6.5.1.1 Loads and Stores

Table 6-1. Load and Store Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	I N Z C
LDA #*opr8i*	Load Accumulator from Memory A ← (M)	IMM	`A6 ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
LDA *opr8a*		DIR	`B6 dd`	3	`rpp`		
LDA *opr16a*		EXT	`C6 hh ll`	4	`prpp`		
LDA *oprx16*,X		IX2	`D6 ee ff`	4	`prpp`		
LDA *oprx8*,X		IX1	`E6 ff`	3	`rpp`		
LDA ,X		IX	`F6`	3	`rfp`		
LDA *oprx16*,SP		SP2	`9E D6 ee ff`	5	`pprpp`		
LDA *oprx8*,SP		SP1	`9E E6 ff`	4	`prpp`		
LDHX #*opr16i*	Load Index Register (H:X) H:X ← (M:M + $0001)	IMM	`45 jj kk`	3	`ppp`	0 1 1 –	– ↕ ↕ –
LDHX *opr8a*		DIR	`55 dd`	4	`rrpp`		
LDHX *opr16a*		EXT	`32 hh ll`	5	`prrpp`		
LDHX ,X		IX	`9E AE`	5	`prrfp`		
LDHX *oprx16*,X		IX2	`9E BE ee ff`	6	`pprrpp`		
LDHX *oprx8*,X		IX1	`9E CE ff`	5	`prrpp`		
LDHX *oprx8*,SP		SP1	`9E FE ff`	5	`prrpp`		
LDX #*opr8i*	Load X (Index Register Low) from Memory X ← (M)	IMM	`AE ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
LDX *opr8a*		DIR	`BE dd`	3	`rpp`		
LDX *opr16a*		EXT	`CE hh ll`	4	`prpp`		
LDX *oprx16*,X		IX2	`DE ee ff`	4	`prpp`		
LDX *oprx8*,X		IX1	`EE ff`	3	`rpp`		
LDX ,X		IX	`FE`	3	`rfp`		
LDX *oprx16*,SP		SP2	`9E DE ee ff`	5	`pprpp`		
LDX *oprx8*,SP		SP1	`9E EE ff`	4	`prpp`		
STA *opr8a*	Store Accumulator in Memory M ← (A)	DIR	`B7 dd`	3	`wpp`	0 1 1 –	– ↕ ↕ –
STA *opr16a*		EXT	`C7 hh ll`	4	`pwpp`		
STA *oprx16*,X		IX2	`D7 ee ff`	4	`pwpp`		
STA *oprx8*,X		IX1	`E7 ff`	3	`wpp`		
STA ,X		IX	`F7`	2	`wp`		
STA *oprx16*,SP		SP2	`9E D7 ee ff`	5	`ppwpp`		
STA *oprx8*,SP		SP1	`9E E7 ff`	4	`pwpp`		
STHX *opr8a*	Store H:X (Index Reg.) (M:M + $0001) ← (H:X)	DIR	`35 dd`	4	`wwpp`	0 1 1 –	– ↕ ↕ –
STHX *opr16a*		EXT	`96 hh ll`	5	`pwwpp`		
STHX *oprx8*,SP		SP1	`9E FF ff`	5	`pwwpp`		
STX *opr8a*	Store X (Low 8 Bits of Index Register) in Memory M ← (X)	DIR	`BF dd`	3	`wpp`	0 1 1 –	– ↕ ↕ –
STX *opr16a*		EXT	`CF hh ll`	4	`pwpp`		
STX *oprx16*,X		IX2	`DF ee ff`	4	`pwpp`		
STX *oprx8*,X		IX1	`EF ff`	3	`wpp`		
STX ,X		IX	`FF`	2	`wp`		
STX *oprx16*,SP		SP2	`9E DF ee ff`	5	`ppwpp`		
STX *oprx8*,SP		SP1	`9E EF ff`	4	`pwpp`		

Load A and load X cause an 8-bit value to be read from memory into accumulator A or into the X register. Load H:X causes one 8-bit value to be read from memory into the H register and a second 8-bit value to be read from the next sequential memory location into the X register. Load A and load X each allow eight different addressing modes for maximum flexibility in accessing memory. LDHX allows seven different addressing modes to specify the memory locations of the values being read.

The following instructions demonstrate some of the uses for load instructions. This collection of instructions is not intended to be a meaningful program. Rather, they are unrelated load instructions to demonstrate the many possible addressing modes that allow access to memory in different ways.

```
               ; load A - various addressing modes
               ; immediate (IMM) addressing mode examples
C089 A6 55                  lda    #$55          ;IMM - $ means hexadecimal
C08B A6 64                  lda    #100          ;decimal 100 (hexadecimal $64)
C08D A6 3F                  lda    #%00111111    ;% means binary
C08F A6 41                  lda    #'A'          ;single quotes around ASCII
C091 A6 8D                  lda    #illegalOp    ;label used as immediate value
               ; direct (DIR) addressing mode examples
C093 B6 55                  lda    $55           ;load from address $0055
C095 B6 9D                  lda    directByte    ;label as a direct address
               ; extended (EXT) addressing mode
C097 C6 FFFE                lda    $FFFE         ;high byte of reset vector
C09A C6 0101                lda    extByte       ;label used as an address
C09D C6 C09D                lda    *             ;* means "here", loads opcode
C0A0 C6 009D                lda    fwdRef        ;forces ext addressing mode
               ; not all assemblers treat forward references the same way
     0000 009D fwdRef:      equ    directByte    ;forward referenced direct

C0A3 45 C007                ldhx   #stringBytes  ;point at string in flash
               ; indexed addressing mode (relative to H:X index register pair)
C0A6 D6 4081                lda    (moveBlk1-stringBytes),x  ;IX2 mode
C0A9 E6 01                  lda    1,x           ;IX1 - 8-bit offset
C0AB F6                     lda    ,x            ;IX - no offset

               ; indexed addressing mode (relative to SP stack pointer)
C0AC 45 0001                ldhx   #1
C0AF 94                     txs                  ;temp move SP for 16-bit offset ex.
C0B0 9ED6 012C              lda    300,sp        ;SP2 - 16-bit offset
C0B4 9EE6 01                lda    1,sp          ;SP1 - 8-bit offset
```

Since one operand input to the arithmetic logic unit (ALU) is connected to the A accumulator, you typically need to use an LDA instruction to read one value into A before performing mathematical or logical operations involving a second operand.

```
; add A + B (assumes sum is < or = 255)
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
```

