

# **Motorola 6809 and Hitachi 6309 Programmer's Reference**

## **A note about cycle counts**

The MPU cycle counts listed throughout this document will sometimes show two different values separated by a slash. In these cases the first value indicates the number of cycles used on a 6809 or a 6309 CPU running in Emulation mode. The second value indicates the number of cycles used on a 6309 CPU only when running in Native mode.

# **Part I**

## **Instruction Reference**

# ABX

## Add Accumulator B to Index Register X

$$X' \leftarrow X + \text{ACCB}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| ABX         | INHERENT        | 3A     | 3 / 1  | 1          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

The ABX instruction performs an unsigned addition of the contents of Accumulator B with the contents of Index Register X. The 16-bit result is placed into Index Register X. None of the Condition Code flags are affected.

The ABX instruction is similar in function to the LEAX B,X instruction. A significant difference is that LEAX B,X treats B as a twos complement value (signed), whereas ABX treats B as unsigned. For example, if X were to contain  $301B_{16}$  and B were to contain  $FF_{16}$ , then ABX would produce  $311A_{16}$  in X, whereas LEAX B,X would produce  $301A_{16}$  in X.

Additionally, the ABX instruction does not affect any flags in the Condition Codes register, whereas the LEAX instruction does affect the Zero flag.

One example of a situation where the ABX instruction may be used is when X contains the base address of a data structure or array and B contains an offset to a specific field or array element. In this scenario, ABX will modify X to point directly to the field or array element.

The ABX instruction was included in the 6x09 instruction set for compatibility with the 6801 microprocessor.

## ADC (8 Bit)

### Add Memory Byte plus Carry with Accumulator A or B

$$r' \leftarrow r + (M) + C$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ADCA            | 89        | 2 | 2 | 99     | 4 / 3 | 2 | A9      | 4+ | 2+ | B9       | 5 / 4 | 3 |
| ADCB            | C9        | 2 | 2 | D9     | 4 / 3 | 2 | E9      | 4+ | 2+ | F9       | 5 / 4 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   | ↑ |   | ↑ | ↑ | ↑ | ↑ |

These instructions add the contents of a byte in memory plus the contents of the Carry flag with either Accumulator A or B. The 8-bit result is placed back into the specified accumulator.

- H** The Half-Carry flag is set if a carry into bit 4 occurred; cleared otherwise.
- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of bit 7 occurred; cleared otherwise.

The ADC instruction is most often used to perform addition of the subsequent bytes of a multi-byte addition. This allows the carry from a previous ADD or ADC instruction to be included when doing addition for the next higher-order byte.

Since the 6x09 provides a 16-bit ADD instruction, it is not necessary to use the 8-bit ADD and ADC instructions for performing 16-bit addition.

See Also: **ADCD**, **ADCR**

# ADCD

6309 ONLY

## Add Memory Word plus Carry with Accumulator D

$$ACCD' \leftarrow ACCD + (M:M+1) + C$$

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| ADCD           | 1089      | 5 / 4 | 4 | 1099   | 7 / 5 | 3 | 10A9    | 7+ / 6+ | 3+ | 10B9     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

The ADCD instruction adds the contents of a double-byte value in memory plus the value of the Carry flag with Accumulator D. The 16 bit result is placed back into Accumulator D.

- H** The Half-Carry flag is not affected by the ADCD instruction.
- N** The Negative flag is set equal to the new value of bit 15 of the accumulator.
- Z** The Zero flag is set if the new Accumulator D value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of bit 15 occurred; cleared otherwise.

The ADCD instruction is most often used to perform addition of subsequent words of a multi-byte addition. This allows the carry from a previous ADD or ADC instruction to be included when doing addition for the next higher-order word.

The following instruction sequence is an example showing how 32-bit addition can be performed on a 6309 microprocessor:

```
LDQ    VAL1        ; Q = first 32-bit value
ADDW   VAL2+2      ; Add lower 16 bits of second value
ADCD   VAL2         ; Add upper 16 bits plus Carry
STQ    RESULT      ; Store 32-bit result
```

See Also: **ADC** (8-bit), **ADCR**

# ADCR

6309 ONLY

## Add Source Register plus Carry to Destination Register

$$r1' \leftarrow r1 + r0 + C$$

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| ADCR <i>r0,r1</i> | IMMEDIATE       | 1031   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

The ADCR instruction adds the contents of a source register plus the contents of the Carry flag with the contents of a destination register. The result is placed into the destination register.

- H** The Half-Carry flag is not affected by the ADCR instruction.
- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of the high-order bit occurred; cleared otherwise.

Any of the 6309 registers except Q and MD may be specified as the source operand, destination operand or both; however specifying the PC register as either the source or destination produces undefined results.

The ADCR instruction will perform either 8-bit or 16-bit addition according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See "6309 Inter-Register Operations" on page 143 for further details.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. See the description of the **TFR** instruction for further details.

See Also: **ADC** (8-bit), **ADCD**

# ADD (8 Bit)

## Add Memory Byte to 8-Bit Accumulator

$$r' \leftarrow r + (M)$$

| SOURCE FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ADDA         | 8B        | 2 | 2 | 9B     | 4 / 3 | 2 | AB      | 4+ | 2+ | BB       | 5 / 4 | 3 |
| ADDB         | CB        | 2 | 2 | DB     | 4 / 3 | 2 | EB      | 4+ | 2+ | FB       | 5 / 4 | 3 |
| ADDE         | 118B      | 3 | 3 | 119B   | 5 / 4 | 3 | 11AB    | 5+ | 3+ | 11BB     | 6 / 5 | 4 |
| ADDF         | 11CB      | 3 | 3 | 11DB   | 5 / 4 | 3 | 11EB    | 5+ | 3+ | 11FB     | 6 / 5 | 4 |

ADDE and ADDF are available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ↕ |   | ↕ | ↕ | ↕ | ↕ |

These instructions add the contents of a byte in memory with one of the 8-bit accumulators (A,B,E,F). The 8-bit result is placed back into the specified accumulator.

- H** The Half-Carry flag is set if a carry into bit 4 occurred; cleared otherwise.
- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of bit 7 occurred; cleared otherwise.

The 8-bit ADD instructions are used for single-byte addition, and for addition of the least-significant byte in multi-byte additions. Since the 6x09 also provides a 16-bit ADD instruction, it is not necessary to use the 8-bit ADD and ADC instructions for performing 16-bit addition.

See Also: **ADD** (16-bit), **ADDR**



# ADD (16 Bit)

## Add Memory Word to 16-Bit Accumulator

$$r' \leftarrow r + (M:M+1)$$

| SOURCE<br>FORMS | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|-----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                 | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| ADDD            | C3        | 4 / 3 | 3 | D3     | 6 / 4 | 2 | E3      | 6+ / 5+ | 2+ | F3       | 7 / 5 | 3 |
| ADDW            | 108B      | 5 / 4 | 4 | 109B   | 7 / 5 | 3 | 10AB    | 7+ / 6+ | 3+ | 10BB     | 8 / 6 | 4 |

ADDW is available on 6309 only.

| E | F | H | I | N  | Z  | V  | C  |
|---|---|---|---|----|----|----|----|
|   |   |   |   | ↑↓ | ↑↓ | ↑↓ | ↑↓ |

These instructions add the contents of a double-byte value in memory with one of the 16-bit accumulators (D,W). The 16-bit result is placed back into the specified accumulator.

- H** The Half-Carry flag is not affected by these instructions.
- N** The Negative flag is set equal to the new value of bit 15 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of bit 15 occurred; cleared otherwise.

The 16-bit ADD instructions are used for double-byte addition, and for addition of the least-significant word of multi-byte additions. See the description of the **ADCD** instruction for an example of how 32-bit addition can be performed on a 6309 processor.

See Also: **ADD** (8-bit), **ADDR**

# ADDR

6309 ONLY

## Add Source Register to Destination Register

$$r1' \leftarrow r1 + r0$$

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| ADDR <i>r0,r1</i> | IMMEDIATE       | 1030   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

The ADDR instruction adds the contents of a source register with the contents of a destination register. The result is placed into the destination register.

- H** The Half-Carry flag is not affected by the ADDR instruction.
- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a carry out of the high-order bit occurred; cleared otherwise.

Any of the 6309 registers except Q and MD may be specified as the source operand, destination operand or both; however specifying the PC register as either the source or destination produces undefined results.

The ADDR instruction will perform either 8-bit or 16-bit addition according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See “6309 Inter-Register Operations” on page 143 for further details.

A *Load Effective Address* instruction which adds one of the 16-bit accumulators to an index register (such as LEAX D,X) could be replaced by an ADDR instruction (ADDR D,X) in order to save 4 cycles (2 cycles in Native Mode). However, since more Condition Code flags are affected by the ADDR instruction, you should avoid this optimization if preservation of the affected flags is desired.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. See the description of the **TFR** instruction for further details.

See Also: **ADD** (8-bit), **ADD** (16-bit)

# AIM

6309 ONLY

## Logical AND of Immediate Value with Memory Byte

$M' \leftarrow (M) \text{ AND } \text{IMM}$

| SOURCE<br>FORM | IMMEDIATE |   |   | DIRECT |   |   | INDEXED |    |    | EXTENDED |   |   |
|----------------|-----------|---|---|--------|---|---|---------|----|----|----------|---|---|
|                | OP        | ~ | # | OP     | ~ | # | OP      | ~  | #  | OP       | ~ | # |
| AIM #i8;EA     |           |   |   | 02     | 6 | 3 | 62      | 7+ | 3+ | 72       | 7 | 4 |

E F H I N Z V C

|  |  |  |  |   |   |   |  |
|--|--|--|--|---|---|---|--|
|  |  |  |  | ↑ | ↑ | 0 |  |
|--|--|--|--|---|---|---|--|

The AIM instruction logically ANDs the contents of a byte in memory with an 8-bit immediate value. The resulting value is placed back into the designated memory location.

- N** The Negative flag is set equal to the new value of bit 7 of the memory byte.
- Z** The Zero flag is set if the new value of the memory byte is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

AIM is one of the more useful additions to the 6309 instruction set. It takes three separate instructions to perform the same operation on a 6809:

6809 (6 instruction bytes; 12 cycles):

LDA       #\$3F  
ANDA       4,U  
STA        4,U

6309 (3 instruction bytes; 8 cycles):

AIM       #\$3F;4,U

Note that the assembler syntax used for the AIM operand is non-typical. Some assemblers may require a comma (,) rather than a semicolon (;) between the immediate operand and the address operand.

The object code format for the AIM instruction is:

|        |             |                         |
|--------|-------------|-------------------------|
| OPCODE | IMMED VALUE | ADDRESS / INDEX BYTE(S) |
|--------|-------------|-------------------------|

See Also: **AND**, **EIM**, **OIM**, **TIM**

## AND (8 Bit)

### Logically AND Memory Byte with Accumulator A or B

$$r' \leftarrow r \text{ AND } (M)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ANDA            | 84        | 2 | 2 | 94     | 4 / 3 | 2 | A4      | 4+ | 2+ | B4       | 5 / 4 | 3 |
| ANDB            | C4        | 2 | 2 | D4     | 4 / 3 | 2 | E4      | 4+ | 2+ | F4       | 5 / 4 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions logically AND the contents of a byte in memory with either Accumulator A or B. The 8-bit result is then placed in the specified accumulator.

