is then loaded with the address stored in the SWI vector located at memory locations \$FFFC and \$FFFD. This instruction is not maskable by the I bit.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
—	1	1	—	1	—	—	—

I: 1  
Set

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
SWI	INH	83		9

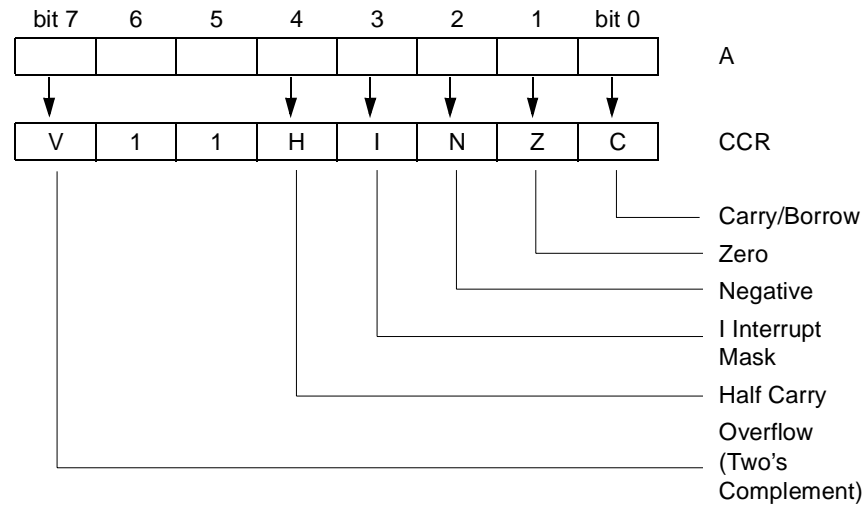
# TAP

## Transfer Accumulator to Processor Status Byte

# TAP

### Operation

$$\text{CCR} \leftarrow (\text{A})$$



### Description

Transfers the contents of A to the condition code register (CCR). The contents of A are unchanged. If this instruction causes the I bit to change from 0 to 1, a one bus cycle delay is imposed before interrupts become masked. This assures that the next instruction after a TAP instruction will always be executed even if an interrupt became pending during the TAP instruction.

### Condition Codes and Boolean Formulae

V			H	I	N	Z	C
↓	1	1	↓	↓	↓	↓	↓

Set or cleared according to the value that was in the accumulator.

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TAP	INH	84		2

# TAX

## Transfer Accumulator to X (Index Register Low)

# TAX

**Operation**

$X \leftarrow (A)$

**Description**

Loads X with the contents of the accumulator (A). The contents of A are unchanged.

**Condition Codes and Boolean Formulae**

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TAX	INH	97		1

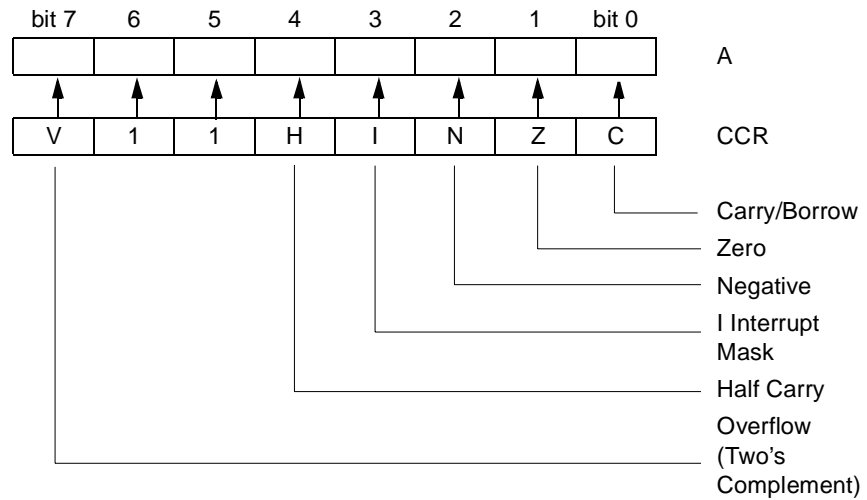
# TPA

## Transfer Processor Status Byte to Accumulator

# TPA

### Operation

$$A \leftarrow (\text{CCR})$$



### Description

Transfers the contents of the condition code register (CCR) into the accumulator (A)

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TPA	INH	85		1

# TST

## Test for Negative or Zero

# TST

### Operation

(A) – \$00  
 Or (X) – \$00  
 Or (M) – \$00

### Description

Sets the N and Z condition codes according to the contents of A, X, or M. The contents of A, X, and M are not altered.

### Condition Codes and Boolean Formulae

V		H	I	N	Z	C
0	1	1	—	—	↓	↓

V: 0  
 Cleared

N: M7  
 Set if MSB of the tested value is 1; cleared otherwise

Z:  $\overline{M7} \& \overline{M6} \& \overline{M5} \& \overline{M4} \& \overline{M3} \& \overline{M2} \& \overline{M1} \& \overline{M0}$   
 Set if A, X, or M contains \$00; cleared otherwise

### Source Forms, Addressing Modes, Machine Code, Cycles, and Access Details

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TST <i>opr8a</i>	DIR	3D	dd	3
TSTA	INH (A)	4D		1
TSTX	INH (X)	5D		1
TST <i>opr8,X</i>	IX1	6D	ff	3
TST <i>,X</i>	IX	7D		2
TST <i>opr8,SP</i>	SP1	9E6D	ff	4



# TSX

## Transfer Stack Pointer to Index Register

# TSX

### Operation

$H:X \leftarrow (SP) + \$0001$

### Description

Loads index register (H:X) with 1 plus the contents of the stack pointer (SP). The contents of SP remain unchanged. After a TSX instruction, H:X points to the last value that was stored on the stack.

### Condition Codes and Boolean Formulae

None affected

V	H	I	N	Z	C
—	1	1	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TSX	INH	95		2

# TXA

## Transfer X (Index Register Low) to Accumulator

# TXA

**Operation**  $A \leftarrow (X)$

**Description** Loads the accumulator (A) with the contents of X. The contents of X are not altered.

**Condition Codes and Boolean Formulae** None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TXA	INH	9F		1

# TXS

## Transfer Index Register to Stack Pointer

# TXS

### Operation

$SP \leftarrow (H:X) - \$0001$

### Description

Loads the stack pointer (SP) with the contents of the index register (H:X) minus 1. The contents of H:X are not altered.

### Condition Codes and Boolean Formulae

None affected

V			H	I	N	Z	C
—	1	1	—	—	—	—	—

### Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
TXS	INH	94		2

# WAIT

## Enable Interrupts; Stop Processor

# WAIT

**Operation**

I bit ← 0; inhibit CPU clocking until interrupted

**Description**

Reduces power consumption by eliminating dynamic power dissipation in some portions of the MCU. The timer, the timer prescaler, and the on-chip peripherals continue to operate (if enabled) because they are potential sources of an interrupt. Wait causes enabling of interrupts by clearing the I bit in the CCR and stops clocking of processor circuits.

Interrupts from on-chip peripherals may be enabled or disabled by local control bits prior to execution of the WAIT instruction.

When either the  $\overline{\text{RESET}}$  or  $\overline{\text{IRQ}}$  pin goes low or when any on-chip system requests interrupt service, the processor clocks are enabled, and the reset,  $\overline{\text{IRQ}}$ , or other interrupt service request is processed.