In some cases, you can plan your program so that the results that were stored in accumulator A as the result of one operation can be used as an operand in a subsequent operation. This can save the need to store one result and reload the accumulator with the next operand.

```
; add A + B + C (assumes sum is < or = 255)
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
             add    oprC          ;accum. + oprC -> accum.
```

The next example shows an intermediate value being saved on the stack. This is sometimes faster than storing temporary results in memory. The amount of savings depends on what addressing mode would be needed to store the temporary value in memory and whether the X register was needed for something else at the time.

```
; compute (A + B) - (C + D) (assumes no carry or borrow)
             lda    oprC          ;oprC -> accumulator
             add    oprD          ;oprC + oprD -> accumulator
             psha                 ;intermediate result to SP+1
             lda    oprA          ;oprA -> accumulator
             add    oprB          ;oprA + oprB -> accumulator
             sub    1,sp          ;(A+B)-(C+D) to accumulator
             ais    #1            ;deallocate local space
```

Table 6-2. BSET, BCLR, Move, and Transfer Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	I N Z C

BSET *n,opr8a*	Set Bit *n* in Memory (Mn ← 1)	DIR (b0)
		DIR (b1)
		DIR (b2)
		DIR (b3)
		DIR (b4)
		DIR (b5)
		DIR (b6)
		DIR (b7)
			`10 dd`
			`12 dd`
			`14 dd`
			`16 dd`
			`18 dd`
			`1A dd`
			`1C dd`
			`1E dd`
				5
				5
				5
				5
				5
				5
				5
				5
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
						– 1 1 –	– – – –

BCLR *n,opr8a*	Clear Bit n in Memory
	(Mn ← 0)
		DIR (b0)
		DIR (b1)
		DIR (b2)
		DIR (b3)
		DIR (b4)
		DIR (b5)
		DIR (b6)
		DIR (b7)
			`11 dd`
			`13 dd`
			`15 dd`
			`17 dd`
			`19 dd`
			`1B dd`
			`1D dd`
			`1F dd`
				5
				5
				5
				5
				5
				5
				5
				5
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
					`rfwpp`
						– 1 1 –	– – – –

MOV *opr8a,opr8a*
MOV *opr8a*,X+
MOV #*opr8i,opr8a*
MOV ,X+,*opr8a*
	Move
	$(M)_{destination} \leftarrow (M)_{source}$
	In IX+/DIR and DIR/IX+ Modes,
	H:X ← (H:X) + $0001
		DIR/DIR
		DIR/IX+
		IMM/DIR
		IX+/DIR
			`4E dd dd`
			`5E dd`
			`6E ii dd`
			`7E dd`
				5
				5
				4
				5
					`rpwpp`
					`rfwpp`
					`pwpp`
					`rfwpp`
						0 1 1 –	– ↕ ↕ –

TAX	Transfer Accumulator to X (Index Register Low) X ← (A)	INH	`97`	1	`p`	– 1 1 –	– – – –
TXA	Transfer X (Index Reg. Low) to Accumulator A ← (X)	INH	`9F`	1	`p`	– 1 1 –	– – – –
TAP	Transfer Accumulator to CCR CCR ← (A)	INH	`84`	1	`p`	↕ 1 1 ↕	↕ ↕ ↕ ↕
TPA	Transfer CCR to Accumulator A ← (CCR)	INH	`85`	1	`p`	– 1 1 –	– – – –
NSA	Nibble Swap Accumulator A ← (A[3:0]:A[7:4])	INH	`62`	1	`p`	– 1 1 –	– – – –

6.5.1.2 Bit Set and Bit Clear

Bit set (BSET) and bit clear (BCLR) instructions can be thought of as bit-sized store instructions, but these instructions actually read a full 8-bit location, modify the specified bit, and then re-write the whole 8-bit location. In certain cases, such as when the target location is something other than a RAM variable, this subtle behavior can lead to unexpected results. If a BSET or BCLR instruction attempts to change a bit in a nonvolatile memory location, naturally, the bit will not change because nonvolatile memories require a more complex sequence of operations to make changes.

Some status bits are cleared by a sequence involving a read of the status bit followed by a write to another register in the peripheral module. Some users are surprised to find that a BSET or BCLR instruction has satisfied the requirement to read the status register. To avoid such problems, just remember that the BSET and BCLR instructions are read-modify-write instructions that access a full 8-bit location in parallel.

Some control or I/O registers do not access the same physical logic states for reads and writes. In general, do not use read-modify-write instructions on these locations because they may produce unexpected results.

```
                  ; BSET example - turns on TE without changing RE
C0D3 16 1B                     bset   TE,SCI1C2     ;enable SCI transmitter
                  ; functionally equivalent to...
C0D5 B6 1B                     lda    SCI1C2        ;read current SCCR2 value
C0D7 AA 08                     ora    #mTE           ;OR in TE bit (mask)
C0D9 B7 1B                     sta    SCI1C2        ;upate value in SCCR2
```

6.5.1.3 Memory-to-Memory Moves

Move instructions can be helpful in an accumulator architecture like the HCS08 where the number of registers is limited. MOV performs a read of an 8-bit value from one memory location and stores the value in a different location. Like the load and store instructions, MOV causes the N and Z bits in the CCR to be updated according to the value of the data being moved.

Although load and store instructions could be used to do the same thing as a MOV instruction, MOV does not require the accumulator to be saved so that A can be used as the transport means for the move operation. In many cases, the MOV approach is faster and smaller (object code size) than the load-store combination. MOV allows four different address mode combinations to specify the source and destination locations for the move.

The following example shows how move instructions can be used to initialize several register values.

```
C0DB 6E 03 00                  mov    #$03,PTAD     ;0011 to 4 LS bits
C0DE 6E 0F 03                  mov    #$0F,PTADD    ;make 4 LS bits outputs
C0E1 6E F0 01                  mov    #$F0,PTAPE    ;pullups on 4 MS bits
```

The next example shows a string move operation using load and store instructions rather than move instructions.

```
                  ; block move example to move a string to a RAM block
C0E4 45 0088                   ldhx   #moveBlk1     ;point at destination block
C0E7 D6 BF7F      movLoop1:    lda    (stringBytes-moveBlk1),x  ;get source byte
C0EA 27 04                     beq    dunLoop1      ;null terminator ends loop
C0EC F7                        sta    ,x            ;save to destination block
C0ED 5C                        incx                 ;next location (assumes DIR)
C0EE 20 F7                     bra    movLoop1      ;continue loop
                  dunLoop1:
```

6.5.1.4 Register Transfers and Nibble Swap

TAX and TXA offer an efficient way to transfer a value from A to X or from X to A. Depending on whether the X register is already being used, this can be an efficient way to temporarily save the accumulator value so A can be used for some other operation.