**N** The Negative flag is set equal to the new value of bit 7 of the accumulator.

**Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.

**V** The Overflow flag is cleared by this instruction.

**C** The Carry flag is not affected by this instruction.

The AND instructions are commonly used for clearing bits and for testing bits. Consider the following examples:

```
ANDA    #%11101111    ;Clears bit 4 in A
ANDA    #%00000100    ;Sets Z flag if bit 2 is not set
```

When testing bits, it is often preferable to use the BIT instructions instead, since they perform the same logical AND operation without modifying the contents of the accumulator.

See Also: **AIM**, **ANDCC**, **ANDD**, **ANDR**, **BAND**, **BIAND**, **BIT**

# ANDCC

## Logically AND Immediate Value with the CC Register

$CC' \leftarrow CC \text{ AND } IMM$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| ANDCC #i8   | IMMEDIATE       | 1C     | 3      | 2          |

This instruction logically ANDs the contents of the Condition Codes register with the immediate byte specified in the instruction. The result is placed back into the Condition Codes register.

The ANDCC instruction provides a method to clear specific flags in the Condition Codes register. All flags that correspond to "0" bits in the immediate operand are cleared, while those corresponding with "1"s are left unchanged.

The bit numbers for each flag are shown below:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| E | F | H | I | N | Z | V | C |

One of the more common uses for the ANDCC instruction is to clear the IRQ and FIRQ Interrupt Masks (I and F) at the completion of a routine that runs with interrupts disabled. This is accomplished by executing:

```
ANDCC    #$AF        ; Clear bits 4 and 6 in CC
```

Some assemblers will accept a comma-delimited list of the bit names to be cleared as an alternative to the immediate expression. For instance, the example above might also be written as:

```
ANDCC    I,F          ; Clear bits 4 and 6 in CC
```

This syntax is generally discouraged due to the confusion it can create as to whether it means clear the I and F bits, or clear all bits except I and F.

More examples:

```
ANDCC    #$FE        ; Clear the Carry flag
ANDCC    #1           ; Clear all flags except Carry
```

See Also: **AND** (8-bit), **ANDD**, **ANDR**, **CWAI**, **ORCC**

# ANDD

6309 ONLY

## Logically AND Memory Word with Accumulator D

$ACCD' \leftarrow ACCD \text{ AND } (M:M+1)$

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| ANDD           | 1084      | 5 / 4 | 4 | 1094   | 7 / 5 | 3 | 10A4    | 7+ / 6+ | 3+ | 10B4     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↓ | ↓ | 0 |   |

The ANDD instruction logically ANDs the contents of a double-byte value in memory with the contents of Accumulator D. The 16-bit result is placed back into Accumulator D.

- N** The Negative flag is set equal to the new value of bit 15 of Accumulator D.
- Z** The Zero flag is set if the new value of the Accumulator D is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

One use for the ANDD instruction is to truncate bits of an address value. For example:

```
ANDD    #$E000    ;Convert address to that of its 8K page
```

For testing bits, it is often preferable to use the **BITD** instruction instead, since it performs the same logical AND operation without modifying the contents of Accumulator D.

See Also: **AND** (8-bit), **ANDCC**, **ANDR**, **BITD**

# ANDR

6309 ONLY

## Logically AND Source Register with Destination Register

$r1' \leftarrow r1 \text{ AND } r0$

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| ANDR <i>r0,r1</i> | IMMEDIATE       | 1034   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The ANDR instruction logically ANDs the contents of a source register with the contents of a destination register. The result is placed into the destination register.

- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

Any of the 6309 registers except Q and MD may be specified as the source operand, destination operand or both; however specifying the PC register as either the source or destination produces undefined results.

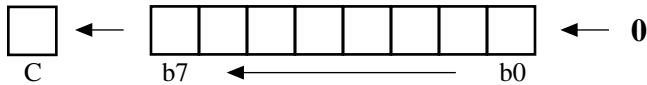
The ANDR instruction will perform either an 8-bit or 16-bit operation according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See "6309 Inter-Register Operations" on page 143 for further details.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. For details, see the description of the **TFR** instruction.

See Also: **AND** (8-bit), **ANDCC**, **ANDD**

# ASL (8 Bit)

## Arithmetic Shift Left of 8-Bit Accumulator or Memory Byte



| SOURCE FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ASLA         | 48       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ASLB         | 58       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ASL          |          |       |   | 08     | 6 / 5 | 2 | 68      | 6+ | 2+ | 78       | 7 / 6 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ~ |   | ↕ | ↕ | ↕ | ↕ |

These instructions shift the contents of the A or B accumulator or a specified byte in memory to the left by one bit, clearing bit 0. Bit 7 is shifted into the Carry flag of the Condition Codes register.

- H** The affect on the Half-Carry flag is undefined for these instructions.
- N** The Negative flag is set equal to the new value of bit 7; previously bit 6.
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is set to the Exclusive-OR of the original values of bits 6 and 7.
- C** The Carry flag receives the value shifted out of bit 7.

The ASL instruction can be used for simple multiplication (a single left-shift multiplies the value by 2). Other uses include conversion of data from serial to parallel and vise-versa.

The 6309 does not provide variants of ASL to operate on the E and F accumulators. However, you can achieve the same functionality using the ADDR instruction. The instructions ADDR E,E and ADDR F,F will perform the same left-shift operation on the E and F accumulators respectively.

The ASL and LSL mnemonics are duplicates. Both produce the same object code.

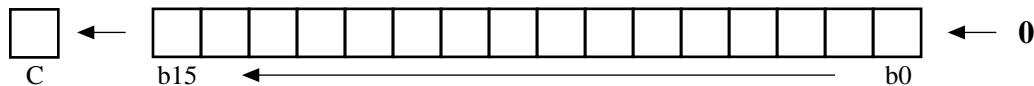
See Also: **ASLD**



# ASLD

6309 ONLY

## Arithmetic Shift Left of Accumulator D



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| ASLD        | INHERENT        | 1048   | 3 / 2  | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

This instruction shifts the contents of Accumulator D to the left by one bit, clearing bit 0. Bit 15 is shifted into the Carry flag of the Condition Codes register.

- N** The Negative flag is set equal to the new value of bit 15; previously bit 14.
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is set to the Exclusive-OR of the original values of bits 14 and 15.
- C** The Carry flag receives the value shifted out of bit 15.

The ASL instruction can be used for simple multiplication (a single left-shift multiplies the value by 2). Other uses include conversion of data from serial to parallel and vice-versa.

The D accumulator is the only 16-bit register for which an ASL instruction has been provided. You can however achieve the same functionality using the ADDR instruction. For example, ADDR W,W will perform the same left-shift operation on the W accumulator.

A left-shift of the 32-bit Q accumulator can be achieved as follows:

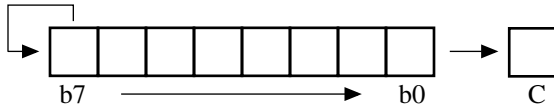
```
ADDR    W,W      ; Shift Low-word, Hi-bit into Carry
ROL     W,W      ; Shift Hi-word, Carry into Low-bit
```

The ASLD and LSLD mnemonics are duplicates. Both produce the same object code.

See Also: **ASL** (8-bit), **ROL** (16-bit)

# ASR (8 Bit)

## Arithmetic Shift Right of 8-Bit Accumulator or Memory Byte



| SOURCE FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ASRA         | 47       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ASRB         | 57       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ASR          |          |       |   | 07     | 6 / 5 | 2 | 67      | 6+ | 2+ | 77       | 7 / 6 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ~ |   | ↑ | ↑ |   | ↑ |

These instructions arithmetically shift the contents of the A or B accumulator or a specified byte in memory to the right by one bit. Bit 0 is shifted into the Carry flag of the Condition Codes register. The value of bit 7 is not changed.

- H** The affect on the Half-Carry flag is undefined for these instructions.
- N** The Negative flag is set equal to the value of bit 7.
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by these instructions.
- C** The Carry flag receives the value shifted out of bit 0.

The ASR instruction can be used in simple division routines (a single right-shift divides the value by 2). Be careful here, as a right-shift is not the same as a division when the value is negative; it rounds in the wrong direction. For example, -5 ( $FB_{16}$ ) divided by 2 should be -2 but, when arithmetically shifted right, is -3 ( $FD_{16}$ ).

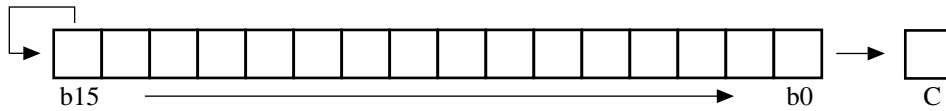
The 6309 does not provide variants of ASR to operate on the E and F accumulators.

See Also: **ASRD**

# ASRD

6309 ONLY

## Arithmetic Shift Right of Accumulator D



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| ASRD        | INHERENT        | 1047   | 3 / 2  | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ |   | ↑ |

This instruction shifts the contents of Accumulator D to the right by one bit. Bit 0 is shifted into the Carry flag of the Condition Codes register. The value of bit 15 is not changed.

- N** The Negative flag is set equal to the value of bit 15.
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by this instruction.
- C** The Carry flag receives the value shifted out of bit 0.

The ASRD instruction can be used in simple division routines (a single right-shift divides the value by 2). Be careful here, as a right-shift is not the same as a division when the value is negative; it rounds in the wrong direction. For example, -5 (FFFB<sub>16</sub>) divided by 2 should be -2 but, when arithmetically shifted right, is -3 (FFFD<sub>16</sub>).

The 6309 does not provide a variant of ASR to operate on the W accumulator, although it does provide the LSRW instruction for performing a logical shift.

An arithmetic right-shift of the 32-bit Q accumulator can be achieved as follows:

```
ASRD      ; Shift Hi-word, Low-bit into Carry
RORW      ; Shift Low-word, Carry into Hi-bit
```

See Also: **ASR** (8-bit), **ROR** (16-bit)

# BAND

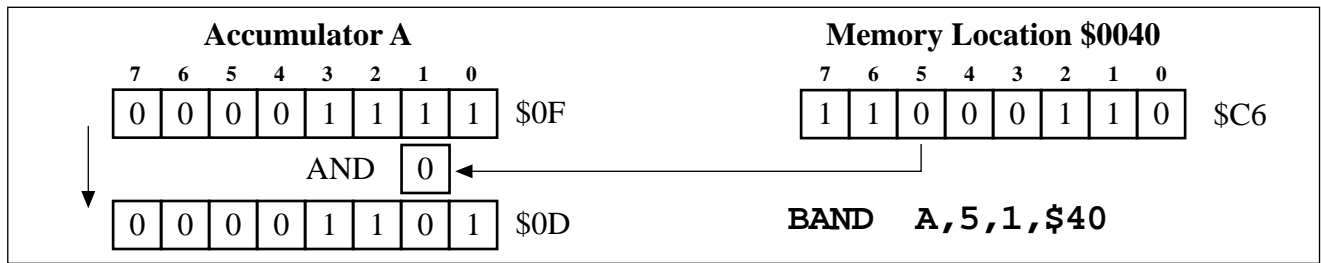
6309 ONLY

## Logically AND Register Bit with Memory Bit

$r.dstBit' \leftarrow r.dstBit \text{ AND } (DPM).srcBit$

| SOURCE FORM                  | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------------------|-----------------|--------|--------|------------|
| BAND <i>r,sBit,dBit,addr</i> | DIRECT          | 1130   | 7 / 6  | 4          |

The BAND instruction logically ANDs the value of a specified bit in either the A, B or CC registers with a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected. The usefulness of the BAND instruction is limited by the fact that only Direct Addressing is permitted.