**Condition Codes and Boolean Formulae**

V			H	I	N	Z	C
—	1	1	—	0	—	—	—

I: 0  
Cleared

**Source Form, Addressing Mode, Machine Code, Cycles, and Access Detail**

Source Form	Address Mode	Machine Code		HC08 Cycles
		Opcode	Operand(s)	
WAIT	INH	8F		1

## Section 6. Instruction Set Examples

### 6.1 Contents

6.2	Introduction . . . . .	190
6.3	M68HC08 Unique Instructions . . . . .	190
6.4	Code Examples . . . . .	191
	AIS      Add Immediate Value (Signed) to Stack Pointer . . . . .	192
	AIX      Add Immediate Value (Signed) to Index Register . . . . .	194
	BGE      Branch if Greater Than or Equal To . . . . .	195
	BGT      Branch if Greater Than . . . . .	196
	BLE      Branch if Less Than or Equal To . . . . .	197
	BLT      Branch if Less Than . . . . .	198
	CBEQ     Compare and Branch if Equal . . . . .	199
	CBEQA   Compare A with Immediate . . . . .	200
	CBEQX   Compare X with Immediate . . . . .	201
	CLRH     Clear H (Index Register High) . . . . .	202
	CPHX     Compare Index Register with Memory . . . . .	203
	DAA      Decimal Adjust Accumulator . . . . .	204
	DBNZ     Decrement and Branch if Not Zero . . . . .	205
	DIV      Divide . . . . .	206
	LDHX     Load Index Register with Memory . . . . .	209
	MOV      Move . . . . .	210
	NSA      Nibble Swap Accumulator . . . . .	211
	PSHA     Push Accumulator onto Stack . . . . .	212
	PSHH     Push H (Index Register High) onto Stack . . . . .	213
	PSHX     Push X (Index Register Low) onto Stack. . . . .	214
	PULA     Pull Accumulator from Stack . . . . .	215
	PULH     Pull H (Index Register High) from Stack . . . . .	216
	PULX     Pull X (Index Register Low) from Stack. . . . .	217
	STHX     Store Index Register . . . . .	218

TAP	Transfer Accumulator to Condition Code Register . . . . .	219
TPA	Transfer Condition Code Register to Accumulator . . . . .	220
TSX	Transfer Stack Pointer to Index Register . . . . .	221
TXS	Transfer Index Register to Stack Pointer . . . . .	222

## 6.2 Introduction

The M68HC08 Family instruction set is an extension of the M68HC05 Family instruction set. This section contains code examples for the instructions unique to the M68HC08 Family.

## 6.3 M68HC08 Unique Instructions

This is a list of the instructions unique to the M68HC08 Family.

- Add Immediate Value (Signed) to Stack Pointer (AIS)
- Add Immediate Value (Signed) to Index Register (AIX)
- Branch if Greater Than or Equal To (BGE)
- Branch if Greater Than (BGT)
- Branch if Less Than or Equal To (BLE)
- Branch if Less Than (BLT)
- Compare and Branch if Equal (CBEQ)
- Compare Accumulator with Immediate, Branch if Equal (CBEQA)
- Compare Index Register Low with Immediate, Branch if Equal (CBEQX)
- Clear Index Register High (CLRH)
- Compare Index Register with Immediate Value (CPHX)
- Decimal Adjust Accumulator (DAA)
- Decrement and Branch if Not Zero (DBNZ)
- Divide (DIV)

- Load Index Register with Immediate Value (LDHX)
- Move (MOV)
- Nibble Swap Accumulator (NSA)
- Push Accumulator onto Stack (PSHA)
- Push Index Register High onto Stack (PSHH)
- Push Index Register Low onto Stack (PSHX)
- Pull Accumulator from Stack (PULA)
- Pull Index Register High from Stack (PULH)
- Pull Index Register Low from Stack (PULX)
- Store Index Register (STHX)
- Transfer Accumulator to Condition Code Register (TAP)
- Transfer Condition Code Register to Accumulator (TPA)
- Transfer Stack Pointer to Index Register (TSX)
- Transfer Index Register to Stack Pointer (TXS)

## 6.4 Code Examples

The following pages contain code examples for the instructions unique to the M68HC08 Family.

# AIS

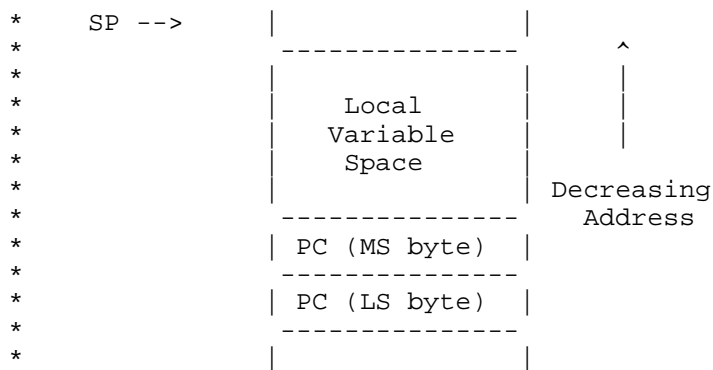
## Add Immediate Value (Signed) to Stack Pointer

# AIS

```

*
* AIS:
* 1) Creating local variable space on the stack

```



```

* NOTE: SP must always point to next unused byte,
*       therefore do not use this byte (0,SP) for storage

```

Label	Operation	Operand	Comments
SUB1	AIS	#-16	;Create 16 bytes of local space
.	.	.	.
.	.	.	.
.	.	.	.
.	AIS	#16	;Clean up stack (Note: AIS ;does not modify CCR)
.	RTS	.	;Return

\*\*\*\*\*

```

* 2) Passing parameters through the stack

```

Label	Operation	Operand	Comments
PARAM1	RMB	1	
PARAM2	RMB	1	
.	.	.	.
.	LDA	PARAM1	
.	PSHA	.	;Push dividend onto stack
.	LDA	PARAM2	
.	PSHA	.	;Push divisor onto stack
.	JSR	DIVIDE	;8/8 divide
.	PULA	.	;Get result
.	AIS	#1	;Clean up stack ;(CCR not modified)
.	BCS	ERROR	;Check result
ERROR	EQU	*	
.	.	.	.



# AIS

## Add Immediate Value (Signed) to Stack Pointer (Continued)

# AIS

```

*****
*   DIVIDE: 8/8 divide
*
*   SP ----> |           |
*             |-----|
*             |     A     |
*             |-----|
*             |     X     |           ^
*             |-----|           |
*             |     H     |           |
*             |-----|           |
*             | PC (MS byte) |           |
*             |-----|           |
*             | PC (LS byte) |           |
*             |-----|           |
*             |   Divisor   |           |
*             |-----|           |
*             |   Dividend  |           |
*             |-----|           |
*
*   Entry:   Dividend and divisor on stack at
*            SP,7 and SP,6 respectively
*   Exit:    8-bit result placed on stack at SP,6
*            A, H:X preserved
*

```

Label	Operation	Operand	Comments
DIVIDE	PSHH		;preserve H:X, A
	PSHX		
	PSHA		
	LDX	6,SP	;Divisor -> X
	CLRH		;0 -> MS dividend
	LDA	7,SP	;Dividend -> A
	DIV		
OK	STA	6,SP	;Save result
	PULA		;restore H:X, A
	PULX		
	PULH		
	RTS		

```

*
*****