TAP and TPA provide a means for moving the value from A into the CCR (processor status byte) or from the CCR into A. This is used more in development tools like debug monitors than in normal user programs.

The nibble swap A (NSA) instruction exchanges the upper and lower nibbles of the accumulator (A). An 8-bit value is called a byte and a nibble is the upper- or lower-order four bits of a byte. Each nibble corresponds to exactly one hexadecimal digit. This instruction is useful for conversions between binary or hexadecimal and ASCII, and for operations on binary-coded-decimal (BCD) numbers.

```
*********************
* chexl - convert upper nibble of A to ASCII
* chexr - convert lower nibble of A to ASCII
*  on entry A contains any binary (hexadecimal) number
*  returns with resulting ASCII character in A
*********************
chexl:       nsa                  ;swap nibble into low half
chexr:       and    #$0F          ;strip off upper nibble
             add    #$30          ;now $30 - $3F
             cmp    #$39          ;check for < or = '9'
             bls    dunChex       ;if so, just return
             add    #7            ;adjust to $41-$46
dunChex:     rts                  ;return with ASCII in A
*********************
```

6.5.2 Math Instructions

Math instructions include the traditional add, subtract, multiply, and divide operations, a collection of utility instructions including increment, decrement, clear, negate (two's complement), compare, and test, and a decimal adjust instruction for computations involving BCD numbers. The compare instructions are actually subtract operations where the CCR bits are affected but the result is not written back to a CPU register. The test instructions affect the N and Z condition code bits, but do not affect the tested value.

6.5.2.1 Add, Subtract, Multiply, and Divide

Table 6-3. Add, Subtract, Multiply, and Divide Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	**I N Z C**
ADC *#opr8i*		IMM	`A9 ii`	2	`pp`		
ADC *opr8a*		DIR	`B9 dd`	3	`rpp`		
ADC *opr16a*		EXT	`C9 hh ll`	4	`prpp`		
ADC *oprx16*,X	Add with Carry	IX2	`D9 ee ff`	4	`prpp`	↕ 1 1 ↕	– ↕ ↕ ↕
ADC *oprx8*,X	A ← (A) + (M) + (C)	IX1	`E9 ff`	3	`rpp`		
ADC ,X		IX	`F9`	3	`rfp`		
ADC *oprx16*,SP		SP2	`9E D9 ee ff`	5	`pprpp`		
ADC *oprx8*,SP		SP1	`9E E9 ff`	4	`prpp`		
ADD *#opr8i*		IMM	`AB ii`	2	`pp`		
ADD *opr8a*		DIR	`BB dd`	3	`rpp`		
ADD *opr16a*		EXT	`CB hh ll`	4	`prpp`		
ADD *oprx16*,X	Add without Carry	IX2	`DB ee ff`	4	`prpp`	↕ 1 1 ↕	– ↕ ↕ ↕
ADD *oprx8*,X	A ← (A) + (M)	IX1	`EB ff`	3	`rpp`		
ADD ,X		IX	`FB`	3	`rfp`		
ADD *oprx16*,SP		SP2	`9E DB ee ff`	5	`pprpp`		
ADD *oprx8*,SP		SP1	`9E EB ff`	4	`prpp`		
AIS *#opr8i*	Add Immediate Value (Signed) to Stack Pointer SP ← (SP) + (M)	IMM	`A7 ii`	2	`pp`	– 1 1 –	– – – –

Table 6-3. Add, Subtract, Multiply, and Divide Instructions (Continued)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR V 1 1 H	Effect on CCR I N Z C
AIX *#opr8i*	Add Immediate Value (Signed) to Index Register (H:X) H:X ← (H:X) + (M)	IMM	`AF ii`	2	`pp`	– 1 1 –	– – – –

SUB *#opr8i*
SUB *opr8a*
SUB *opr16a*
SUB *oprx16*,X
SUB *oprx8*,X
SUB ,X
SUB *oprx16*,SP
SUB *oprx8*,SP
	Subtract
	A ← (A) – (M)
		IMM
		DIR
		EXT
		IX2
		IX1
		IX
		SP2
		SP1
			`A0 ii`
			`B0 dd`
			`C0 hh ll`
			`D0 ee ff`
			`E0 ff`
			`F0`
			`9E D0 ee ff`
			`9E E0 ff`
				2
				3
				4
				4
				3
				3
				5
				4
					`pp`
					`rpp`
					`prpp`
					`prpp`
					`rpp`
					`rfp`
					`pprpp`
					`prpp`
						↕ 1 1 –	– ↕ ↕ ↕

SBC *#opr8i*
SBC *opr8a*
SBC *opr16a*
SBC *oprx16*,X
SBC *oprx8*,X
SBC ,X
SBC *oprx16*,SP
SBC *oprx8*,SP
	Subtract with Carry
	A ← (A) – (M) – (C)
		IMM
		DIR
		EXT
		IX2
		IX1
		IX
		SP2
		SP1
			`A2 ii`
			`B2 dd`
			`C2 hh ll`
			`D2 ee ff`
			`E2 ff`
			`F2`
			`9E D2 ee ff`
			`9E E2 ff`
				2
				3
				4
				4
				3
				3
				5
				4
					`pp`
					`rpp`
					`prpp`
					`prpp`
					`rpp`
					`rfp`
					`pprpp`
					`prpp`
						↕ 1 1 –	– ↕ ↕ ↕

MUL	Unsigned multiply X:A ← (X) × (A)	INH	`42`	5	`ffffp`	– 1 1 0	– – – 0
DIV	Divide A ← (H:A)÷(X); H ← Remainder	INH	`52`	6	`fffffp`	– 1 1 –	– – ↕ ↕