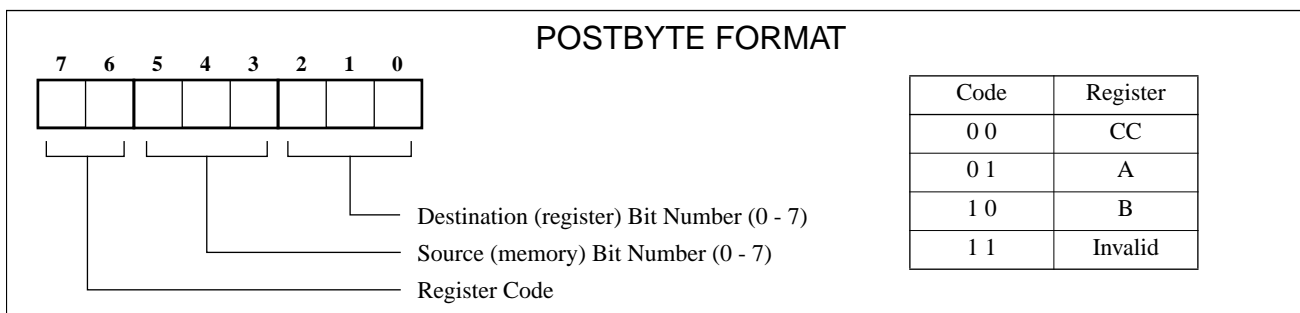
The figure above shows an example of the BAND instruction where bit 1 of Accumulator A is ANDed with bit 5 of the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BAND instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$30 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|



See Also: **BEOR, BLAND, BIEOR, BIOR, BOR, LDBT, STBT**

# BCC

## Branch If Carry Clear

IF  $CC.C = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| BCC address | RELATIVE        | 24     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is clear (0), causes a relative branch. If the Carry flag is 1, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BCC instruction will branch if the source value was higher than or the same as the original destination value. For this reason, 6809/6309 assemblers will accept BHS as an alternate mnemonic for BCC.

BCC is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag. BCC will always branch following a CLR instruction and will never branch following a COM instruction due to the way those instructions affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the BCC instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BCC instruction. If a larger range is required then the LBCC instruction may be used instead.

See Also: **BCS**, **BGE**, **LBCC**

# BCS

## Branch If Carry Set

IF  $CC.C \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BCS <i>address</i> | RELATIVE        | 25     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is set (1), causes a relative branch. If the Carry flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BCS instruction will branch if the source value was lower than the original destination value. For this reason, 6809/6309 assemblers will accept BLO as an alternate mnemonic for BCS.

BCS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag. BCS will never branch following a CLR instruction and will always branch following a COM instruction due to the way those instructions affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the BCS instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BCS instruction. If a larger range is required then the **LBCS** instruction may be used instead.

See Also: **BCC**, **BLT**, **LBCS**

# BEOR

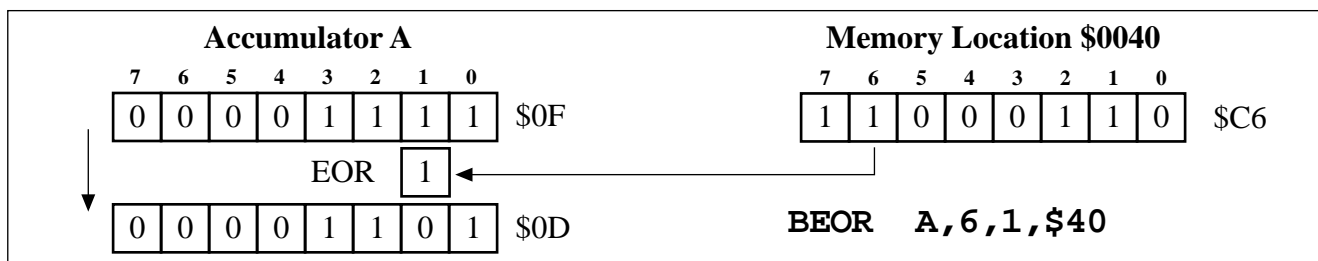
6309 ONLY

## Exclusive-OR Register Bit with Memory Bit

$$r.dstBit' \leftarrow r.dstBit \oplus (DPM).srcBit$$

| SOURCE FORM                  | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------------------|-----------------|--------|--------|------------|
| BEOR <i>r,sBit,dBit,addr</i> | DIRECT          | 1134   | 7 / 6  | 4          |

The BEOR instruction Exclusively ORs the value of a specified bit in either the A, B or CC registers with a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected. The usefulness of the BEOR instruction is limited by the fact that only Direct Addressing is permitted.



The figure above shows an example of the BEOR instruction where bit 1 of Accumulator A is Exclusively ORed with bit 6 of the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BEOR instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$34 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|

See the description of the **BAND** instruction on page 20 for details about the postbyte format used by this instruction.

See Also: **BAND**, **BIAND**, **BIEOR**, **BIOR**, **BOR**, **LDBT**, **STBT**

# BEQ

## Branch If Equal to Zero

IF  $CC.Z \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BEQ <i>address</i> | RELATIVE        | 27     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero flag in the CC register and, if it is set (1), causes a relative branch. If the Z flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following almost any instruction that produces, tests or moves a value, the BEQ instruction will branch if that value is equal to zero. In the case of an instruction that performs a subtract or compare, the BEQ instruction will branch if the source value was equal to the original destination value.

BEQ is generally not useful following a CLR instruction since the Z flag is always set.

The following instructions produce or move values, but do not affect the Z flag:

|      |      |      |       |       |
|------|------|------|-------|-------|
| ABX  | BAND | BEOR | BIAND | BIEOR |
| BOR  | BIOR | EXG  | LDBT  | LDMD  |
| LEAS | LEAU | PSH  | PUL   | STBT  |
| TFM  | TFR  |      |       |       |

The branch address is calculated by adding the current value of the PC register (after the BEQ instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BEQ instruction. If a larger range is required then the **LBEQ** instruction may be used instead.

See Also: **BNE**, **LBEQ**



# BGE

## Branch If Greater than or Equal to Zero

IF  $CC.N = CC.V$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BGE <i>address</i> | RELATIVE        | 2C     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) and Overflow (V) flags in the CC register and, if both are set OR both are clear, causes a relative branch. If the N and V flags do not have the same value then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the BGE instruction will branch if the source value was greater than or equal to the original destination value.

The branch address is calculated by adding the current value of the PC register (after the BGE instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BGE instruction. If a larger range is required then the **LBGE** instruction may be used instead.

See Also: **BHS**, **BLT**, **LBGE**

# BGT

## Branch If Greater Than Zero

IF (CC.N = CC.V) AND (CC.Z = 0) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BGT <i>address</i> | RELATIVE        | 2E     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) flag in the CC register and, if it is clear AND the values of the Negative (N) and Overflow (V) flags are equal (both set OR both clear), causes a relative branch. If the N and V flags do not have the same value or if the Z flag is set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the BGT instruction will branch if the source value was greater than the original destination value.

The branch address is calculated by adding the current value of the PC register (after the BGT instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BGT instruction. If a larger range is required then the **LBGT** instruction may be used instead.

See Also: **BHI**, **BLE**, **LBGT**

# BHI

## Branch If Higher

IF (CC.Z = 0) AND (CC.C = 0) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BHI <i>address</i> | RELATIVE        | 22     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) and Carry (C) flags in the CC register and, if both are zero, causes a relative branch. If either the Z or C flags are set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BHI instruction will branch if the source value was higher than the original destination value.

BHI is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the BHI instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BHI instruction. If a larger range is required then the **LBHI** instruction may be used instead.

See Also: **BGT**, **BLS**, **LBHI**

# BHS

## Branch If Higher or Same

IF  $CC.C = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BHS <i>address</i> | RELATIVE        | 24     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is clear (0), causes a relative branch. If the Carry flag is 1, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BHS instruction will branch if the source value was higher or the same as the original destination value.

BHS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

BHS is an alternate mnemonic for the BCC instruction. Both produce the same object code.

The branch address is calculated by adding the current value of the PC register (after the BHS instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BHS instruction. If a larger range is required then the **LBHS** instruction may be used instead.

See Also: **BGE**, **BLO**, **LBHS**

# BIAND

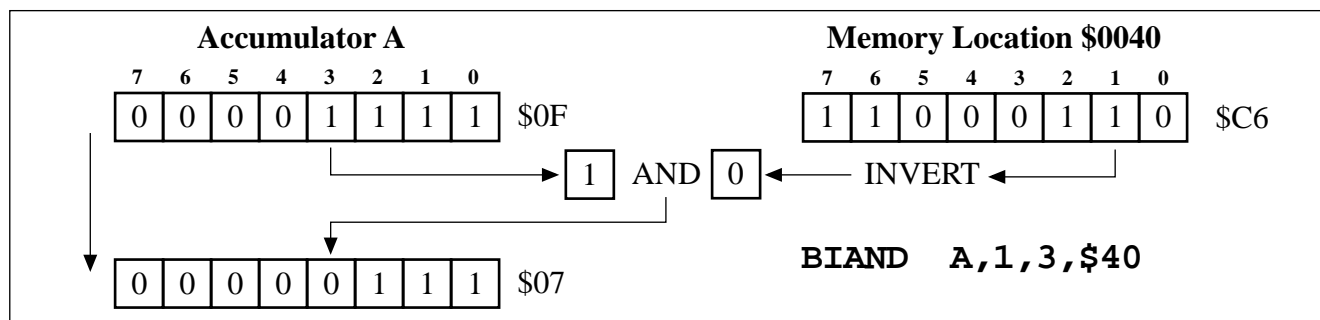
6309 ONLY

## Logically AND Register Bit with Inverted Memory Bit

$$r.dstBit' \leftarrow r.dstBit \text{ AND } \overline{(DPM).srcBit}$$

| SOURCE FORM                   | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------------------|-----------------|--------|--------|------------|
| BIAND <i>r,sBit,dBit,addr</i> | DIRECT          | 1131   | 7 / 6  | 4          |

The BIAND instruction logically ANDs the value of a specified bit in either the A, B or CC registers with the inverted value of a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected. The usefulness of the BIAND instruction is limited by the fact that only Direct Addressing is permitted.



The figure above shows an example of the BIAND instruction where bit 3 of Accumulator A is ANDed with the inverted value of bit 1 from the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BIAND instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$31 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|

See the description of the **BAND** instruction on page 20 for details about the postbyte format used by this instruction.