```

**AIX**

**Add Immediate Value (Signed) to Index Register**

**AIX**

```

* AIX:
* 1) Find the 8-bit checksum for a 512 byte table
*
Label      Operation  Operand      Comments
TABLE      ORG          $7000
           FDB          512
           ORG          $6E00      ;ROM/EPROM address space
           LDHX         #511      ;Initialize byte count (0..511)
           CLRA         ;Clear result
ADDLOOP    ADD          TABLE,X
           AIX          #-1       ;Decrement byte counter
*
* NOTE: DECX will not carry from X through H. AIX will.
*
           CPHX         #0         ;Done?
*
* NOTE: DECX does affect the CCR. AIX does not (CPHX required).
*
           BPL          ADDLOOP    ;Loop if not complete.
*
*****
*
* 2) Round a 16-bit signed fractional number
* Radix point is assumed fixed between bits 7 and 8
*
* Entry: 16-bit fractional in fract
* Exit: Integer result after round operation in A
*
Label      Operation  Operand      Comments
FRACT      ORG          $50         ;RAM address space
           RMB          2
*
           ORG          $6E00      ;ROM/EPROM address space
           LDHX         FRACT
           AIX          #1
           AIX          #$7F       ;Round up if X >= $80 (fraction >= 0.5)
*
* NOTE: AIX operand is a signed 8-bit number. AIX #$80 would
* therefore be equivalent to AIX #-128 (signed extended
* to 16-bits). Splitting the addition into two positive
* operations is required to perform the round correctly.
*
           PSHH
           PULA
*

```

**BGE**

**Branch if Greater Than or Equal To  
(Signed Operands)**

**BGE**

```
* 8 x 8 signed multiply
*
*      Entry: Multiplier and multiplicand in VAR1 and VAR2
*      Exit  : Signed result in X:A
*
```

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
NEG_FLG	RMB	1	;Sign flag byte
VAR1	RMB	1	;Multiplier
VAR2	RMB	1	;Multiplicand
*			
*			
	ORG	\$6E00	;ROM/EPROM address space
S_MULT	CLR	NEG_FLG	;Clear negative flag
	TST	VAR1	;Check VAR1
	BGE	POS	;Continue is =>0
	INC	NEG_FLG	;Else set negative flag
	NEG	VAR1	;Make into positive number
*			
POS	TST	VAR2	;Check VAR2
	BGE	POS2	;Continue is =>0
	INC	NEG_FLG	;Else toggle negative flag
	NEG	VAR2	;Make into positive number
*			
POS2	LDA	VAR2	;Load VAR1
	LDX	VAR1	;Load VAR2
	MUL		;Unsigned VAR1 x VAR2 -> X:A
	BRCLR	0,NEG_FLG,EXIT	;Quit if operands both ;positive or both neg.
	COMA		;Else one's complement A and X
	COMX		
	ADD	#1	;Add 1 for 2's complement ;(LS byte)
	PSHA		;Save LS byte of result
	TXA		;Transfer unsigned MS byte of ;result
	ADC	#0	;Add carry result to complete ;2's complement
	TAX		;Return to X
	PULA		;Restore LS byte of result
EXIT	RTS		;Return
*			

## BGT

### Branch if Greater Than (Signed Operands)

## BGT

```
* BGT:
* Read an 8-bit A/D register, sign it and test for valid range
*
*      Entry: New reading in AD_RES
*      Exit  : Signed result in A. ERR_FLG set if out of range.
*
*
```

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
ERR_FLG	RMB	1	;Out of range flag
AD_RES	RMB	1	;A/D result register
*			
*			
	ORG	\$6E00	;ROM/EPROM address space
	BCLR	0,ERR_FLG	
	LDA	AD_RES	;Get latest reading (0 thru 256)
	EOR	#\$80	;Sign it (-128 thru 128)
	CMP	#\$73	;If greater than upper limit,
	BGT	OUT	; branch to error flag set
	CMP	#\$8D	;If greater than lower limit
			;\$8D = -\$73)
	BGT	IN	; branch to exit
OUT	BSET	0,ERR_FLG	;Set error flag
IN	RTS		;Return
*			

**BLE**

**Branch if Less Than or Equal To  
(Signed Operands)**

**BLE**

```
* Find the most negative of two 16-bit signed integers
*
*      Entry: Signed 16-bit integers in VAL1 and VAL2
*      Exit  : Most negative integer in H:X
*
```

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
VAL1	RMB	2	;16-bit signed integer
VAL2	RMB	2	;16-bit signed integer
*			
*			
	ORG	\$6E00	;ROM/EPROM address space
	LDHX	VAL1	
	CPHX	VAL2	
	BLE	EXIT1	;If VAL1 =< VAL2, exit
	LDHX	VAL2	; else load VAL2 into H:X
EXIT1	EQU	*	
*			

## BLT

### Branch if Less Than (Signed Operands)

## BLT

\* Compare 8-bit signed integers in A and X and place the  
\* most negative in A.

\*

\* Entry: Signed 8-bit integers in A and X

\* Exit : Most negative integer in A. X preserved.

\*

\*

Label	Operation	Operand	Comments
	ORG	\$6E00	;ROM/EPROM address space
	PSHX		;Move X onto stack
	CMP	1,SP	;Compare it with A
	BLT	EXIT2	;If A =< stacked X, quit
	TXA		;else move X to A
EXIT2	PULX		;Clean up stack

\*

# CBEQ

## Compare and Branch if Equal

# CBEQ

\* Skip spaces in a string of ASCII characters. String must  
\* contain at least one non-space character.

\*  
\*       Entry: H:X points to start of string  
\*       Exit : H:X points to first non-space character in  
\*       string

Label	Operation	Operand	Comments
	LDA	#\$20	;Load space character
SKIP	CBEQ	X+,SKIP	;Increment through string until ;non-space character found.

\*  
\* NOTE: X post increment will occur irrespective of whether  
\* branch is taken. In this example, H:X will point to the  
\* non-space character+1 immediately following the CBEQ  
\* instruction.

Label	Operation	Operand	Comments
	AIX	#-1	;Adjust pointer to point to 1st ;non-space char.
	RTS		;Return

\*

## CBEQA

### Compare A with Immediate (Branch if Equal)

## CBEQA

\* Look for an End-of-Transmission (EOT) character from a  
\* serial peripheral. Exit if true, otherwise process data  
\* received.

\*

Label	Operation	Operand	Comments
EOT	EQU	\$04	

\*

DATA_RX	EQU	1	
---------	-----	---	--

\*

	LDA	DATA_RX	;get receive data
	CBEQA	#EOT,EXIT3	;check for EOT

\*

\* NOTE: CBEQ, CBEQA, CBEQX instructions do NOT modify the  
\* CCR. In this example, Z flag will remain in the state the  
\* LDA instruction left it in.

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

\*

	Process data
--	-----------------

EXIT3	RTS
-------	-----



# CBEQX

## Compare X with Immediate (Branch if Equal)

# CBEQX

\* Keyboard wake-up interrupt service routine. Return to sleep  
\* (WAIT mode) unless "ON" key has been depressed.  
\*

Label	Operation	Operand	Comments
ON_KEY	EQU	\$02	
*			
SLEEP	WAIT		
	BSR	DELAY	;Debounce delay routine
	LDX	PORTA	;Read keys
	CBEQX	#ON_KEY,WAKEUP	;Wake up if "ON" pressed,
	BRA	SLEEP	;otherwise return to sleep
*			
WAKEUP	EQU	*	;Start of main code
*			

## CLR H

### Clear H (Index Register High)

## CLR H

\* Clear H:X register

\*

Label	Operation	Operand	Comments
-------	-----------	---------	----------

	CLR X		
--	-------	--	--

	CLR H		
--	-------	--	--

\*

\* NOTE: This sequence takes 2 cycles and uses 2 bytes

\* LDHX #0 takes 3 cycles and uses 3 bytes.

\*

# CPHX

## Compare Index Register with Memory

# CPHX

\* Stack pointer overflow test. Branch to a fatal error  
\* handler if overflow detected.