The ADD instructions add the value in A to a memory operand and store the result in A. ADC adds the value in A, plus the carry bit from a previous operation, to a memory operand and stores the result in A. This operation allows performance of multibyte additions as demonstrated by the following example.

```
; add 8-bit operand to 24-bit sum
             lda    oprA          ;8-bit operand to A
             add    sum24+2       ;LS byte of 24-bit sum
             sta    sum24+2       ;update LS byte
             lda    sum24+1       ;middle byte of 24-bit sum
             adc    #0            ;propigate any carry
             sta    sum24+1       ;update middle byte
             lda    sum24         ;get MS byte of 24-bit sum
             adc    #0            ;propigate carry into MS byte
             sta    sum24         ;update MS byte
```

The AIX instruction adds a signed 8-bit value to the 16-bit H:X index register pair and stores the result back into H:X. Unlike other arithmetic instructions, AIX does not affect the CCR bits.

```
             ldhx   #tblOfStruct  ;H:X pointing at first struct
; aix to update pointer into table of 5-byte structures
             aix    #5            ;point to next 5-byte struct
```

The SUB instructions subtract a memory operand from the value in A and store the result in A. The carry status bit acts as a borrow indicator for this subtraction. SBC subtracts a memory operand and the carry bit from a previous operation from the value in A and stores the result back in A. This operation allows performance of multibyte subtractions as demonstrated by the following example.

```
; 16-bit subtract... result16 = oprE - oprF
             lda    oprE+1        ;low half of oprE
             sub    oprF+1        ;oprE(lo) - oprF(lo)
             sta    result16+1    ;low half of result
             lda    oprE          ;high half of oprE
             sbc    oprF          ;oprE(hi) - oprF(hi) - borrow
             sta    result16      ;high half of result
```

MUL multiplies the unsigned 8-bit value in X by the unsigned 8-bit value in A and stores the 16-bit result in X:A where the upper eight bits of the result are stored in X and the lower eight bits of the result are in A. There is no possibility of a carry (or overflow) since the result will always fit into X:A, so C is cleared after this operation.

DIV divides the 16-bit unsigned value in H:A by the 8-bit unsigned value in X and stores the 8-bit result in A and the 8-bit remainder in H. The divisor in X is left unchanged so it could be used in later calculations. Z indicates whether the result was zero, and C indicates whether there was an attempt to divide by zero or if there was an overflow. An overflow will occur if the result was greater than 255.

This first divide example shows a simple 8-bit by 8-bit integer divide to get an 8-bit result.

```
; divide examples
; 8/8 integer divide... A = A/X
             clrh                 ;clear MS byte of dividend
             lda    divid8        ;load 8-bit dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient8     ;save result
```

The second divide example demonstrates how to use DIV to perform an 8-bit by 8-bit divide and another DIV to resolve the remainder into a fractional result (eight more places to the right of the radix point).

```
; 8/8 integer divide, resolve remainder to 8 fractional bits...
; r8.f8 = A/X, remainder resolved into 8-bit binary fraction
; 16-bit result -> (8-bit integer result).(8-bit fraction)
             clrh                 ;clear MS byte of dividend
             lda    divid8        ;load 8-bit dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient16    ;upper integer part of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient16+1  ;lower fractional part
```

In the third divide example, we divide an 8-bit dividend by a larger 8-bit divisor to get a 16-bit fractional result where the radix point is just left of the MSB of the result. In a binary fraction, the MSB has a weight of one-half, the next bit to the right has a weight of one-fourth, and so on.

```
; 8/8 fractional divide, 16-bit fractional result
; .r16 = H/X, result is a 16-bit binary fraction
; radix assumed to be in same position for H and X
; 16-bit result -> .(16-bit fraction)
; divid8 and divisor defined so H & X both loaded with one ldhx
             clra                 ;clear LS byte of dividend
             ldhx   divid8        ;H:X = dividend:divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient16    ;upper byte of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient16+1  ;next 8 bits of result
```

The fourth divide example uses a technique like long division to do an unbounded 16-bit by 8-bit integer divide.

```
; unbounded 16/8 integer divide (equivalent to long division)
; r16.f8 = H:A/X, result is 16-bit int.8-bit binary fraction
             clrh                 ;clear MS byte of dividend
             lda    divid16       ;upper byte of dividend
             ldx    divisor       ;load divisor
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient24    ;upper byte of result
             lda    divid16+1     ;H:A = remainder:dividend(lo)
             div                  ;H:A/X -> A, remainder -> H
             sta    quotient24+1  ;next byte of result
             clra                 ;H:A = remainder:0
             div                  ;H:A/X -> A
             sta    quotient24+2  ;fractional bits of result
```

The fifth divide example demonstrates a 16-bit by 8-bit divide with overflow checking.

```
; bounded 16/8 integer divide (with overflow checking)
; r8 = H:A/X, result is 8-bit integer
             ldhx   divid16       ;H:X = 16-bit dividend
             txa                  ;H:A = 16-bit dividend
             ldx    divisor       ;X = 8-bit divisor
             div                  ;H:A/X -> A, remainder -> H
             bcs    divOvrflow    ;Overflow?
             sta    quotient8     ;upper byte of result

divOvrflow:                       ;here on overflow
```