See Also: **BAND**, **BEOR**, **BIEOR**, **BIOR**, **BOR**, **LDBT**, **STBT**

# BIEOR

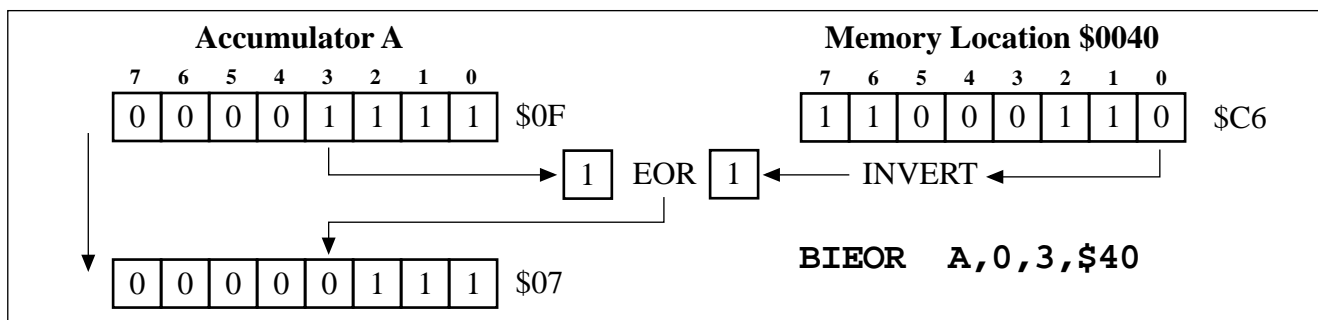
6309 ONLY

## Exclusively-OR Register Bit with Inverted Memory Bit

$$r.dstBit' \leftarrow r.dstBit \oplus \overline{(DPM).srcBit}$$

| SOURCE FORM                   | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------------------|-----------------|--------|--------|------------|
| BIEOR <i>r,sBit,dBit,addr</i> | DIRECT          | 1135   | 7 / 6  | 4          |

The BIEOR instruction exclusively ORs the value of a specified bit in either the A, B or CC registers with the inverted value of a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected. The usefulness of the BIEOR instruction is limited by the fact that only Direct Addressing is permitted.



The figure above shows an example of the BIEOR instruction where bit 3 of Accumulator A is Exclusively ORed with the inverted value of bit 0 from the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BIEOR instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$35 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|

See the description of the **BAND** instruction on page 20 for details about the Postbyte format used by this instruction.

See Also: **BAND**, **BEOR**, **BIAND**, **BIOR**, **BOR**, **LDBT**, **STBT**

# BIOR

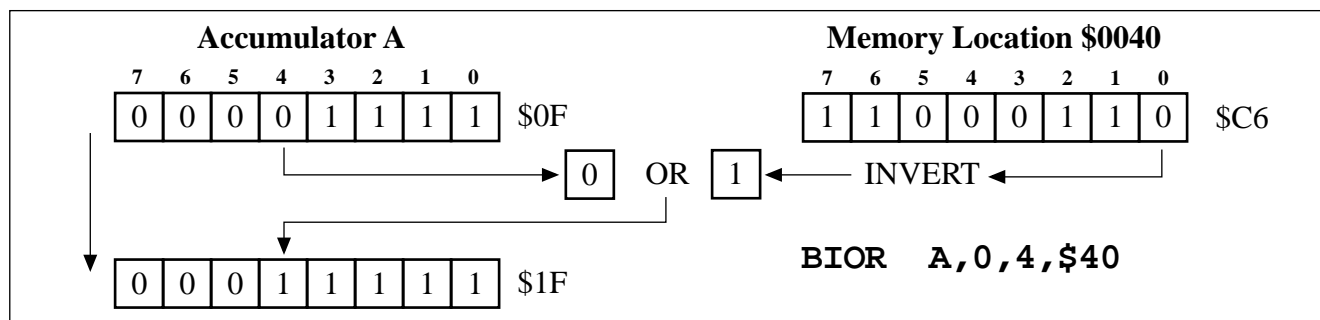
6309 ONLY

## Logically OR Register Bit with Inverted Memory Bit

$$r.dstBit' \leftarrow r.dstBit \text{ OR } \overline{(DPM).srcBit}$$

| SOURCE FORM                  | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------------------|-----------------|--------|--------|------------|
| BIOR <i>r,sBit,dBit,addr</i> | DIRECT          | 1133   | 7 / 6  | 4          |

The BIOR instruction ORs the value of a specified bit in either the A, B or CC registers with the inverted value of a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected. The usefulness of the BIOR instruction is limited by the fact that only Direct Addressing is permitted.



The figure above shows an example of the BIOR instruction where bit 4 of Accumulator A is logically ORed with the inverted value of bit 0 from the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BIOR instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$33 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|

See the description of the **BAND** instruction on page 20 for details about the Postbyte format used by this instruction.

See Also: **BAND**, **BEOR**, **BIAND**, **BIEOR**, **BOR**, **LDBT**, **STBT**

## BIT (8 Bit)

### Bit Test Accumulator A or B with Memory Byte Value

TEMP  $\leftarrow$  r AND (M)

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| BITA            | 85        | 2 | 2 | 95     | 4 / 3 | 2 | A5      | 4+ | 2+ | B5       | 5 / 4 | 3 |
| BITB            | C5        | 2 | 2 | D5     | 4 / 3 | 2 | E5      | 4+ | 2+ | F5       | 5 / 4 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions logically AND the contents of a byte in memory with either Accumulator A or B. The 8-bit result is tested to set or clear the appropriate flags in the CC register. Neither the accumulator nor the memory byte are modified.

**N** The Negative flag is set equal to bit 7 of the resulting value.

**Z** The Zero flag is set if the resulting value was zero; cleared otherwise.

**V** The Overflow flag is cleared by this instruction.

**C** The Carry flag is not affected by this instruction.

The BIT instructions are used for testing bits. Consider the following example:

```
ANDA    %#00000100    ;Sets Z flag if bit 2 is not set
```

BIT instructions differ from AND instructions only in that they do not modify the specified accumulator.

See Also: **AND** (8-bit), **BITD**, **BITMD**



# BITD

6309 ONLY

## Bit Test Accumulator D with Memory Word Value

TEMP  $\leftarrow$  ACCD AND (M:M+1)

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| BITD           | 1085      | 5 / 4 | 4 | 1095   | 7 / 5 | 3 | 10A5    | 7+ / 6+ | 3+ | 10B5     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The BITD instruction logically ANDs the contents of a double-byte value in memory with the contents of Accumulator D. The 16-bit result is tested to set or clear the appropriate flags in the CC register. Neither Accumulator D nor the memory bytes are modified.

- N** The Negative flag is set equal to bit 15 of the resulting value.
- Z** The Zero flag is set if the resulting value was zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

The BITD instruction differs from ANDD only in that Accumulator D is not modified.

See Also: **ANDD**, **BIT** (8-bit), **BITMD**

# BITMD

6309 ONLY

## Bit Test the MD Register with an Immediate Value

$CC.Z \leftarrow (MD.IL \text{ AND } IMM.6 = 0) \text{ AND } (MD./0 \text{ AND } IMM.7 = 0)$   
 $MD.IL' \leftarrow MD.IL \text{ AND } \overline{IMM.6}$   
 $MD./0' \leftarrow MD./0 \text{ AND } \overline{IMM.7}$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| BITMD #i8   | IMMEDIATE       | 113C   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   | ↑ |   |   |

This instruction logically ANDs the two most-significant bits of the MD register (the *Divide-by-Zero* and *Illegal Instruction* status bits) with the two most-significant bits of the immediate operand. The Z flag in the CC register is set if the AND operation produces a zero result, otherwise Z is cleared. No other condition code flags are affected. The BITMD instruction also clears those status bits in the MD register which correspond to '1' bits in the immediate operand. The values of bits 0 through 5 in the immediate operand have no relevance and do not affect the operation of the BITMD instruction in any way.

The BITMD instruction provides a method to test the *Divide-by-Zero* (/0) and *Illegal Instruction* (IL) status bits of the MD register after an Illegal Instruction Exception has occurred. At most, only one of these flags will be set, indicating which condition caused the exception. Since the status bit(s) tested are also cleared by this instruction, you can only test for each condition once.

Bits 0 through 5 of the MD register are neither tested nor cleared by this instruction. Therefore BITMD cannot be used to determine or change the current execution mode of the CPU. See “Determining the 6309 Execution Mode” on page 144 for more information on this topic.

The figure below shows the layout of the MD register:

|    |    |   |   |   |   |    |    |
|----|----|---|---|---|---|----|----|
| 7  | 6  | 5 | 4 | 3 | 2 | 1  | 0  |
| /0 | IL |   |   |   |   | FM | NM |

See Also: **LDMD**

# BLE

## Branch If Less than or Equal to Zero

IF (CC.N  $\neq$  CC.V) OR (CC.Z = 1) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BLE <i>address</i> | RELATIVE        | 2F     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction performs a relative branch if the value of the Zero (Z) flag is 1, OR if the values of the Negative (N) and Overflow (V) flags are not equal. If the N and V flags have the same value and the Z flag is not set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the BLE instruction will branch if the source value was less than or equal to the original destination value.

The branch address is calculated by adding the current value of the PC register (after the BLE instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BLE instruction. If a larger range is required then the **LBLE** instruction may be used instead.

See Also: **BGT**, **BLS**, **LBLE**

# BLO

## Branch If Lower

IF  $CC.C \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BLO <i>address</i> | RELATIVE        | 25     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is set (1), causes a relative branch. If the Carry flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BLO instruction will branch if the source value was lower than the original destination value.

BLO is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

BLO is an alternate mnemonic for the BCS instruction. Both produce the same object code.

The branch address is calculated by adding the current value of the PC register (after the BLO instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BLO instruction. If a larger range is required then the **LBLO** instruction may be used instead.

See Also: **BHS**, **BLT**, **LBLO**

# BLS

## Branch If Lower or Same

IF (CC.Z  $\neq$  0) OR (CC.C  $\neq$  0) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BLS <i>address</i> | RELATIVE        | 23     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) and Carry (C) flags in the CC register and, if either are set, causes a relative branch. If both the Z and C flags are clear then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the BLS instruction will branch if the source value was lower than or the same as the original destination value.

BLS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the BLS instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BLS instruction. If a larger range is required then the **LBLS** instruction may be used instead.

See Also: **BHI**, **BLE**, **LBLS**

# BLT

## Branch If Less Than Zero

IF  $CC.N \neq CC.V$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BLT <i>address</i> | RELATIVE        | 2D     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction performs a relative branch if the values of the Negative (N) and Overflow (V) flags are not equal. If the N and V flags have the same value then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the BLT instruction will branch if the source value was less than the original destination value.

The branch address is calculated by adding the current value of the PC register (after the BLT instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BLT instruction. If a larger range is required then the **LBLT** instruction may be used instead.

See Also: **BGE**, **BLO**, **LBLT**

# BMI

## Branch If Minus

IF  $CC.N \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BMI <i>address</i> | RELATIVE        | 2B     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) flag in the CC register and, if it is set (1), causes a relative branch. If the N flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the BMI instruction will branch if the resulting value is negative. It is generally preferable to use the BLT instruction following such an operation because the sign bit may be invalid due to a twos-complement overflow.