Label	Operation	Operand	Comments
STACK	EQU	\$1000	;Stack start address (empty)
SIZE	EQU	\$100	;Maximum stack size
*			
	PSHH		;Save H:X (assuming stack is OK!)
	PSHX		
	TSX		;Move SP+1 to H:X
	CPHX	#STACK-SIZE	;Compare against stack lowest ;address
	BLO	FATAL	;Branch out if lower ; otherwise continue executing ;main code
*			
	PULX		;Restore H:X
	PULH		
*			
*			
*			
*			
*			
FATAL	EQU	*	;FATAL ERROR HANDLER
*			

## DAA

## Decimal Adjust Accumulator

## DAA

\* Add 2 BCD 8-bit numbers (e.g. 78 + 49 = 127)

\*

Label	Operation	Operand	Comments
VALUE1	FCB	\$78	
VALUE2	FCB	\$49	
*			
	LDA	VALUE1	;A = \$78
	ADD	VALUE2	;A = \$78+\$49 = \$C1; C=0, H=1
	DAA		;Add \$66; A = \$27; C=1 {=127 BCD}
*			

# DBNZ

## Decrement and Branch if Not Zero

# DBNZ

```

* Delay routine:
* Delay = N x (153.6+0.36)uS for 60nS CPU clock
* For example, delay=10mS for N=$41 and 60nS CPU clock
*
*      Entry: COUNT = 0
*      Exit:  COUNT = 0; A = N
*
Label    Operation    Operand    Comments
N          EQU          $41          ;Loop constant for 10mS delay
*
          ORG          $50          ;RAM address space
COUNT    RMB          1          ;Loop counter
*
          ORG          $6E00       ;ROM/EPROM address space
DELAY     LDA          #N          ;Set delay constant
LOOPY    DBNZ         COUNT,LOOPY ;Inner loop (5x256 cycles)
          DBNZA        LOOPY       ;Outer loop (3 cycles)
*

```

## DIV

## Divide

## DIV

\* 1) 8/8 integer divide > 8-bit integer quotient  
 \* Performs an unsigned integer divide of an 8-bit dividend  
 \* in A by an 8-bit divisor in X. H must be cleared. The  
 \* quotient is placed into A and the remainder in H.  
 \*

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
DIVID1	RMB	1	;storage for dividend
DIVISOR1	RMB	1	;storage for divisor
QUOTIENT1	RMB	1	;storage for quotient
			*
	ORG	\$6E00	;ROM/EPROM address space
	LDA	DIVID1	;Load dividend
	CLR H		;Clear MS byte of dividend
	LDX	DIVISOR1	;Load divisor
	DIV		;8/8 divide
	STA	QUOTIENT1	;Store result; remainder in H

\*

\*

\* 2) 8/8 integer divide > 8-bit integer and 8-bit fractional  
 \* quotient. Performs an unsigned integer divide of an 8-bit  
 \* dividend in A by an 8-bit divisor in X. H must be  
 \* cleared. The quotient is placed into A and the remainder  
 \* in H. The remainder may be further resolved by executing  
 \* additional DIV instructions as shown below. The radix point  
 \* of the quotient will be between bits 7 and 8.  
 \*

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
DIVID2	RMB	1	;storage for dividend
DIVISOR2	RMB	1	;storage for divisor
QUOTIENT2	RMB	2	;storage for quotient
			*
	ORG	\$6E00	;ROM/EPROM address space
	LDA	DIVID2	;Load dividend
	CLR H		;Clear MS byte of dividend
	LDX	DIVISOR2	;Load divisor
	DIV		;8/8 divide
	STA	QUOTIENT2	;Store result; remainder in H
	CLRA		
	DIV		;Resolve remainder
	STA	QUOTIENT2+1	

\*

\*

**DIV**

**Divide  
(Continued)**

**DIV**

\* 3) 8/8 fractional divide > 16-bit fractional quotient  
 \* Performs an unsigned fractional divide of an 8-bit dividend  
 \* in H by the 8-bit divisor in X. A must be cleared. The  
 \* quotient is placed into A and the remainder in H. The  
 \* remainder may be further resolved by executing additional  
 \* DIV instructions as shown below.  
 \* The radix point is assumed to be in the same place for both  
 \* the dividend and the divisor. The radix point is to the  
 \* left of the MS bit of the quotient. An overflow will occur  
 \* when the dividend is greater than or equal to the divisor.  
 \* The quotient is an unsigned binary weighted fraction with  
 \* a range of \$00 to \$FF (0.9961).  
 \*

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
DIVID3	RMB	1	;storage for dividend
DIVISOR3	RMB	1	;storage for divisor
QUOTIENT3	RMB	2	;storage for quotient
*			
	ORG	\$6E00	;ROM/EPROM address space
	LDHX	DIVID3	;Load dividend into H (and ;divisor into X)
	CLRA		;Clear LS byte of dividend
	DIV		;8/8 divide
	STA	QUOTIENT3	;Store result; remainder in H
	CLRA		
	DIV		;Resolve remainder
	STA	QUOTIENT3+1	

\*  
\*

\* 4) Unbounded 16/8 integer divide  
 \* This algorithm performs the equivalent of long division.  
 \* The initial divide is an 8/8 (no overflow possible).  
 \* Subsequent divide are 16/8 using the remainder from the  
 \* previous divide operation (no overflow possible).  
 \* The DIV instruction does not corrupt the divisor and leaves  
 \* the remainder in H, the optimal position for successive  
 \* divide operations. The algorithm may be extended to any  
 \* precision of dividend by performing additional divides.  
 \* This, of course, includes resolving the remainder of a  
 \* divide operation into a fractional result as shown below.  
 \*

## DIV

### Divide (Concluded)

## DIV

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
DIVIDEND4	RMB	2	;storage for dividend
DIVISOR4	RMB	1	;storage for divisor
QUOTIENT4	RMB	3	;storage for quotient
*			
*			
	ORG	\$6E00	;ROM/EPROM address space
	LDA	DIVIDEND4	;Load MS byte of dividend into ;LS dividend reg.
	CLRH		;Clear H (MS dividend register)
	LDX	DIVISOR4	;Load divisor
	DIV		;8/8 integer divide [A/X -> A; r->H]
	STA	QUOTIENT4	;Store result (MS result of ;complete operation)
*			;Remainder in H (MS dividend ;register)
	LDA	DIVIDEND4+1	;Load LS byte of dividend into ;LS dividend reg.
	DIV		;16/8 integer divide ;[H:A/X -> A; r->H]
	STA	QUOTIENT4+1	;Store result (LS result of ;complete operation)
	CLRA		;Clear LS dividend (prepare for ;fract. divide)
	DIV		;Resolve remainder
	STA	QUOTIENT4+2	;Store fractional result.

\*

\*

\* 5) Bounded 16/8 integer divide

\* Although the DIV instruction will perform a 16/8 integer  
\* divide, it can only generate an 8-bit quotient. Quotient  
\* overflows are therefore possible unless the user knows the  
\* bounds of the dividend and divisor in advance.