Table 6-4. Other Arithmetic Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	I N Z C
INC *opr8a*	Increment M ← (M) + $01	DIR	`3C dd`	5	`rfwpp`	↕ 1 1 –	– ↕ ↕ –
INCA	A ← (A) + $01	INH	`4C`	1	`p`		
INCX	X ← (X) + $01	INH	`5C`	1	`p`		
INC *oprx8*,X	M ← (M) + $01	IX1	`6C ff`	5	`rfwpp`		
INC ,X	M ← (M) + $01	IX	`7C`	4	`rfwp`		
INC *oprx8*,SP	M ← (M) + $01	SP1	`9E 6C ff`	6	`prfwpp`		
DEC *opr8a*	Decrement M ← (M) – $01	DIR	`3A dd`	5	`rfwpp`	↕ 1 1 –	– ↕ ↕ –
DECA	A ← (A) – $01	INH	`4A`	1	`p`		
DECX	X ← (X) – $01	INH	`5A`	1	`p`		
DEC *oprx8*,X	M ← (M) – $01	IX1	`6A ff`	5	`rfwpp`		
DEC ,X	M ← (M) – $01	IX	`7A`	4	`rfwp`		
DEC *oprx8*,SP	M ← (M) – $01	SP1	`9E 6A ff`	6	`prfwpp`		
CLR *opr8a*	Clear M ← $00	DIR	`3F dd`	5	`rfwpp`	0 1 1 –	– 0 1 –
CLRA	A ← $00	INH	`4F`	1	`p`		
CLRX	X ← $00	INH	`5F`	1	`p`		
CLRH	H ← $00	INH	`8C`	1	`p`		
CLR *oprx8*,X	M ← $00	IX1	`6F ff`	5	`rfwpp`		
CLR ,X	M ← $00	IX	`7F`	4	`rfwp`		
CLR *oprx8*,SP	M ← $00	SP1	`9E 6F ff`	6	`prfwpp`		
NEG *opr8a*	Negate M ← – (M) = $00 – (M)	DIR	`30 dd`	5	`rfwpp`	↕ 1 1 –	– ↕ ↕ ↕
NEGA	(Two's Complement) A ← – (A) = $00 – (A)	INH	`40`	1	`p`		
NEGX	X ← – (X) = $00 – (X)	INH	`50`	1	`p`		
NEG *oprx8*,X	M ← – (M) = $00 – (M)	IX1	`60 ff`	5	`rfwpp`		
NEG ,X	M ← – (M) = $00 – (M)	IX	`70`	4	`rfwp`		
NEG *oprx8*,SP	M ← – (M) = $00 – (M)	SP1	`9E 60 ff`	6	`prfwpp`		
CMP #*opr8i*		IMM	`A1 ii`	2	`pp`	↕ 1 1 –	– ↕ ↕ ↕
CMP *opr8a*		DIR	`B1 dd`	3	`rpp`		
CMP *opr16a*	Compare Accumulator with Memory	EXT	`C1 hh ll`	4	`prpp`		
CMP *oprx16*,X	A – M	IX2	`D1 ee ff`	4	`prpp`		
CMP *oprx8*,X	(CCR Updated But Operands Not Changed)	IX1	`E1 ff`	3	`rpp`		
CMP ,X		IX	`F1`	3	`rfp`		
CMP *oprx16*,SP		SP2	`9E D1 ee ff`	5	`pprpp`		
CMP *oprx8*,SP		SP1	`9E E1 ff`	4	`prpp`		
CPHX *opr16a*	Compare Index Register (H:X) with Memory	EXT	`3E hh ll`	6	`prrfpp`	↕ 1 1 –	– ↕ ↕ ↕
CPHX #*opr16i*	(H:X) – (M:M + $0001)	IMM	`65 jj kk`	3	`ppp`		
CPHX *opr8a*	(CCR Updated But Operands Not Changed)	DIR	`75 dd`	5	`rrfpp`		
CPHX *oprx8*,SP		SP1	`9E F3 ff`	6	`prrfpp`		
CPX #*opr8i*		IMM	`A3 ii`	2	`pp`	↕ 1 1 –	– ↕ ↕ ↕
CPX *opr8a*		DIR	`B3 dd`	3	`rpp`		
CPX *opr16a*	Compare X (Index Register Low) with Memory	EXT	`C3 hh ll`	4	`prpp`		
CPX *oprx16*,X	X – M	IX2	`D3 ee ff`	4	`prpp`		
CPX *oprx8*,X	(CCR Updated But Operands Not Changed)	IX1	`E3 ff`	3	`rpp`		
CPX ,X		IX	`F3`	3	`rfp`		
CPX *oprx16*,SP		SP2	`9E D3 ee ff`	5	`pprpp`		
CPX *oprx8*,SP		SP1	`9E E3 ff`	4	`prpp`		
TST *opr8a*	Test for Negative or Zero (M) – $00	DIR	`3D dd`	4	`rfpp`	0 1 1 –	– ↕ ↕ –
TSTA	(A) – $00	INH	`4D`	1	`p`		
TSTX	(X) – $00	INH	`5D`	1	`p`		
TST *oprx8*,X	(M) – $00	IX1	`6D ff`	4	`rfpp`		
TST ,X	(M) – $00	IX	`7D`	3	`rfp`		
TST *oprx8*,SP	(M) – $00	SP1	`9E 6D ff`	5	`prfpp`		
DAA	Decimal Adjust Accumulator After ADD or ADC of BCD Values	INH	`72`	1	`p`	U 1 1 –	– ↕ ↕ ↕

6.5.2.2 Increment, Decrement, Clear, and Negate

Increment and decrement instructions let you adjust the value in A, X, or a memory location by one. Clear instructions let you force an 8-bit value in A, X, H, or a memory location to zero.

Negate instructions perform a two's complement operation that is equivalent to multiplying a signed 8-bit value by negative one. Functionally, this instruction inverts all the bits in A, X, or the memory location and then adds one. The value $80 represents the signed number –128. The negative of this value would be +128, but the largest positive number that can be represented with a two's complement, 8-bit number is +127. If A was $80 and you execute a NEGA instruction, the CPU first inverts all the bits to get $7F and then adds one to get $80. Since this causes the sign to change from positive to negative, the V bit in the CCR is set to indicate the error.

6.5.2.3 Compare and Test

CMP instructions affect CCR bits exactly like the corresponding SUB instruction, but the result is not stored back into the accumulator so A and the memory operand are left unchanged. Compare instructions compare the contents of A, X, or the H:X register pair to a memory operand. In the case of CPHX, M is the address of the referenced memory location, H corresponds to memory location M, and X corresponds to memory location M+1. CPHX performs a 16-bit subtraction (without storing the result back to H:X).

The test instructions are equivalent to subtracting zero from A, X, or a memory operand. This operation clears V and sets or clears N and Z according to what was in the tested value. The tested value is not changed.

6.5.2.4 BCD Arithmetic

In a binary coded decimal (BCD) number, one hexadecimal digit (4 binary bits) represents a single decimal number from 0 to 9. When two 8-bit BDC numbers are added, the CPU actually does a normal binary addition. Depending on the BCD values involved, this could result in a value that is no longer a valid 2-digit BCD number. Based on the H and C condition code bits that resulted from an ADD or ADC instruction involving two legal BCD numbers, the decimal adjust A (DAA) instruction "corrects" the result to the proper BCD result and sets or clears the C bit as needed to reflect the result of the BCD addition. In the past, this was done with a relatively complex set of instructions that tested the values of each BCD digit of the result and the H and C bits. The DAA instruction greatly simplifies this operation.