The branch address is calculated by adding the current value of the PC register (after the BMI instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BMI instruction. If a larger range is required then the **LBMI** instruction may be used instead.

See Also: **BLT**, **BPL**, **LBMI**

# BNE

## Branch If Not Equal to Zero

IF  $CC.Z = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BNE <i>address</i> | RELATIVE        | 26     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero flag in the CC register and, if it is clear (0), causes a relative branch. If the Z flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following almost any instruction that produces, tests or moves a value, the BNE instruction will branch if that value is not equal to zero. In the case of an instruction that performs a subtract or compare, the BNE instruction will branch if the source value was not equal to the original destination value.

BNE is generally not useful following a CLR instruction since the Z flag is always set.

The following instructions produce or move values, but do not affect the Z flag:

|      |      |      |       |       |
|------|------|------|-------|-------|
| ABX  | BAND | BEOR | BIAND | BIEOR |
| BOR  | BIOR | EXG  | LDBT  | LDMD  |
| LEAS | LEAU | PSH  | PUL   | STBT  |
| TFM  | TFR  |      |       |       |

The branch address is calculated by adding the current value of the PC register (after the BNE instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BNE instruction. If a larger range is required then the **LBNE** instruction may be used instead.

See Also: **BEQ**, **LBNE**



# BOR

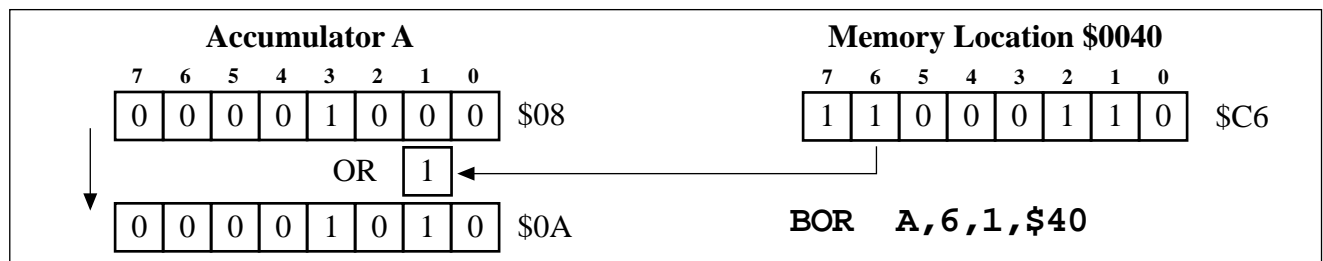
6309 ONLY

## Logically OR Memory Bit with Register Bit

$r.dstBit' \leftarrow r.dstBit \text{ OR } (DPM).srcBit$

| SOURCE FORM                 | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-----------------------------|-----------------|--------|--------|------------|
| BOR <i>r,sBit,dBit,addr</i> | DIRECT          | 1132   | 7 / 6  | 4          |

The BOR instruction logically ORs the value of a specified bit in either the A, B or CC registers with a specified bit in memory. The resulting value is placed back into the register bit. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit may be affected.



The figure above shows an example of the BOR instruction where bit 1 of Accumulator A is ORed with bit 6 of the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

The usefulness of the BOR instruction is limited by the fact that only Direct Addressing is permitted. Since the Condition Code flags are not affected by the operation, additional instructions would be needed to test the result for conditional branching.

The object code format for the BOR instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$32 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|

See the description of the **BAND** instruction on page 20 for details about the postbyte format used by this instruction.

See Also: **BAND**, **BEOR**, **BIAND**, **BIEOR**, **BIOR**, **LDBT**, **STBT**

# BPL

## Branch If Plus

IF  $CC.N = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BPL <i>address</i> | RELATIVE        | 2A     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) flag in the CC register and, if it is clear (0), causes a relative branch. If the N flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the BPL instruction will branch if the resulting value is positive. It is generally preferable to use the BGE instruction following such an operation because the sign bit may be invalid due to a twos-complement overflow.

The branch address is calculated by adding the current value of the PC register (after the BPL instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BPL instruction. If a larger range is required then the **LBPL** instruction may be used instead.

See Also: **BGE**, **BMI**, **LBPL**

# BRA

## Branch Always

$$PC' \leftarrow PC + IMM$$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BRA <i>address</i> | RELATIVE        | 20     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction causes an unconditional relative branch. None of the Condition Code flags are affected.

The BRA instruction is similar in function to the JMP instruction in that it always causes execution to be transferred to the effective address specified by the operand. The primary difference is that BRA uses the Relative Addressing mode which allows the code to be position-independent.

The branch address is calculated by adding the current value of the PC register (after the BRA instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BPL instruction. If a larger range is required then the **LBRA** instruction may be used instead.

See Also: **BRN**, **JMP**, **LBRA**

# BRN

## Branch Never

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BRN <i>address</i> | RELATIVE        | 21     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction is essentially a no-operation; that is, the CPU never branches but merely advances to the next instruction in sequence. No Condition Code flags are affected. BRN is effectively the equivalent of `BRA *+2`

The BRN instruction provides a 2-byte no-op that consumes 3 bus cycles, whereas NOP is a single-byte instruction that consumes either 1 or 2 bus cycles. In addition, there is the LBRN instruction which provides a 4-byte no-op that consumes 5 bus cycles.

Since the branch is never taken, the second byte of the instruction does not serve any purpose and may contain any value. This permits an optimization technique in which a BRN opcode can be used to skip over some other single byte instruction. In this technique, the second byte of the BRN instruction contains the opcode of the instruction which is to be skipped. The two code examples shown below both perform identically. The difference is that Example 2 uses a BRN opcode to reduce the code size by one byte.

Example 1 - conventional:

```
                CMPA    #$40
                BLO     @1
                SUBA    #$20
                BRA     @2      ; SKIP NEXT INSTRUCTION
@1              CLRA
@2              STA     RESULT
```

Example 2 - use BRN opcode (\$21) to reduce code size:

```
                CMPA    #$40
                BLO     @1
                SUBA    #$20
                FCB     $21    ; SKIP NEXT INSTRUCTION
@1              CLRA
                STA     RESULT
```

See Also: **BRA**, **NOP**, **LBRN**

# BSR

## Branch to Subroutine

$$\begin{aligned} S' &\leftarrow S - 2 \\ (S : S+1) &\leftarrow PC \\ PC' &\leftarrow PC + IMM \end{aligned}$$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BSR <i>address</i> | RELATIVE        | 8D     | 7 / 6  | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction pushes the value of the PC register (after the BSR instruction bytes have been fetched) onto the hardware stack and then performs an unconditional relative branch. None of the Condition Code flags are affected.

By pushing the PC value onto the stack, the called subroutine can "return" to this address after it has completed.

The BSR instruction is similar in function to the JSR instruction. The significant difference is that BSR uses the Relative Addressing mode which implies that both the BSR instruction and the called subroutine may be contained in relocatable code, so long as both are contained in the same module.

The branch address is calculated by adding the current value of the PC register (after the BSR instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BSR instruction. If a larger range is required then the **LBSR** instruction may be used instead.

See Also: **JSR**, **LBSR**, **RTS**

# BVC

## Branch If Overflow Clear

IF  $CC.V = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BVC <i>address</i> | RELATIVE        | 28     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Overflow (V) flag in the CC register and, if it is clear (0), causes a relative branch. If the V flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the BVC instruction will branch if there was no overflow.

The branch address is calculated by adding the current value of the PC register (after the BVC instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BVC instruction. If a larger range is required then the **LBVC** instruction may be used instead.

See Also: **BVS**, **LBVC**

# BVS

## Branch If Overflow Set

IF  $CC.V \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|--------------------|-----------------|--------|--------|------------|
| BVS <i>address</i> | RELATIVE        | 29     | 3      | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Overflow (V) flag in the CC register and, if it is set (1), causes a relative branch. If the V flag is clear, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the BVS instruction will branch if an overflow occurred.

The branch address is calculated by adding the current value of the PC register (after the BVS instruction bytes have been fetched) with the 8-bit twos-complement value contained in the second byte of the instruction. The range of the branch destination is limited to -126 to +129 bytes from the address of the BVS instruction. If a larger range is required then the **LBVS** instruction may be used instead.

See Also: **BVC**, **LBVS**

# CLR (accumulator)

## Load Zero into Accumulator

$r \leftarrow 0$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| CLRA        | INHERENT        | 4F     | 2 / 1  | 1          |
| CLRB        | INHERENT        | 5F     | 2 / 1  | 1          |
| CLRD        | INHERENT        | 104F   | 3 / 2  | 2          |
| CLRE        | INHERENT        | 114F   | 3 / 2  | 2          |
| CLRF        | INHERENT        | 115F   | 3 / 2  | 2          |
| CLRW        | INHERENT        | 105F   | 3 / 2  | 2          |

CLRD, CLRE, CLRF and CLRW are available on 6309 only.

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>E</b> | <b>F</b> | <b>H</b> | <b>I</b> | <b>N</b> | <b>Z</b> | <b>V</b> | <b>C</b> |
|          |          |          |          | 0        | 1        | 0        | 0        |

Each of these instructions clears (sets to zero) the specified accumulator. The Condition Code flags are also modified as follows:

- N** The Negative flag is cleared.
- Z** The Zero flag is set.
- V** The Overflow flag is cleared.
- C** The Carry flag is cleared.

Clearing the Q accumulator can be accomplished by executing both CLRD and CLRW.

To clear any of the Index Registers (X, Y, U or S), you can use either an Immediate Mode LD instruction or, on 6309 processors only, a TFR or EXG instruction which specifies the Zero register (0) as the source.

The CLRA and CLRB instructions provide the smallest, fastest way to clear the Carry flag in the CC register.

See Also: **CLR** (memory), **LD**



# CLR (memory)

## Store Zero into a Memory Byte

(M) ← 0

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| CLR             |           |   |   | 0F     | 6 / 5 | 2 | 6F      | 6+ | 2+ | 7F       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | 0 | 1 | 0 | 0 |

This instruction clears (sets to zero) the byte in memory at the Effective Address specified by the operand. The Condition Code flags are also modified as follows:

- N** The Negative flag is cleared.
- Z** The Zero flag is set.
- V** The Overflow flag is cleared.
- C** The Carry flag is cleared.