\*

Label	Operation	Operand	Comments
	ORG	\$50	;RAM address space
DIVID5	RMB	2	;storage for dividend
DIVISOR5	RMB	1	;storage for divisor
QUOTIENT5	RMB	1	;storage for quotient
*			
	ORG	\$6E00	;ROM/EPROM address space
	LDHX	DIVID5	;Load dividend into H:X
	TXA		;Move X to A
	LDX	DIVISOR5	;Load divisor into X
	DIV		;16/8 integer divide
	BCS	ERROR5	;Overflow?
	STA	QUOTIENT5	;Store result
ERROR5	EQU	*	



# LDHX

## Load Index Register with Memory

# LDHX

\* Clear RAM block of memory

\*

Label	Operation	Operand	Comments
RAM	EQU	\$0050	;Start of RAM
SIZE1	EQU	\$400	;Length of RAM array
*			
	LDHX	#RAM	;Load RAM pointer
LOOP	CLR	,X	;Clear byte
	AIX	#1	;Bump pointer
	CPHX	#RAM+SIZE1	;Done?
	BLO	loop	;Loop if not

## MOV

## Move

## MOV

\* 1) Initialize Port A and Port B data registers in page 0.

```
*
Label      Operation      Operand      Comments
PORTA      EQU                $0000        ;port a data register
PORTB      EQU                $0001        ;port b data register
*
          MOV                #$AA,PORTA    ;store $AA to port a
          MOV                #$55,PORTB    ;store $55 to port b
*
*
*
```

\* 2) Move REG1 to REG2 if REG1 positive; clear REG2\*

```
Label      Operation      Operand      Comments
REG1       EQU                $0010
REG2       EQU                $0011
*
          MOV                REG1,REG2
          BMI                NEG
          CLR                REG2
*
*
*
```

```
NEG        EQU                *
```

\* 3) Move data to a page 0 location from a table anywhere in memory

```
Label      Operation      Operand      Comments
SPIOUT     EQU                $0012
*
          ORG                $50          ;RAM address space
TABLE_PTR  RMB                2          ;storage for table pointer
*
          ORG                $6E00       ;ROM/EPROM address space
          LDHX               TABLE_PTR  ;Restore table pointer
          MOV                X+,SPIOUT   ;Move data
*
*
*
```

```
* NOTE: X+ is a 16-bit increment of the H:X register
* NOTE: The increment occurs after the move operation is
* completed
*
*
*
          STHX               TABLE_PTR  ;Save modified pointer
*
*
```

# NSA

## Nibble Swap Accumulator

# NSA

\* NSA:  
 \* Compress 2 bytes, each containing one BCD nibble, into 1  
 \* byte. Each byte contains the BCD nibble in bits 0-3. Bits  
 \* 4-7 are clear.

Label	Operation	Operand	Comments
BCD1	RMB	1	
BCD2	RMB	1	
*			
	LDA	BCD1	;Read first BCD byte
	NSA		;Swap LS and MS nibbles
	ADD	BCD2	;Add second BCD byte
*			

## PSHA

### Push Accumulator onto Stack

## PSHA

```

* PSHA:
* Jump table index calculation.
* Jump to a specific code routine based on a number held in A
*
*      Entry : A = jump selection number, 0-3
*
Label    Operation    Operand    Comments
PSHA      PSHA              ;Save selection number
          LSLA              ;Multiply by 2
          ADD               1,SP    ;Add stacked number;
          ;A now = A x 3
          TAX               ;Move to index reg
          CLRH              ;and clear MS byte
          PULA              ;Clean up stack
          JMP               TABLE1,X ;Jump into table....
TABLE1    JMP               PROG_0
          JMP               PROG_1
          JMP               PROG_2
          JMP               PROG_3
*
PROG_0    EQU               *
PROG_1    EQU               *
PROG_2    EQU               *
PROG_3    EQU               *
*

```

# PSHH

## Push H (Index Register High) onto Stack

# PSHH

```
* PSHH:
* 1) Save contents of H register at the start of an interrupt
* service routine
*
```

Label	Operation	Operand	Comments
SCI_INT	PSHH		;Save H (all other registers ;already stacked)
*			
*			
*			
*			
*			
	PULH		;Restore H
	RTI		;Unstack all other registers; ;return to main

```
*
*
* 2) Effective address calculation
*
```

```
* Entry : H:X=pointer, A=offset
* Exit  : H:X = A + H:X (A = H)
*
```

Label	Operation	Operand	Comments
	PSHX		;Push X then H onto stack
	PSHH		
	ADD	2,SP	;Add stacked X to A
	TAX		;Move result into X
	PULA		;Pull stacked H into A
	ADC	#0	;Take care of any carry
	PSHA		;Push modified H onto stack
	PULH		;Pull back into H
	AIS	#1	;Clean up stack
*			

## PSHX

### Push X (Index Register Low) onto Stack

## PSHX

\* PSHX:  
 \* 1) Implement the transfer of the X register to the H  
 \* register  
 \*

Label	Operation	Operand	Comments
	PSHX		;Move X onto the stack
	PULH		;Return back to H

\*  
 \* 2) Implement the exchange of the X register and A  
 \*

Label	Operation	Operand	Comments
	PSHX		;Move X onto the stack
	TAX		;Move A into X
	PULA		;Restore X into A

\*

# PULA

## Pull Accumulator from Stack

# PULA

\* Implement the transfer of the H register to A  
\*

Label	Operation	Operand	Comments
	PSHH		;Move H onto stack
	PULA		;Return back to A

## PULH

### Pull H (Index Register High) from Stack

## PULH

\* Implement the exchange of the H register and A  
\*

Label	Operation	Operand	Comments
	PSHA		;Move A onto the stack
	PSHH		;Move H onto the stack
	PULA		;Pull H into A
	PULH		;Pull A into H



# PULX

## Pull X (Index Register Low) from Stack

# PULX

\* Implement the exchange of the X register and A  
\*

Label	Operation	Operand	Comments
	PSHA		;Move A onto the stack
	TXA		;Move X into A
	PULX		;Restore A into X

## STHX

### Store Index Register

## STHX

```

* Effective address calculation
*
*      Entry : H:X=pointer, A=offset
*      Exit  : H:X = A + H:X
*
Label      Operation      Operand      Comments
      ORG          $50          ;RAM address space
TEMP      RMB          2
*
      ORG          $6E00       ;ROM/EPROM address space
      STHX        TEMP        ;Save H:X
      ADD         TEMP+1      ;Add saved X to A
      TAX
      LDA         TEMP        ;Load saved X into A
      ADC         #0          ;Take care of any carry
      PSHA
      PULH        ;Pull back into H
*

```

# TAP

## Transfer Accumulator to Condition Code Register

# TAP

\*  
\* NOTE: The TAP instruction was added to improve testability of  
\* the CPU08, and so few practical applications of the  
\* instruction exist.  
\*

## TPA

### Transfer Condition Code Register to Accumulator

## TPA

\* Implement branch if 2's complement signed overflow bit  
 \* (V-bit) is set

\*

Label	Operation	Operand	Comments
-------	-----------	---------	----------

	TPA		
--	-----	--	--

\*

\* NOTE: Transferring the CCR to A does not modify the CCR.

\*

	TSTA		
--	------	--	--

	BMI	V_SET	
--	-----	-------	--

\*

V_SET	EQU	*	
-------	-----	---	--

\*

# TSX

## Transfer Stack Pointer to Index Register

# TSX

```

* TSX:
* Create a stack frame pointer. H:X points to the stack frame
* irrespective of stack depth. Useful for handling nested
* subroutine calls (e.g. recursive routines) which reference
* the stack frame data.
*
Label      Operation      Operand      Comments
LOCAL      EQU              $20
*
*          AIS              #LOCAL      ;Create local variable space in
*                                     ;stack frame
*          TSX              ;SP +1 > H:X
*
* NOTE: TSX transfers SP+1 to allow the H:X register to point
* to the first used stack byte (SP always points to the next
* available stack byte). The SP itself is not modified.
*
*          |
*          |
*          |
*          LDA              0,X          ;Load the 1st byte in local space
*          |
*          |
*          |
*          |

```

## TXS

### Transfer Index Register to Stack Pointer

## TXS

\* Initialize the SP to a value other than the reset state

\*

Label	Operation	Operand	Comments
STACK1	EQU	\$0FFF	
*			
	LDHX	#STACK1+1	;\$1000 > H:X
	TXS		;\$0FFF > SP

\*

\* NOTE: TXS subtracts 1 from the value in H:X before it

\* transfers to SP.