The following examples demonstrate two of the possible cases that can result from adding 8-bit BDC numbers and the actions taken by a DAA instruction to correct the results to the appropriate BCD result and carry flag. The first example shows a BCD addition that does not require adjustment. The second example shows a case where the result was not a legal BCD value and the carry did not reflect the correct BCD result. In this second example, the DAA instruction adds a correction factor and adjusts the carry flag to reflect the correct BCD result.

```
             lda    #$11          ;BCD 11
             add    #$22          ;11 + 22 = 33
             daa                  ;no adjustment in this case
```

```
             LDA      #$59          ;BCD 59
             ADD      #$57          ;59 + 57 = $B0
; C=0, H=1, A=$B0 - wanted 59 + 57 = 116 or A=$16 with carry set
             DAA                    ;adds $66 and sets carry
; $B0 + $66 = $16 with carry bit set
```

6.5.3 Logical Operation Instructions

These instructions perform eight bitwise Boolean operations in parallel. For the complement instruction, each bit of the register or memory operand is inverted. The other logical instructions involve two operands, one in the accumulator (A) and the other in memory. Immediate, direct, extended, or indexed (relative to H:X or SP) addressing modes may be used to access the memory operand. Each bit of the accumulator is ANDed, ORed, or exclusive-ORed with the corresponding bit of the memory operand. The result of the logical operation is stored into the accumulator, overwriting the original operand.

Table 6-5. Logical Operation Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	I N Z C
AND #*opr8i*	Logical AND A ← (A) & (M)	IMM	`A4 ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
AND *opr8a*		DIR	`B4 dd`	3	`rpp`		
AND *opr16a*		EXT	`C4 hh ll`	4	`prpp`		
AND *oprx16*,X		IX2	`D4 ee ff`	4	`prpp`		
AND *oprx8*,X		IX1	`E4 ff`	3	`rpp`		
AND ,X		IX	`F4`	3	`rfp`		
AND *oprx16*,SP		SP2	`9E D4 ee ff`	5	`pprpp`		
AND *oprx8*,SP		SP1	`9E E4 ff`	4	`prpp`		
ORA #*opr8i*	Inclusive OR Accumulator and Memory A ← (A) | (M)	IMM	`AA ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
ORA *opr8a*		DIR	`BA dd`	3	`rpp`		
ORA *opr16a*		EXT	`CA hh ll`	4	`prpp`		
ORA *oprx16*,X		IX2	`DA ee ff`	4	`prpp`		
ORA *oprx8*,X		IX1	`EA ff`	3	`rpp`		
ORA ,X		IX	`FA`	3	`rfp`		
ORA *oprx16*,SP		SP2	`9E DA ee ff`	5	`pprpp`		
ORA *oprx8*,SP		SP1	`9E EA ff`	4	`prpp`		
EOR #*opr8i*	Exclusive OR Memory with Accumulator A ← (A ⊕ M)	IMM	`A8 ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
EOR *opr8a*		DIR	`B8 dd`	3	`rpp`		
EOR *opr16a*		EXT	`C8 hh ll`	4	`prpp`		
EOR *oprx16*,X		IX2	`D8 ee ff`	4	`prpp`		
EOR *oprx8*,X		IX1	`E8 ff`	3	`rpp`		
EOR ,X		IX	`F8`	3	`rfp`		
EOR *oprx16*,SP		SP2	`9E D8 ee ff`	5	`pprpp`		
EOR *oprx8*,SP		SP1	`9E E8 ff`	4	`prpp`		
COM *opr8a*	Complement M ← $(\overline{M})$= \$FF – (M)	DIR	`33 dd`	5	`rfwpp`	0 1 1 –	– ↕ ↕ 1
COMA	(One's Complement) A ← $(\overline{A})$ = \$FF – (A)	INH	`43`	1	`p`		
COMX	X ← $(\overline{X})$ = \$FF – (X)	INH	`53`	1	`p`		
COM *oprx8*,X	M ← $(\overline{M})$ = \$FF – (M)	IX1	`63 ff`	5	`rfwpp`		
COM ,X	M ← $(\overline{M})$ = \$FF – (M)	IX	`73`	4	`rfwp`		
COM *oprx8*,SP	M ← $(\overline{M})$ = \$FF – (M)	SP1	`9E 63 ff`	6	`prfwpp`		
BIT #*opr8i*	Bit Test (A) & (M) (CCR Updated but Operands Not Changed)	IMM	`A5 ii`	2	`pp`	0 1 1 –	– ↕ ↕ –
BIT *opr8a*		DIR	`B5 dd`	3	`rpp`		
BIT *opr16a*		EXT	`C5 hh ll`	4	`prpp`		
BIT *oprx16*,X		IX2	`D5 ee ff`	4	`prpp`		
BIT *oprx8*,X		IX1	`E5 ff`	3	`rpp`		
BIT ,X		IX	`F5`	3	`rfp`		
BIT *oprx16*,SP		SP2	`9E D5 ee ff`	5	`pprpp`		
BIT *oprx8*,SP		SP1	`9E E5 ff`	4	`prpp`		

6.5.3.1 AND, OR, Exclusive-OR, and Complement

These instructions provide the basic AND, OR, exclusive-OR, and invert functions needed to perform Boolean logical functions.

```
              lda     #$0C          ;bit pattern 00001100
              and     #$0A          ;bit pattern 00001010
; result is..........$08......................00001000

              lda     #$35          ;bit pattern 00110101
              and     #$0F          ;bit pattern 00001111
; result is..........$05......................00000101
```

You may notice some similarity between the AND operation and the BCLR instruction. However, BCLR can be used only on memory locations $0000–$00FF and can clear only one bit at a time while AND can clear any combination of bits and may be used with several different addressing modes to identify the memory operand to be ANDed with A.

```
              lda     #$0C          ;bit pattern 00001100
              ora     #$0A          ;bit pattern 00001010
; result is..........$0E......................00001110
```

You may notice some similarity between the ORA operation and the BSET instruction; however, BSET can be used only on memory locations $0000–$00FF and can set only one bit at a time while ORA can set any combination of bits and may be used with several different addressing modes to identify the memory operand to be ORed with A.