The CPU performs a Read-Modify-Write sequence when this instruction is executed and is therefore slower than an instruction which only writes to memory. When more than one byte needs to be cleared, you can optimize for speed by first clearing an accumulator and then using ST instructions to clear the memory bytes. The following examples illustrate this optimization:

Executes in 21 cycles (NM=0):

```
CLR    $200    ; 7 cycles
CLR    $210    ; 7 cycles
CLR    $220    ; 7 cycles
```

Adds one additional code byte, but saves 4 cycles:

```
CLRA                    ; 2 cycles
STA    $200             ; 5 cycles
STA    $210             ; 5 cycles
STA    $220             ; 5 cycles
```

See Also: **CLR** (accumulator), **ST**

# CMP (8 Bit)

## Compare Memory Byte from 8-Bit Accumulator

$$\text{TEMP} \leftarrow r - (M)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| CMPA            | 81        | 2 | 2 | 91     | 4 / 3 | 2 | A1      | 4+ | 2+ | B1       | 5 / 4 | 3 |
| CMPB            | C1        | 2 | 2 | D1     | 4 / 3 | 2 | E1      | 4+ | 2+ | F1       | 5 / 4 | 3 |
| CMPE            | 1181      | 3 | 3 | 1191   | 5 / 4 | 3 | 11A1    | 5+ | 3+ | 11B1     | 6 / 5 | 4 |
| CMPF            | 11C1      | 3 | 3 | 11D1   | 5 / 4 | 3 | 11E1    | 5+ | 3+ | 11F1     | 6 / 5 | 4 |

CMPE and CMPF are available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ~ |   | ↑ | ↑ | ↑ | ↑ |

These instructions subtract the contents of a byte in memory from the value contained in one of the 8-bit accumulators (A,B,E,F) and set the Condition Codes accordingly. Neither the memory byte nor the accumulator are modified.

- H** The affect on the Half-Carry flag is undefined for these instructions.
- N** The Negative flag is set equal to the value of bit 7 of the result.
- Z** The Zero flag is set if the resulting value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into bit-7 was needed; cleared otherwise.

The Compare instructions are usually used to set the Condition Code flags prior to executing a conditional branch instruction.

The 8-bit CMP instructions perform exactly the same operation as the 8-bit SUB instructions, with the exception that the value in the accumulator is not changed. Note that since a subtraction is performed, the Carry flag actually represents a Borrow.

See Also: **CMP** (16-bit), **CMPR**

# CMP (16 Bit)

## Compare Memory Word from 16-Bit Register

TEMP  $\leftarrow$  r - (M:M+1)

| SOURCE FORMS | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|--------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|              | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| CMPD         | 083       | 5 / 4 | 4 | 093    | 7 / 5 | 3 | 0A3     | 7+ / 6+ | 3+ | 0B3      | 8 / 6 | 4 |
| CMPS         | 18C       | 5 / 4 | 4 | 19C    | 7 / 5 | 3 | 1AC     | 7+ / 6+ | 3+ | 1BC      | 8 / 6 | 4 |
| CMPU         | 183       | 5 / 4 | 4 | 193    | 7 / 5 | 3 | 1A3     | 7+ / 6+ | 3+ | 1B3      | 8 / 6 | 4 |
| CMPW         | 081       | 5 / 4 | 4 | 091    | 7 / 5 | 3 | 0A1     | 7+ / 6+ | 3+ | 0B1      | 8 / 6 | 4 |
| CMPX         | 8C        | 4 / 3 | 3 | 9C     | 6 / 4 | 2 | AC      | 6+ / 5+ | 2+ | BC       | 7 / 5 | 3 |
| CMPY         | 08C       | 5 / 4 | 4 | 09C    | 7 / 5 | 3 | 0AC     | 7+ / 6+ | 3+ | 0BC      | 8 / 6 | 4 |

CMPW is available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

These instructions subtract the contents of a double-byte value in memory from the value contained in one of the 16-bit accumulators (D,W) or one of the Index/Stack registers (X,Y,U,S) and set the Condition Codes accordingly. Neither the memory bytes nor the register are modified unless an auto-increment / auto-decrement addressing mode is used with the same register.

- H** The Half-Carry flag is not affected by these instructions.
- N** The Negative flag is set equal to the value of bit 15 of the result.
- Z** The Zero flag is set if the resulting value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into bit 15 was needed; cleared otherwise.

The Compare instructions are usually used to set the Condition Code flags prior to executing a conditional branch instruction.

The 16-bit CMP instructions for accumulators perform exactly the same operation as the 16-bit SUB instructions, with the exception that the value in the accumulator is not changed. Note that since a subtraction is performed, the Carry flag actually represents a Borrow.

See Also: **CMP** (8-bit), **CMPR**

# CMPR

6309 ONLY

## Compare Source Register from Destination Register

TEMP  $\leftarrow$  r1 - r0

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| CMPR <i>r0,r1</i> | IMMEDIATE       | 1037   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

The CMPR instruction subtracts the contents of a source register from the contents of a destination register and sets the Condition Codes accordingly. Neither register is modified.

- H** The Half-Carry flag is not affected by this instruction.
- N** The Negative flag is set equal to the value of the high-order bit of the result.
- Z** The Zero flag is set if the resulting value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into the high-order bit was needed; cleared otherwise.

Any of the 6309 registers except Q and MD may be specified as the source operand, destination operand or both; however specifying the PC register as either the source or destination produces undefined results.

The CMPR instruction will perform either an 8-bit or 16-bit comparison according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See “6309 Inter-Register Operations” on page 143 for further details.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. See the description of the **TFR** instruction starting on page 137 for further details.

See Also: **ADD** (8-bit), **ADD** (16-bit)

# COM (accumulator)

## Complement Accumulator

$$r' \leftarrow \overline{r}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| COMA        | INHERENT        | 43     | 2 / 1  | 1          |
| COMB        | INHERENT        | 53     | 2 / 1  | 1          |
| COMD        | INHERENT        | 1043   | 3 / 2  | 2          |
| COME        | INHERENT        | 1143   | 3 / 2  | 2          |
| COMF        | INHERENT        | 1153   | 3 / 2  | 2          |
| COMW        | INHERENT        | 1053   | 3 / 2  | 2          |

COMD, COME, COMF and COMW are available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 | 1 |

Each of these instructions change the value of the specified accumulator to that of it's logical complement; that is each 1 bit is changed to a 0, and each 0 bit is changed to a 1. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of the accumulators high-order bit.
- Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is always set.

This instruction performs a ones-complement operation. A twos-complement can be achieved with the NEG instruction.

Complementing the Q accumulator requires executing both COMW and COMD.

The COMA and COMB instructions provide the smallest, fastest way to set the Carry flag in the CC register.

See Also: **COM** (memory), **NEG**

# COM (memory)

## Complement a Byte in Memory

$$(M)' \leftarrow \overline{(M)}$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| COM             |           |   |   | 03     | 6 / 5 | 2 | 63      | 6+ | 2+ | 73       | 7 / 6 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 | 1 |

This instruction changes the value of a byte in memory to that of its logical complement; that is each 1 bit is changed to a 0, and each 0 bit is changed to a 1. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of bit 7.
- Z** The Zero flag is set if the new value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is always set.

This instruction performs a ones-complement operation. A twos-complement can be achieved with the NEG instruction.

See Also: **COM** (accumulator), **NEG**

# CWAI

## Clear Condition Code Bits and Wait for Interrupt

$CC' \leftarrow CC \text{ AND } IMM$

$CC' \leftarrow CC \text{ OR } 80_{16} \text{ (E flag)}$

Push Onto S Stack: PC, U, Y, X, DP, [WIf NM = 1], D, CC

Halt Execution and Wait for Unmasked Interrupt

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|-------------|-----------------|--------|---------|------------|
| CWAI #i8    | IMMEDIATE       | 3C     | 22 / 20 | 2          |

This instruction logically ANDs the contents of the Condition Codes register with the 8-bit value specified by the immediate operand. The result is placed back into the Condition Codes register. The E flag in the CC register is then set and the *entire* machine state is pushed onto the hardware stack (S). The CPU then halts execution and waits for an unmasked interrupt to occur. When such an interrupt occurs, the CPU resumes execution at the address obtained from the corresponding interrupt vector.

You can specify a value in the immediate operand to clear either or both the I and F interrupt masks to ensure that the desired interrupt types are enabled. One of the following values is typically used for the immediate operand:

\$FF = Leave CC unmodified  
\$EF = Enable IRQ  
\$BF = Enable FIRQ  
\$AF = Enable both IRQ and FIRQ

Some assemblers will accept a comma-delimited list of the Condition Code bits to be cleared as an alternative to the immediate value. For example:

CWAI I,F ; Clear I and F, wait for interrupt

It is important to note that because the *entire* machine state is stacked prior to the actual occurrence of an interrupt, any FIRQ service routine that may be invoked must not assume that PC and CC are the only registers that have been stacked. The RTI instruction will operate correctly in this situation because CWAI sets the E flag prior to stacking the CC register.

Unlike SYNC, the CWAI instruction does not place the data and address busses in a high-impedance state while waiting for an interrupt.

See Also: **ANDCC**, **RTI**, **SYNC**

# DAA

## Decimal Addition Adjust

$A[4..7]' \leftarrow A[4..7] + 6 \text{ IF:}$   
     $CC.C = 1$   
    OR:  $A[4..7] > 9$   
    OR:  $A[4..7] > 8 \text{ AND } A[0..3] > 9$   
 $A[0..3]' \leftarrow A[0..3] + 6 \text{ IF:}$   
     $CC.H = 1$   
    OR:  $A[0..3] > 9$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| DAA         | INHERENT        | 19     | 2 / 1  | 1          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ~ | ↑ |

The DAA instruction is used after performing an 8-bit addition of Binary Coded Decimal values using either the ADDA or ADCA instructions. DAA adjusts the value resulting from the binary addition in accumulator A so that it contains the desired BCD result instead. The Carry flag is also updated to properly reflect BCD addition. That is, the Carry flag is set when addition of the most-significant digits (plus any carry from the addition of the least-significant digits) produces a value greater than 9.

- H** The Half-Carry flag is not affected by this instruction.
- N** The Negative flag is set equal to the new value of bit 7 in Accumulator A.
- Z** The Zero flag is set if the new value of Accumulator A is zero; cleared otherwise.
- V** The affect this instruction has on the Overflow flag is undefined.
- C** The Carry flag is set if the BCD addition produced a carry; cleared otherwise.

The code below adds the BCD values of 64 and 27, producing the BCD sum of 91:

```
LDA    #$64
ADDA    #$27    ; Produces binary result of $8B
DAA          ; Adjusts A to $91 (BCD result of 64 + 27)
```

DAA is the only instruction which is affected by the value of the Half Carry flag (H) in the Condition Codes register.

See Also: **ADCA**, **ADDA**



# DEC (accumulator)

## Decrement Accumulator

$$r' \leftarrow r - 1$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| DECA        | INHERENT        | 4A     | 2 / 1  | 1          |
| DECB        | INHERENT        | 5A     | 2 / 1  | 1          |
| DECD        | INHERENT        | 104A   | 3 / 2  | 2          |
| DECE        | INHERENT        | 114A   | 3 / 2  | 2          |
| DECF        | INHERENT        | 115A   | 3 / 2  | 2          |
| DECW        | INHERENT        | 105A   | 3 / 2  | 2          |

DECD, DECE, DECF and DECW are available on 6309 only.

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>E</b> | <b>F</b> | <b>H</b> | <b>I</b> | <b>N</b> | <b>Z</b> | <b>V</b> | <b>C</b> |
|          |          |          |          | ↕        | ↕        | ↕        |          |

These instructions subtract 1 from the specified accumulator. The Condition Code flags are affected as follows:

- N** The Negative flag is set equal to the new value of the accumulators high-order bit.
- Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.
- V** The Overflow flag is set if the original value was  $80_{16}$  (8-bit) or  $8000_{16}$  (16-bit); cleared otherwise.
- C** The Carry flag is not affected by these instructions.