## Glossary

**\$xxxx** — The digits following the “\$” are in hexadecimal format.

**#xxxx** — The digits following the “#” indicate an immediate operand.

**A** — Accumulator. See “accumulator.”

**accumulator (A)** — An 8-bit general-purpose register in the CPU08. The CPU08 uses the accumulator to hold operands and results of arithmetic and non-arithmetic operations.

**address bus** — The set of conductors used to select a specific memory location so that the CPU can write information into the memory location or read its contents.

**addressing mode** — The way that the CPU obtains (addresses) the information needed to complete an instruction. The M68HC08 CPU has 16 addressing modes.

**algorithm** — A set of specific procedures by which a solution is obtained in a finite number of steps, often used in numerical calculation.

**ALU** — Arithmetic logic unit. See “arithmetic logic unit.”

**arithmetic logic unit (ALU)** — The portion of the CPU of a computer where mathematical and logical operations take place. Other circuitry decodes each instruction and configures the ALU to perform the necessary arithmetic or logical operations at each step of an instruction.

**assembly language** — A method used by programmers for representing machine instructions (binary data) in a more convenient form. Each machine instruction is given a simple, short name, called a mnemonic (or memory aid), which has a

one-to-one correspondence with the machine instruction. The mnemonics are translated into an object code program that a microcontroller can use.

**ASCII** — American Standard Code for Information Interchange. A widely accepted correlation between alphabetic and numeric characters and specific 7-bit binary numbers.

**asynchronous** — Refers to circuitry and operations without common clock signals.

**BCD** — Binary-coded decimal. See “binary-coded decimal.”

**binary** — The binary number system using 2 as its base and using only the digits 0 and 1. Binary is the numbering system used by computers because any quantity can be represented by a series of 1s and 0s. Electrically, these 1s and 0s are represented by voltage levels of approximately  $V_{DD}$  (input) and  $V_{SS}$  (ground), respectively.

**binary-coded decimal (BCD)** — A notation that uses binary values to represent decimal quantities. Each BCD digit uses four binary bits. Six of the possible 16 binary combinations are considered illegal.

**bit** — A single binary digit. A bit can hold a single value of 0 or 1.

**Boolean** — A mathematical system of representing logic through a series of algebraic equations that can only be true or false, using operators such as AND, OR, and NOT.

**branch instructions** — Computer instructions that cause the CPU to continue processing at a memory location other than the next sequential address. Most branch instructions are conditional. That is, the CPU continues to the next sequential address (no branch) if a condition is false, or continue to some other address (branch) if the condition is true.

**bus** — A collection of logic lines (conductor paths) used to transfer data.

**byte** — A set of exactly eight binary bits.



**C** — Abbreviation for carry/borrow in the condition code register of the CPU08. The CPU08 sets the carry/borrow flag when an addition operation produces a carry out of bit 7 of the accumulator or when a subtraction operation requires a borrow. Some logical operations and data manipulation instructions also clear or set the C flag (as in bit test and branch instructions and shifts and rotates).

**CCR** — Abbreviation for condition code register in the CPU08. See “condition code register.”

**central processor unit (CPU)** — The primary functioning unit of any computer system. The CPU controls the execution of instructions.

**checksum** — A value that results from adding a series of binary numbers. When exchanging information between computers, a checksum gives an indication about the integrity of the data transfer. If values were transferred incorrectly, it is unlikely that the checksum would match the value that was expected.

**clear** — To establish logic 0 state on a bit or bits; the opposite of “set.”

**clock** — A square wave signal used to sequence events in a computer.

**condition code register (CCR)** — An 8-bit register in the CPU08 that contains the interrupt mask bit and five bits (flags) that indicate the results of the instruction just executed.

**control unit** — One of two major units of the CPU. The control unit contains logic functions that synchronize the machine and direct various operations. The control unit decodes instructions and generates the internal control signals that perform the requested operations. The outputs of the control unit drive the execution unit, which contains the arithmetic logic unit (ALU), CPU registers, and bus interface.

**CPU** — Central processor unit. See “central processor unit.”

**CPU08** — The central processor unit of the M68HC08 Family.

**CPU cycles** — A CPU clock cycle is one period of the internal bus-rate clock, normally derived by dividing a crystal oscillator source by two or more so the high and low times are equal. The length of time required to execute an instruction is measured in CPU clock cycles.

**CPU registers** — Memory locations that are wired directly into the CPU logic instead of being part of the addressable memory map. The CPU always has direct access to the information in these registers. The CPU registers in an M68HC08 are:

- A (8-bit accumulator)
- H:X (16-bit accumulator)
- SP (16-bit stack pointer)
- PC (16-bit program counter)
- CCR (condition code register containing the V, H, I, N, Z, and C bits)

**cycles** — See “CPU cycles.”

**data bus** — A set of conductors used to convey binary information from a CPU to a memory location or from a memory location to a CPU.

**decimal** — Base 10 numbering system that uses the digits zero through nine.

**direct address** — Any address within the first 256 addresses of memory (\$0000–\$00FF). The high-order byte of these addresses is always \$00. Special instructions allow these addresses to be accessed using only the low-order byte of their address. These instructions automatically fill in the assumed \$00 value for the high-order byte of the address.

**direct addressing mode** — Direct addressing mode uses a program-supplied value for the low-order byte of the address of an operand. The high-order byte of the operand address is assumed to be \$00 and so it does not have to be explicitly specified. Most direct addressing mode instructions can access any of the first 256 memory addresses.

**direct memory access (DMA)** — One of a number of modules that handle a variety of control functions in the modular M68HC08 Family. The DMA can perform interrupt-driven and software-initiated data transfers between any two CPU-addressable locations. Each DMA channel can independently transfer data between any addresses in the memory map. DMA transfers reduce CPU overhead required for data movement interrupts.

**direct page** — The first 256 bytes of memory (\$0000–\$00FF); also called page 0.

**DMA** — Direct memory access. See “direct memory access.”

**EA** — Effective address. See “effective address.”

**effective address (EA)** — The address where an instruction operand is located. The addressing mode of an instruction determines how the CPU calculates the effective address of the operand.

**EPROM** — Erasable, programmable, read-only memory. A non-volatile type of memory that can be erased by exposure to an ultraviolet light source.

**EU** — Execution unit. See “execution unit.”

**execution unit (EU)** — One of the two major units of the CPU containing the arithmetic logic unit (ALU), CPU registers, and bus interface. The outputs of the control unit drive the execution unit.

**extended addressing mode** — In this addressing mode, the high-order byte of the address of the operand is located in the next memory location after the opcode. The low-order byte of the operand address is located in the second memory location after the opcode. Extended addressing mode instructions can access any address in a 64-Kbyte memory map.

**H** — Abbreviation for the upper byte of the 16-bit index register (H:X) in the CPU08.

**H** — Abbreviation for “half-carry” in the condition code register of the CPU08. This bit indicates a carry from the low-order four bits of the accumulator value to the high-order four bits. The half-carry bit is required for binary-coded decimal arithmetic operations. The decimal adjust accumulator (DAA) instruction uses the state of the H and C flags to determine the appropriate correction factor.

**hexadecimal** — Base 16 numbering system that uses the digits 0 through 9 and the letters A through F. One hexadecimal digit can exactly represent a 4-bit binary value. Hexadecimal is used by people to represent binary values because a 2-digit number is easier to use than the equivalent 8-digit number.

**high order** — The leftmost digit(s) of a number; the opposite of low order.

**H:X** — Abbreviation for the 16-bit index register in the CPU08. The upper byte of H:X is called H. The lower byte is called X. In the indexed addressing modes, the CPU uses the contents of H:X to determine the effective address of the operand. H:X can also serve as a temporary data storage location.

**I** — Abbreviation for “interrupt mask bit” in the condition code register of the CPU08. When I is set, all interrupts are disabled. When I is cleared, interrupts are enabled.