Exclusive-OR can be used to toggle bits in an operand. One operand is considered a mask where each bit that is set in the mask corresponds to a bit value in the other operand that will be toggled (inverted). The next example reads an I/O port, exclusive-ORs it with an immediate mask value of $03 to toggle the two least significant bits, and then writes the updated result to the I/O port.

```
C162 A6 0C                 lda     #$0C          ;bit pattern 00001100
C164 A8 0A                 eor     #$0A          ;bit pattern 00001010
                ; result is..........$06......................00000110

C166 B6 00                 lda     PTAD          ;read I/O port A
C168 A8 03                 eor     #$03          ;inverts 2 LSBs
C16A B7 00                 sta     PTAD          ;update I/O port A
```

Complement instructions simply invert each bit of the operand. Don’t confuse this with the negate instruction which performs the arithmetic operation equivalent to multiplication by minus one.

```
              lda     #$C5          ;bit pattern 11000101
              coma                  ;result is   00111010
```

6.5.3.2 BIT Instruction

The BIT instruction ANDs each bit of A with the corresponding bit of the addressed memory operand (just like AND), but the result is not stored to the accumulator. The N and Z condition codes are set or cleared according to the results of the AND operation to allow conditional branches after the BIT instruction. If you load A with a mask value where each bit that is set in the mask corresponds to a bit in the memory operand to be tested, then execute a BIT instruction, the Z bit will be set if none of the tested bits were 1s.

```
             lda    SCI1S1             ;read SCI status register
             bit    #(mOR+mNF+mFE+mPF) ;mask of all error flags
             bne    sciError           ;branch if any flags set
; A still contains undisturbed status register

sciError:                              ;here if any error flags
```

6.5.4 Shift and Rotate Instructions

All of the shift and rotate instructions operate on a 9-bit field consisting of an 8-bit value in A, X, or a memory location and the C bit in the CCR. Drawings are provided in the instruction descriptions to show where the C bit fits into the shift or rotate operation. The logical shift instructions are simple shifts which shift a zero into the first bit of the value and shift the last bit into the carry bit. The arithmetic shifts treat the value to be shifted as a signed two’s complement number. An arithmetic shift left is like multiplying a value by 2 and an arithmetic shift right is like dividing the number by 2. The arithmetic shift right (ASR) instruction copies the original most significant bit (MSB) back into the MSB to preserve the sign of the operand. ASL and LSL are just two different mnemonics for the same instruction because there is no functional difference between the logical and arithmetic shifts to the left.

Table 6-6. Shift and Rotate Instructions

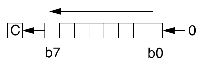
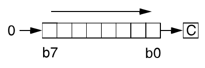
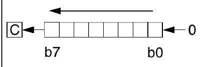
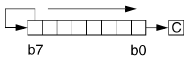
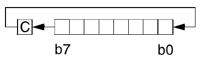
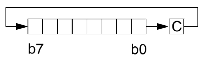
Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	I N Z C
LSL *opr8a*	Logical Shift Left  (Same as ASL)	DIR	38 dd	5	rfwpp	↕ 1 1 –	– ↕ ↕ ↕
LSLA		INH	48	1	p		
LSLX		INH	58	1	p		
LSL *oprx8*,X		IX1	68 ff	5	rfwpp		
LSL ,X		IX	78	4	rfwp		
LSL *oprx8*,SP		SP1	9E 68 ff	6	prfwpp		
LSR *opr8a*	Logical Shift Right 	DIR	34 dd	5	rfwpp	↕ 1 1 –	– 0 ↕ ↕
LSRA		INH	44	1	p		
LSR*X*		INH	54	1	p		
LSR *oprx8*,X		IX1	64 ff	5	rfwpp		
LSR ,X		IX	74	4	rfwp		
LSR *oprx8*,SP		SP1	9E 64 ff	6	prfwpp		
ASL *opr8a*	Arithmetic Shift Left  (Same as LSL)	DIR	38 dd	5	rfwpp	↕ 1 1 –	– ↕ ↕ ↕
ASLA		INH	48	1	p		
ASLX		INH	58	1	p		
ASL *oprx8*,X		IX1	68 ff	5	rfwpp		
ASL ,X		IX	78	4	rfwp		
ASL *oprx8*,SP		SP1	9E 68 ff	6	prfwpp		

Table 6-6. Shift and Rotate Instructions (Continued)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR V 1 1 H	Effect on CCR I N Z C
ASR *opr8a* ASRA ASRX ASR *oprx8*,X ASR ,X ASR *oprx8*,SP	Arithmetic Shift Right 	DIR INH INH IX1 IX SP1	37 dd 47 57 67 ff 77 9E 67 ff	5 1 1 5 4 6	rfwpp p p rfwpp rfwp prfwpp	↕ 1 1 –	– ↕ ↕ ↕
ROL *opr8a* ROLA ROLX ROL *oprx8*,X ROL ,X ROL *oprx8*,SP	Rotate Left through Carry 	DIR INH INH IX1 IX SP1	39 dd 49 59 69 ff 79 9E 69 ff	5 1 1 5 4 6	rfwpp p p rfwpp rfwp prfwpp	↕ 1 1 –	– ↕ ↕ ↕
ROR *opr8a* RORA RORX ROR *oprx8*,X ROR ,X ROR *oprx8*,SP	Rotate Right through Carry 	DIR INH INH IX1 IX SP1	36 dd 46 56 66 ff 76 9E 66 ff	5 1 1 5 4 6	rfwpp p p rfwpp rfwp prfwpp	↕ 1 1 –	– ↕ ↕ ↕

Including the carry bit in the shifts and rotates allows extension of these operations to multibyte values. The following examples show a 24-bit value being shifted either right or left.

```
; 24-bit left shift
             clc                    ;clear C bit
; initial condition sum24 = hhhh hhhh : mmmm mmmm : llll llll : 0
             lsl     sum24+2        ;C to LSB of low byte
; now sum24 = hhhh hhhh : mmmm mmmm : C=l(7) : llll lll0
             rol     sum24+1        ;rotate middle byte
; now sum24 = hhhh hhhh : C=m(7) : mmmm mmml : llll lll0
             rol     sum24          ;rotate high byte
; now sum24 = C=h(7) : hhhh hhhm : mmmm mmml : llll lll0
```

```
; 24-bit right shift
             clc                    ;clear C bit
; initial condition sum24 = 0 : hhhh hhhh : mmmm mmmm : llll llll
             lsr     sum24          ;C to MSB of high byte
; now sum24 = 0hhh hhhh : C=h(0) : mmmm mmmm : llll llll
             rol     sum24+1        ;rotate middle byte
; now sum24 = 0hhh hhhh : hmmm mmmm : C=m(0) : llll lll0
             rol     sum24+2        ;rotate low byte
; now sum24 = 0hhh hhhm : hmmm mmmm : mlll llll : C=l(0)
```

Figure 6-5. Multibyte Shifts

6.5.5 Jump, Branch, and Loop Control Instructions

The instructions in this group cause a change of flow which means that the CPU loads a new address into the program counter so program execution continues at a location other than the next memory location after the current instruction.