It is important to note that the DEC instructions do not affect the Carry flag. This means that it is not always possible to optimize code by simply replacing a `SUBr #1` instruction with a corresponding `DECr`. Because the DEC instructions do not affect the Carry flag, they can be used to implement loop counters within multiple precision computations.

When used to decrement an unsigned value, only the BEQ and BNE branches will always behave as expected. When operating on a signed value, all of the signed conditional branch instructions will behave as expected.

See Also: **DEC** (memory), **INC**, **SUB**

# DEC (memory)

## Decrement a Byte in Memory

$$(M)' \leftarrow (M) - 1$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| DEC             |           |   |   | 0A     | 6 / 5 | 2 | 6A      | 6+ | 2+ | 7A       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ |   |

This instruction subtracts 1 from the value contained in a memory byte. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of bit 7.
- Z** The Zero flag is set if the new value of the memory byte is zero; cleared otherwise.
- V** The Overflow flag is set if the original value of the memory byte was \$80; cleared otherwise.
- C** The Carry flag is not affected by this instruction.

Because the DEC instruction does not affect the Carry flag, it can be used to implement a loop counter within a multiple precision computation.

When used to decrement an unsigned value, only the BEQ and BNE branches will always behave as expected. When operating on a signed value, all of the signed conditional branch instructions will behave as expected.

See Also: **DEC** (accumulator), **INC**, **SUB**

# DIVD

6309 ONLY

## Signed Divide of Accumulator D by 8-bit value in Memory

$ACCB' \leftarrow ACCD \div (M)$

$ACCA' \leftarrow ACCD \text{ MOD } (M)$

| SOURCE<br>FORMS | IMMEDIATE |     |   | DIRECT |        |   | INDEXED |      |    | EXTENDED |        |   |
|-----------------|-----------|-----|---|--------|--------|---|---------|------|----|----------|--------|---|
|                 | OP        | ~   | # | OP     | ~      | # | OP      | ~    | #  | OP       | ~      | # |
| DIVD            | 118D      | 25* | 3 | 119D   | 27/26* | 3 | 11AD    | 27+* | 3+ | 11BD     | 28/27* | 4 |

\* If a two's complement overflow occurs, the DIVD instruction uses one fewer cycle than what is shown in the table. If a range overflow occurs, DIVD uses 13 fewer cycles than what is shown in the table.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

This instruction divides the 16-bit value in Accumulator D (the dividend) by an 8-bit value contained in a memory byte (the divisor). The operation is performed using two's complement binary arithmetic. The 16-bit result consists of the 8-bit quotient placed in Accumulator B and the 8-bit remainder placed in Accumulator A. The sign of the remainder is always the same as the sign of the dividend unless the remainder is zero.

**N** The Negative flag is set equal to the new value of bit 7 in Accumulator B.

**Z** The Zero flag is set if the new value of Accumulator B is zero; cleared otherwise.

**V** The Overflow flag is set if an overflow occurred; cleared otherwise.

**C** The Carry flag is set if the quotient in Accumulator B is odd; cleared if even.

When the value of the specified memory byte (divisor) is zero, a *Division-By-Zero* exception is triggered. This causes the CPU to set bit 7 in the MD register, stack the machine state and jump to the address taken from the Illegal Instruction vector at \$FFF0.

Two types of overflow may occur when the DIVD instruction is executed:

- A two's complement overflow occurs when the sign of the resulting quotient is incorrect. For example, when 300 is divided by 2, the result of 150 can be represented in 8 bits only as an unsigned value. Since DIVD performs a signed operation, it interprets the result as -106 and sets the Negative (N) and Overflow (V) flags.
- A range overflow occurs when the quotient is larger than can be represented in 8 bits. For example, when 900 is divided by 3, the result of 300 exceeds the 8-bit range. In this case, the CPU aborts the operation, leaving the accumulators unmodified while setting the Overflow flag (V) and clearing the N, Z and C flags.

See Also: **DIVQ**

# DIVQ

6309 ONLY

## Signed Divide of Accumulator Q by 16-bit value in Memory

$ACCW' \leftarrow ACCQ \div (M:M+1)$

$ACCD' \leftarrow ACCQ \text{ MOD } (M:M+1)$

| SOURCE FORMS | IMMEDIATE |     |   | DIRECT |        |   | INDEXED |      |    | EXTENDED |        |   |
|--------------|-----------|-----|---|--------|--------|---|---------|------|----|----------|--------|---|
|              | OP        | ~   | # | OP     | ~      | # | OP      | ~    | #  | OP       | ~      | # |
| DIVQ         | 118E      | 34* | 4 | 119E   | 36/35* | 3 | 11AE    | 36+* | 3+ | 11BE     | 37/36* | 4 |

\* When a range overflow occurs, the DIVQ instruction uses 21 fewer cycles than what is shown in the table.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

This instruction divides the 32-bit value in Accumulator Q (the dividend) by a 16-bit value contained in memory (the divisor). The operation is performed using two's complement binary arithmetic. The 32-bit result consists of the 16-bit quotient placed in Accumulator W and the 16-bit remainder placed in Accumulator D. The sign of the remainder is always the same as the sign of the dividend unless the remainder is zero.

**N** The Negative flag is set equal to the new value of bit 15 in Accumulator W.

**Z** The Zero flag is set if the new value of Accumulator W is zero; cleared otherwise.

**V** The Overflow flag is set if an overflow occurred; cleared otherwise.

**C** The Carry flag is set if the quotient in Accumulator W is odd; cleared if even.

When the value of the specified memory word (divisor) is zero, a *Division-By-Zero* exception is triggered. This causes the CPU to set bit 7 in the MD register, stack the machine state and jump to the address taken from the Illegal Instruction vector at  $FFF0_{16}$ .

Two types of overflow are possible when the DIVQ instruction is executed:

- A two's complement overflow occurs when the sign of the resulting quotient is incorrect. For example, when 80,000 is divided by 2, the result of 40,000 can be represented in 16 bits only as an unsigned value. Since DIVQ is a signed operation, it interprets the result as -25,536 and sets the Negative (N) and Overflow (V) flags.
- A range overflow occurs when the quotient is larger than can be represented in 16 bits. For example, when 210,000 is divided by 3, the result of 70,000 exceeds the 16-bit range. In this case, the CPU aborts the operation, leaving the accumulators unmodified while setting the Overflow flag (V) and clearing the N, Z and C flags.

See Also: **DIVD**

# EIM

**6309 ONLY**

## Exclusive-OR of Immediate Value with Memory Byte

$$(M)' \leftarrow (M) \oplus IMM$$

| SOURCE FORM | IMMEDIATE |   |   | DIRECT |   |   | INDEXED |    |    | EXTENDED |   |   |
|-------------|-----------|---|---|--------|---|---|---------|----|----|----------|---|---|
|             | OP        | ~ | # | OP     | ~ | # | OP      | ~  | #  | OP       | ~ | # |
| EIM #i8;EA  |           |   |   | 05     | 6 | 3 | 65      | 7+ | 3+ | 75       | 7 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

The EIM instruction exclusively-ORs the contents of a byte in memory with an 8-bit immediate value. The resulting value is placed back into the designated memory location.

- N** The Negative flag is set equal to the new value of bit 7 of the memory byte.
- Z** The Zero flag is set if the new value of the memory byte is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

EIM is one of the instructions added to the 6309 which allow logical operations to be performed directly in memory instead of having to use an accumulator. It takes three separate instructions to perform the same operation on a 6809:

6809 (6 instruction bytes; 12 cycles):

```
LDA    #$3F
EORA   4,U
STA    4,U
```

6309 (3 instruction bytes; 8 cycles):

```
EIM    #$3F;4,U
```

Note that the assembler syntax used for the EIM operand is non-typical. Some assemblers may require a comma (,) rather than a semicolon (;) between the immediate operand and the address operand.

The object code format for the EIM instruction is:

|        |             |                         |
|--------|-------------|-------------------------|
| OPCODE | IMMED VALUE | ADDRESS / INDEX BYTE(S) |
|--------|-------------|-------------------------|

See Also: **AIM**, **OIM**, **TIM**

## EOR (8 Bit)

### Exclusively-OR Memory Byte with Accumulator A or B

$$r' \leftarrow r \oplus (M)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| EORA            | 88        | 2 | 2 | 98     | 4 / 3 | 2 | A8      | 4+ | 2+ | B8       | 5 / 4 | 3 |
| EORB            | C8        | 2 | 2 | D8     | 4 / 3 | 2 | E8      | 4+ | 2+ | F8       | 5 / 4 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions exclusively-OR the contents of a byte in memory with either Accumulator A or B. The 8-bit result is then placed in the specified accumulator.

- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

The EOR instruction produces a result containing '1' bits in the positions where the corresponding bits in the two operands have different values. Exclusive-OR logic is often used in parity functions.

EOR can also be used to perform "bit-flipping" since a '1' bit in the source operand will invert the value of the corresponding bit in the destination operand. For example:

```
EORA    #%00000100    ;Invert value of bit 2 in Accumulator A
```

See Also: **BEOR, BIEOR, EIM, EORD, EORR**

# EORD

6309 ONLY

## Exclusively-OR Memory Word with Accumulator D

$$\text{ACCD}' \leftarrow \text{ACCD} \oplus (\text{M}:\text{M}+1)$$

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| EORD           | 1088      | 5 / 4 | 4 | 1098   | 7 / 5 | 3 | 10A8    | 7+ / 6+ | 3+ | 10B8     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

The EORD instruction exclusively-ORs the contents of a double-byte value in memory with the contents of Accumulator D. The 16-bit result is placed back into Accumulator D.

- N** The Negative flag is set equal to the new value of bit 15 of Accumulator D.
- Z** The Zero flag is set if the new value of the Accumulator D is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

The EORD instruction produces a result containing '1' bits in the positions where the corresponding bits in the two operands have different values. Exclusive-OR logic is often used in parity functions.

EOR can also be used to perform "bit-flipping" since a '1' bit in the source operand will invert the value of the corresponding bit in the destination operand. For example:

```
EORD    #$8080    ;Invert values of bits 7 and 15 in D
```

See Also: **EOR** (8-bit), **EORR**

# EORR

6309 ONLY

## Exclusively-OR Source Register with Destination Register

$$r1' \leftarrow r1 \oplus r0$$

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| EORR <i>r0,r1</i> | IMMEDIATE       | 1036   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The EORR instruction exclusively-ORs the contents of a source register with the contents of a destination register. The result is placed into the destination register.

- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

All of the 6309 registers except Q and MD can be specified as either the source or destination; however specifying the PC register as either the source or destination produces undefined results.

The EORR instruction produces a result containing '1' bits in the positions where the corresponding bits in the two operands have different values. Exclusive-OR logic is often used in parity functions.

See “6309 Inter-Register Operations” on page 143 for details on how this instruction operates when registers of different sizes are specified.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. For details, see the description of the **TFR** instruction.