**immediate addressing mode** — In immediate addressing mode, the operand is located in the next memory location(s) after the opcode. The immediate value is one or two bytes, depending on the size of the register involved in the instruction.

**index register (H:X)** — A 16-bit register in the CPU08. The upper byte of H:X is called H. The lower byte is called X. In the indexed addressing modes, the CPU uses the contents of H:X to determine the effective address of the operand. H:X can also serve as a temporary data storage location.

**indexed addressing mode** — Indexed addressing mode instructions access data with variable addresses. The effective address of the operand is determined by the current value of the H:X register added to a 0-, 8-, or 16-bit value (offset) in the

instruction. There are separate opcodes for 0-, 8-, and 16-bit variations of indexed mode instructions, and so the CPU knows how many additional memory locations to read after the opcode.

**indexed, post increment addressing mode** — In this addressing mode, the effective address of the operand is determined by the current value of the index register, added to a 0- or 8-bit value (offset) in the instruction, after which the index register is incremented. Operands with variable addresses can be addressed with the 8-bit offset instruction.

**inherent addressing mode** — The inherent addressing mode has no operand because the opcode contains all the information necessary to carry out the instruction. Most inherent instructions are one byte long.

**input/output (I/O)** — Input/output interfaces between a computer system and the external world. A CPU reads an input to sense the level of an external signal and writes to an output to change the level on an external signal.

**instructions** — Instructions are operations that a CPU can perform. Instructions are expressed by programmers as assembly language mnemonics. A CPU interprets an opcode and its associated operand(s) and instruction(s).

**instruction set** — The instruction set of a CPU is the set of all operations that the CPU can perform. An instruction set is often represented with a set of shorthand mnemonics, such as LDA, meaning “load accumulator (A).” Another representation of an instruction set is with a set of opcodes that are recognized by the CPU.

**interrupt** — Interrupts provide a means to temporarily suspend normal program execution so that the CPU is freed to service sets of instructions in response to requests (interrupts) from peripheral devices. Normal program execution can be resumed later from its original point of departure. The CPU08 can process up to 128 separate interrupt sources, including a software interrupt (SWI).

**I/O** — Input/output. See “input/output.”

**IRQ** — Interrupt request. The overline indicates an active-low signal.

**least significant bit (LSB)** — The rightmost digit of a binary value; the opposite of most significant bit (MSB).

**logic 1** — A voltage level approximately equal to the input power voltage ( $V_{DD}$ ).

**logic 0** — A voltage level approximately equal to the ground voltage ( $V_{SS}$ ).

**low order** — The rightmost digit(s) of a number; the opposite of high order.

**LS** — Least significant.

**LSB** — Least significant bit. See “least significant bit.”

**M68HC08** — The Motorola Family of 8-bit MCUs.

**machine codes** — The binary codes processed by the CPU as instructions. Machine code includes both opcodes and operand data.

**MCU** — Microcontroller unit. See “microcontroller unit.”

**memory location** — In the M68HC08, each memory location holds one byte of data and has a unique address. To store information into a memory location, the CPU places the address of the location on the address bus, the data information on the data bus, and asserts the write signal. To read information from a memory location, the CPU places the address of the location on the address bus and asserts the read signal. In response to the read signal, the selected memory location places its data onto the data bus.

**memory map** — A pictorial representation of all memory locations in a computer system.

**memory-to-memory addressing mode** — In this addressing mode, the accumulator has been eliminated from the data transfer process, thereby reducing execution cycles. This addressing mode, therefore, provides rapid data transfers because it does

not use the accumulator and associated load and store instructions. There are four memory-to-memory addressing mode instructions. Depending on the instruction, operands are found in the byte following the opcode, in a direct page location addressed by the byte immediately following the opcode, or in a location addressed by the index register.

**microcontroller unit (MCU)** — A complete computer system, including a CPU, memory, a clock oscillator, and input/output (I/O) on a single integrated circuit.

**mnemonic** — Three to five letters that represent a computer operation. For example, the mnemonic form of the “load accumulator” instruction is LDA.

**most significant bit (MSB)** — The leftmost digit of a binary value; the opposite of least significant bit (LSB).

**MS** — Abbreviation for “most significant.”

**MSB** — Most significant bit. See “most significant bit.”

**N** — Abbreviation for “negative,” a bit in the condition code register of the CPU08. The CPU sets the negative flag when an arithmetic operation, logical operation, or data manipulation produces a negative result.

**nibble** — Half a byte; four bits.

**object code** — The output from an assembler or compiler that is itself executable machine code or is suitable for processing to produce executable machine code.

**one** — A logic high level, a voltage level approximately equal to the input power voltage ( $V_{DD}$ ).

**one’s complement** — An infrequently used form of signed binary numbers. Negative numbers are simply the complement of their positive counterparts. One’s complement is the result of a bit-by-bit complement of a binary word: All 1s are changed to 0s and all 0s changed to 1s. One’s complement is two’s complement without the increment.

**opcode** — A binary code that instructs the CPU to do a specific operation in a specific way.

**operand** — The fundamental quantity on which a mathematical operation is performed. Usually a statement consists of an operator and an operand. The operator may indicate an add instruction; the operand therefore will indicate what is to be added.

**oscillator** — A circuit that produces a constant frequency square wave that is used by the computer as a timing and sequencing reference.

**page 0** — The first 256 bytes of memory (\$0000–\$00FF). Also called direct page.

**PC** — Program counter. See “program counter.”

**pointer** — Pointer register. An index register is sometimes called a pointer register because its contents are used in the calculation of the address of an operand, and therefore “points” to the operand.

**program** — A set of computer instructions that cause a computer to perform a desired operation or operations.

**programming model** — The registers of a particular CPU.

**program counter (PC)** — A 16-bit register in the CPU08. The PC register holds the address of the next instruction or operand that the CPU will use.

**pull** — The act of reading a value from the stack. In the M68HC08, a value is pulled by the following sequence of operations. First, the stack pointer register is incremented so that it points to the last value saved on the stack. Next, the value at the address contained in the stack pointer register is read into the CPU.

**push** — The act of storing a value at the address contained in the stack pointer register and then decrementing the stack pointer so that it points to the next available stack location.



**random access memory (RAM)** — A type of memory that can be read or written by the CPU. The contents of a RAM memory location remain valid until the CPU writes a different value or until power is turned off.

**RAM** — Random access memory. See “random-access memory.”

**read** — To transfer the contents of a memory location to the CPU.

**read-only memory** — A type of memory that can be read but cannot be changed (written) by the CPU. The contents of ROM must be specified before manufacturing the MCU.

**registers** — Memory locations wired directly into the CPU logic instead of being part of the addressable memory map. The CPU always has direct access to the information in these registers. The CPU registers in an M68HC08 are:

- A (8-bit accumulator)
- (H:X) (16-bit index register)
- SP (16-bit stack pointer)
- PC (16-bit program counter)
- CCR (condition code register containing the V, H, I, N, Z, and C bits)

Memory locations that hold status and control information for on-chip peripherals are called input/output (I/O) and control registers.

**relative addressing mode** — Relative addressing mode is used to calculate the destination address for branch instructions. If the branch condition is true, the signed 8-bit value after the opcode is added to the current value of the program counter to get the address where the CPU will fetch the next instruction. If the branch condition is false, the effective address is the content of the program counter.

**reset** — Reset is used to force a computer system to a known starting point and to force on-chip peripherals to known starting conditions.

**ROM** — Read-only memory. See “read-only memory.”

**set** — To establish a logic 1 state on a bit or bits; the opposite of “clear.”

**signed** — A form of binary number representation accommodating both positive and negative numbers. The most significant bit is used to indicate whether the number is positive or negative, normally zero for positive and one for negative, and the other seven bits indicate the magnitude.