Jump instructions cause an unconditional change in the execution sequence to a new location in a program. Branch and loop control instructions cause a conditional change in the execution sequence. Branch and loop control instructions use relative addressing mode to conditionally branch to a location that is relative to the location of the branch. Processor status indicators in the CCR control whether a conditional branch or loop control instruction will branch to a new address or simply continue to the next instruction in the program. BRA is a special case because the branch *always* occurs and BRN is special because the branch is *never* taken (this is functionally equivalent to a 2-byte, 3-cycle NOP). BIL and BIH are special because they use the state of the IRQ pin rather than the condition of a bit(s) in the CCR to decide whether to branch.

Table 6-7. Jump and Branch Instructions

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR	
						V 1 1 H	**I N Z C**

JMP *opr8a*
JMP *opr16a*
JMP *oprx16*,X
JMP *oprx8*,X
JMP ,X
	Jump
	PC ← Jump Address
		DIR
		EXT
		IX2
		IX1
		IX
			`BC dd`
			`CC hh ll`
			`DC ee ff`
			`EC ff`
			`FC`
				3
				4
				4
				3
				3
					`ppp`
					`pppp`
					`pppp`
					`ppp`
					`ppp`
						– 1 1 –	– – – –

BRA *rel*	Branch Always (if I = 1)	REL	`20 rr`	3	`ppp`	– 1 1 –	– – – –
BRN *rel*	Branch Never (if I = 0)	REL	`21 rr`	3	`ppp`	– 1 1 –	– – – –
BEQ *rel*	Branch if Equal (if Z = 1)	REL	`27 rr`	3	`ppp`	– 1 1 –	– – – –
BNE *rel*	Branch if Not Equal (if Z = 0)	REL	`26 rr`	3	`ppp`	– 1 1 –	– – – –
BCC *rel*	Branch if Carry Bit Clear (if C = 0)	REL	`24 rr`	3	`ppp`	– 1 1 –	– – – –
BCS *rel*	Branch if Carry Bit Set (if C = 1) (Same as BLO)	REL	`25 rr`	3	`ppp`	– 1 1 –	– – – –
BPL *rel*	Branch if Plus (if N = 0)	REL	`2A rr`	3	`ppp`	– 1 1 –	– – – –
BMI *rel*	Branch if Minus (if N = 1)	REL	`2B rr`	3	`ppp`	– 1 1 –	– – – –
BIL *rel*	Branch if IRQ Pin Low (if IRQ pin = 0)	REL	`2E rr`	3	`ppp`	– 1 1 –	– – – –
BIH *rel*	Branch if IRQ Pin High (if IRQ pin = 1)	REL	`2F rr`	3	`ppp`	– 1 1 –	– – – –
BMC *rel*	Branch if Interrupt Mask Clear (if I = 0)	REL	`2C rr`	3	`ppp`	– 1 1 –	– – – –
BMS *rel*	Branch if Interrupt Mask Set (if I = 1)	REL	`2D rr`	3	`ppp`	– 1 1 –	– – – –
BHCC *rel*	Branch if Half Carry Bit Clear (if H = 0)	REL	`28 rr`	3	`ppp`	– 1 1 –	– – – –
BHCS *rel*	Branch if Half Carry Bit Set (if H = 1)	REL	`29 rr`	3	`ppp`	– 1 1 –	– – – –
BLT *rel*	Branch if Less Than (if N ⊕ V = 1) (Signed)	REL	`91 rr`	3	`ppp`	– 1 1 –	– – – –
BLE *rel*	Branch if Less Than or Equal To (if Z I (N ⊕ V) = 1) (Signed)	REL	`93 rr`	3	`ppp`	– 1 1 –	– – – –
BGE *rel*	Branch if Greater Than or Equal To (if N ⊕ V = 0) (Signed)	REL	`90 rr`	3	`ppp`	– 1 1 –	– – – –

Table 6-7. Jump and Branch Instructions (Continued)

Source Form	Operation	Address Mode	Object Code	Cycles	Cyc-by-Cyc Details	Effect on CCR							
						V	1	1	H	I	N	Z	C
BGT *rel*	Branch if Greater Than (if Z I (N ⊕ V) = 0) (Signed)	REL	`92 rr`	3	`ppp`	–	1	1	–	–	–	–	–
BLO *rel*	Branch if Lower (if C = 1) (Same as BCS)	REL	`25 rr`	3	`ppp`	–	1	1	–	–	–	–	–
BLS *rel*	Branch if Lower or Same (if C I Z = 1)	REL	`23 rr`	3	`ppp`	–	1	1	–	–	–	–	–
BHS *rel*	Branch if Higher or Same (if C = 0) (Same as BCC)	REL	`24 rr`	3	`ppp`	–	1	1	–	–	–	–	–
BHI *rel*	Branch if Higher (if C I Z = 0)	REL	`22 rr`	3	`ppp`	–	1	1	–	–	–	–	–

6.5.5.1 Unconditional Jump and Branch

Jump (JMP), branch always (BRA), and branch never (BRN) are unconditional and do not depend on the state of any CCR bits. Jump may be used to go to any memory location in the 64-Kbyte address space while branch instructions are limited to destinations within –128 to +127 locations from the address immediately after the branch offset byte.

The following example illustrates the use of a JMP instruction to extend the range of a conditional branch. For every conditional branch instruction there is another branch that uses the opposite condition. For example the opposite of a branch if equal (BEQ) instruction is the branch if not equal (BNE) instruction. Suppose you wrote the instruction:

```
;            beq    farAway        ;more than 128 locs away
```

and the assembler flagged an error because farAway was more than 128 locations away. You can replace the BEQ with a BNE that branches around a jump instruction like this:

```
             bne    aroundJ        ;skip if NOT equal
             jmp    farAway        ;jump if equal
aroundJ:                           ;here if not equal
```

6.5.5.2 Simple Branches

The simple branches only depend on the state of a single condition (a CCR bit or the IRQ pin state).

Table 6-8. Simple Branch Summary

Branch Condition	Branch if True	Branch if False
Z	BEQ	BNE
C	BCS	BCC
N	BMI	BPL
IRQ pin	BIH	BIL
I	BMS	BMC
H	BHCS	BHCC