**SIM** — System integration module. See “system integration module.”

**SP** — Stack pointer. See “stack pointer.”

**stack** — A mechanism for temporarily saving CPU register values during interrupts and subroutines. The CPU maintains this structure with the stack pointer (SP) register, which contains the address of the next available (empty) storage location on the stack. When a subroutine is called, the CPU pushes (stores) the low-order and high-order bytes of the return address on the stack before starting the subroutine instructions. When the subroutine is done, a return from subroutine (RTS) instruction causes the CPU to recover the return address from the stack and continue processing where it left off before the subroutine. Interrupts work in the same way except that all CPU registers are saved on the stack instead of just the program counter.

**stack pointer (SP)** — A 16-bit register in the CPU08 containing the address of the next available (empty) storage on the stack.

**stack pointer addressing mode** — Stack pointer (SP) addressing mode instructions operate like indexed addressing mode instructions except that the offset is added to the stack pointer instead of the index register (H:X). The effective address of the operand is formed by adding the unsigned byte(s) in the stack pointer to the unsigned byte(s) following the opcode.

**subroutine** — A sequence of instructions to be used more than once in the course of a program. The last instruction in a subroutine is a return-from-subroutine (RTS) instruction. At each place in the main program where the subroutine instructions are needed, a jump or branch to subroutine (JSR or BSR) instruction is used to

call the subroutine. The CPU leaves the flow of the main program to execute the instructions in the subroutine. When the RTS instruction is executed, the CPU returns to the main program where it left off.

**synchronous** — Refers to two or more things made to happen simultaneously in a system by means of a common clock signal.

**system integration module (SIM)** — One of a number of modules that handle a variety of control functions in the modular M68HC08 Family. The SIM controls mode of operation, resets and interrupts, and system clock generation.

**table** — A collection or ordering of data (such as square root values) laid out in rows and columns and stored in a computer memory as an array.

**two's complement** — A means of performing binary subtraction using addition techniques. The most significant bit of a two's complement number indicates the sign of the number (1 indicates negative). The two's complement negative of a number is obtained by inverting each bit in the number and then adding 1 to the result.

**unsigned** — Refers to a binary number representation in which all numbers are assumed positive. With signed binary, the most significant bit is used to indicate whether the number is positive or negative, normally 0 for positive and 1 for negative, and the other seven bits are used to indicate the magnitude.

**variable** — A value that changes during the course of executing a program.

**word** — Two bytes or 16 bits, treated as a unit.

**write** — The transfer of a byte of data from the CPU to a memory location.

**X** — Abbreviation for the lower byte of the index register (H:X) in the CPU08.

**Z** — Abbreviation for zero, a bit in the condition code register of the CPU08. The CPU08 sets the zero flag when an arithmetic operation, logical operation, or data manipulation produces a result of \$00.

**zero** — A logic low level, a voltage level approximately equal to the ground voltage ( $V_{SS}$ ).

## Index

### A

Accumulator (A).....	25
Addressing modes	
direct .....	61
extended .....	63
immediate.....	59
indexed with post increment.....	77
indexed, 16-bit offset .....	66
indexed, 8-bit offset .....	65
indexed, 8-bit offset with post increment .....	78
indexed, no offset.....	65
inherent .....	57
memory to memory direct to direct.....	73
memory to memory direct to indexed with post increment.....	76
memory to memory immediate to direct.....	73
memory to memory indexed to direct with post increment.....	74
relative .....	71
stack pointer, 16-bit offset .....	68
stack pointer, 8-bit offset .....	68

### C

Carry/borrow flag (C).....	29
Condition code register (CCR)	
carry/borrow flag (C) .....	29
half-carry flag (H) .....	28
interrupt mask (I) .....	29
negative flag (N).....	29
overflow flag (V).....	28
zero flag (Z) .....	29

<b>CPU08</b>	
accumulator (A) .....	25
block diagram .....	30
condition code register (CCR) .....	28
control unit .....	32
execution unit .....	33
features .....	20
functional description .....	30
index register (HX) .....	25
instruction execution .....	33–35
internal timing .....	31
low-power modes .....	22
program counter (PC) .....	27
programming model .....	24
registers .....	24
stack pointer (SP) .....	26
<b>D</b>	
Direct addressing mode .....	61
DMA (direct memory access module) .....	39
<b>E</b>	
Extended addressing mode .....	63
<b>H</b>	
HX (index register) .....	25
<b>I</b>	
Immediate addressing mode .....	59
Index register (HX) .....	25
Indexed with post increment addressing mode .....	77
Indexed, 16-bit offset addressing mode .....	66
Indexed, 8-bit offset addressing mode .....	65
Indexed, 8-bit offset with post increment addressing mode .....	78
Indexed, no offset addressing mode .....	65
Inherent addressing mode .....	57

Instruction execution .....	33–35
instruction boundaries .....	34
Instruction set	
convention definition .....	96
nomenclature .....	92
Interrupts	
allocating scratch space .....	53
arbitration .....	41
DMA (direct memory access module) .....	39
flow and timing .....	42
H register storage .....	41
interrupt processing .....	49
interrupt recognition .....	43–44
masking .....	43
nesting of multiple interrupts .....	50, 52
priority .....	51
recognition .....	39
return to calling program .....	45
SIM (system integration module) .....	41
sources .....	51
stacking .....	40
vectors .....	51

## L

Legal label .....	95
Literal expression .....	95

## M

Memory to memory direct to direct addressing mode .....	73
Memory to memory direct to indexed	
with post increment addressing mode .....	76
Memory to memory immediate to direct addressing mode .....	73
Memory to memory indexed to direct with post increment	
addressing mode .....	74
Monitor mode .....	47

<b>N</b>	
Negative flag (N) .....	29
Notation	
Source forms .....	94
<b>O</b>	
Opcode map .....	88
Overflow flag (V) .....	28
<b>P</b>	
Program counter (PC) .....	27
<b>R</b>	
Register designators .....	95
Registers	
accumulator (A) .....	24
condition code (CCR) .....	24
index (HX) .....	24
program counter (PC) .....	24
stack pointer (SP) .....	24
Relative addressing mode	
conditional branch .....	71
Resets	
arbitration .....	41
CPU .....	47
DMA (direct memory access module) .....	39
exiting .....	46
external .....	49
H register storage .....	41
I bit .....	50
initial conditions .....	47
internal .....	49
local enable mask bits .....	50
masking .....	43
mode selection .....	47
monitor mode .....	47
recognition .....	39
resetting processing .....	46



Resets (continued)	
SIM (system integration module) .....	41, 48
sources .....	48
stacking .....	40
user mode .....	47

**S**

SIM (system integration module) .....	41, 48
Source form notation .....	94
Stack pointer (SP) .....	26
Stack pointer, 16-bit offset addressing mode .....	68
Stack pointer, 8-bit offset addressing mode .....	68
System integration module (SIM) .....	41, 48

**T**

Timing	
control unit .....	33
internal .....	31
interrupt processing flow .....	42

**U**

User mode .....	47
-----------------	----

**V**

V (overflow flag) .....	28
-------------------------	----

**Z**

Zero flag (Z) .....	29
---------------------	----





## How to Reach Us:

### **USA/EUROPE/LOCATIONS NOT LISTED:**

Motorola Literature Distribution  
P.O. Box 5405  
Denver, Colorado 80217  
1-303-675-2140  
1-800-441-2447

### **TECHNICAL INFORMATION CENTER:**

1-800-521-6274

### **JAPAN:**

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

### **ASIA/PACIFIC:**

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

### **HOME PAGE:**

<http://www.motorola.com/semiconductors/>



**MOTOROLA**

**CPU08RM/AD  
REV 3**