See Also: **EOR** (8-bit), **EORD**



# EXG

## 6309 IMPLEMENTATION

### Exchange Registers

$r0 \leftrightarrow r1$

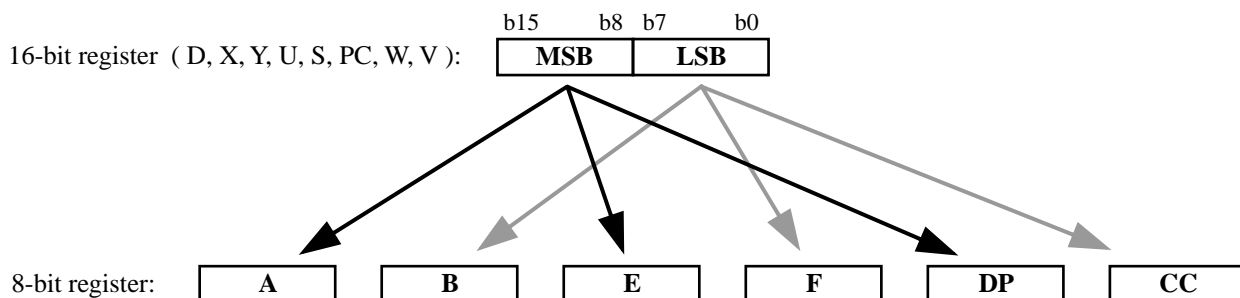
| SOURCE FORM      | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------|-----------------|--------|--------|------------|
| EXG <i>r0,r1</i> | IMMEDIATE       | 1E     | 8 / 5  | 2          |

This instruction exchanges the contents of two registers. None of the Condition Code flags are affected unless CC is one of the registers involved in the exchange.

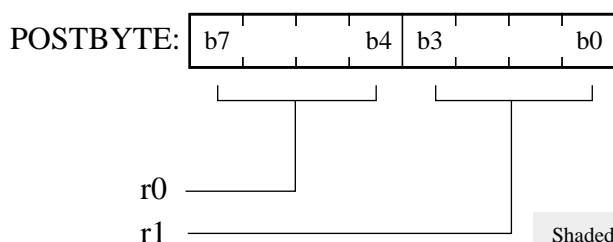
Program flow can be altered by specifying PC as one of the registers. When this occurs, the other register is set to the address of the instruction that follows EXG.

Any of the 6309 registers except Q and MD may be used in the exchange. The order in which the two registers are specified is irrelevant. For example, EXG A,B will operate exactly the same as EXG B,A although the object code will be different.

When an 8-bit register is exchanged with a 16-bit register, the contents of the 8-bit register are placed into both halves of the 16-bit register. Conversely, only the upper or the lower half of the 16-bit register is placed into the 8-bit register. As illustrated in the diagram below, which half is transferred depends on which 8-bit register is involved.



The EXG instruction requires a postbyte in which the two registers that are involved are encoded into the upper and lower nibbles.



Shaded encodings are invalid  
on 6809 microprocessors

| Code | Register | Code | Register |
|------|----------|------|----------|
| 0000 | D        | 1000 | A        |
| 0001 | X        | 1001 | B        |
| 0010 | Y        | 1010 | CC       |
| 0011 | U        | 1011 | DP       |
| 0100 | S        | 1100 | 0        |
| 0101 | PC       | 1101 | 0        |
| 0110 | W        | 1110 | E        |
| 0111 | V        | 1111 | F        |

See Also: **EXG** (6809 implementation), **TFR**

# EXG

## 6809 IMPLEMENTATION

### Exchange Registers

$r0 \leftrightarrow r1$

| SOURCE FORM      | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------|-----------------|--------|--------|------------|
| EXG <i>r0,r1</i> | IMMEDIATE       | 1E     | 8      | 2          |

This instruction exchanges the contents of two registers. None of the Condition Code flags are affected unless CC is one of the registers involved in the exchange.

Program flow can be altered by specifying PC as one of the registers. When this occurs, the other register is set to the address of the instruction that follows EXG.

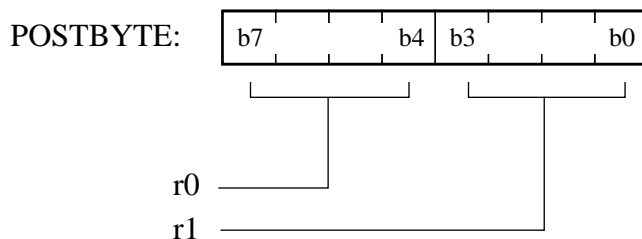
Any of the 6809 registers may be used in the exchange. When exchanging registers of the same size, the order in which they are specified is irrelevant. For example, EXG A,B will operate exactly the same as EXG B,A although the object code will be different.

When exchanging registers of different sizes, a 6809 operates differently than a 6309. The 8-bit register is always exchanged with the lower half of the 16-bit register, and the the upper half of the 16-bit register is then set to the value shown in the table below.

| Operand Order | 8-bit Register Used | 16-bit Register's MSB after EXG |
|---------------|---------------------|---------------------------------|
| 16 , 8        | Any                 | FF <sub>16</sub>                |
| 8 , 16        | A or B              | FF <sub>16</sub> *              |
| 8 , 16        | CC or DP            | Same as LSB                     |

\*The one exception is for EXG A,D which produces exactly the same result as EXG A,B

The EXG instruction requires a postbyte in which the two registers are encoded into the upper and lower nibbles.



| Code | Register       | Code | Register       |
|------|----------------|------|----------------|
| 0000 | D              | 1000 | A              |
| 0001 | X              | 1001 | B              |
| 0010 | Y              | 1010 | CC             |
| 0011 | U              | 1011 | DP             |
| 0100 | S              | 1100 | <i>invalid</i> |
| 0101 | PC             | 1101 | <i>invalid</i> |
| 0110 | <i>invalid</i> | 1110 | <i>invalid</i> |
| 0111 | <i>invalid</i> | 1111 | <i>invalid</i> |

If an invalid register encoding is specified for either register, a constant value of FF<sub>16</sub> or FFFF<sub>16</sub> is used for the exchange. **The invalid register encodings have valid meanings on 6309 processors, and should be avoided in code intended to run on both CPU's.**

See Also: **EXG** (6309 implementation), **TFR**

# INC (accumulator)

## Increment Accumulator

$$r' \leftarrow r + 1$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| INCA        | INHERENT        | 4C     | 2 / 1  | 1          |
| INCB        | INHERENT        | 5C     | 2 / 1  | 1          |
| INCD        | INHERENT        | 104C   | 3 / 2  | 2          |
| INCE        | INHERENT        | 114C   | 3 / 2  | 2          |
| INCF        | INHERENT        | 115C   | 3 / 2  | 2          |
| INCW        | INHERENT        | 105C   | 3 / 2  | 2          |

INCD, INCE, INCF and INCW are available on 6309 only.

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>E</b> | <b>F</b> | <b>H</b> | <b>I</b> | <b>N</b> | <b>Z</b> | <b>V</b> | <b>C</b> |
|          |          |          |          | ↑↓       | ↑↓       | ↑↓       |          |

These instructions add 1 to the contents of the specified accumulator. The Condition Code flags are affected as follows:

- N** The Negative flag is set equal to the new value of the accumulators high-order bit.
- Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.
- V** The Overflow flag is set if the original value was \$7F (8-bit) or \$7FFF (16-bit); cleared otherwise.
- C** The Carry flag is not affected by these instructions.

It is important to note that the INC instructions do not affect the Carry flag. This means that it is not always possible to optimize code by simply replacing an `ADD r #1` instruction with a corresponding `INCr`.

When used to increment an unsigned value, only the BEQ and BNE branches will consistently behave as expected. When operating on a signed value, all of the signed conditional branch instructions will behave as expected.

See Also: **ADD**, **DEC**, **INC** (memory)

# INC (memory)

## Increment a Byte in Memory

$$(M)' \leftarrow (M) + 1$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| INC             |           |   |   | 0C     | 6 / 5 | 2 | 6C      | 6+ | 2+ | 7C       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ |   |

This instruction adds 1 to the contents of a memory byte. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of bit 7.
- Z** The Zero flag is set if the new value of the memory byte is zero; cleared otherwise.
- V** The Overflow flag is set if the original value of the memory byte was \$7F; cleared otherwise.
- C** The Carry flag is not affected by this instruction.

Because the INC instruction does not affect the Carry flag, it can be used to implement a loop counter within a multiple precision computation.

When used to increment an unsigned value, only the BEQ and BNE branches will consistently behave as expected. When operating on a signed value, all of the signed conditional branch instructions will behave as expected.

See Also: **ADD**, **DEC**, **INC** (accumulator)

# JMP

## Unconditional Jump

$PC' \leftarrow EA$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| JMP             |           |   |   | 0E     | 3 / 2 | 2 | 6E      | 3+ | 2+ | 7E       | 4 / 3 | 3 |

E F H I N Z V C

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

This instruction causes an unconditional jump. None of the Condition Code flags are affected by this instruction.

The JMP instruction is similar in function to the BRA and LBRA instructions in that it always causes execution to be transferred to the effective address specified by the operand. The primary difference is that BRA and LBRA use only the Relative Addressing mode, whereas JMP uses only the Direct, Indexed or Extended modes.

Unlike most other instructions which use the Direct, Indexed and Extended addressing modes, the operand value used by the JMP instruction is the Effective Address itself, rather than the memory contents stored at that address (unless Indirect Indexing is used). Here are some examples:

```
JMP    $4000    ; Jumps to address $4000
JMP    [$4000]  ; Jumps to address stored at $4000
JMP    ,X       ; Jumps to the address in X
JMP    B,X      ; Jumps to computed address X + B
JMP    [B,X]    ; Jumps to address stored at X + B
JMP    <$80     ; Jumps to address (DP * $100) + $80
```

Indexed operands are useful in that they provide the ability to compute the destination address at run-time. The use of an Indirect Indexing mode is frequently used to call routines through a jump-table in memory.

Using Direct or Extended operands with the JMP instruction should be avoided in position-independent code unless the destination address is within non-relocatable code (such as a ROM routine).

See Also: **BRA, JSR, LBRA**

# JSR

## Unconditional Jump to Subroutine

$$\begin{aligned} S' &\leftarrow S - 2 \\ (S : S+1) &\leftarrow PC \\ PC' &\leftarrow EA \end{aligned}$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| JSR             |           |   |   | 9D     | 7 / 6 | 2 | AD      | 7+ / 6+ | 2+ | BD       | 8 / 7 | 3 |

E F H I N Z V C

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

This instruction pushes the value of the PC register (after the JSR instruction bytes have been fetched) onto the hardware stack and then performs an unconditional jump. None of the Condition Code flags are affected. By pushing the PC value onto the stack, the called subroutine can "return" to this address after it has completed.

The JSR instruction is similar in function to that of the BSR and LBSR instructions. The primary difference is that BSR and LBSR use only the Relative Addressing mode, whereas JSR uses only the Direct, Indexed or Extended modes.

Unlike most other instructions which use the Direct, Indexed and Extended addressing modes, the operand value used by the JSR instruction is the Effective Address itself, rather than the memory contents stored at that address (unless Indirect Indexing is used). Here are some examples:

```
JSR    $4000    ; Calls to address $4000
JSR    [$4000]  ; Calls to the address stored at $4000
JSR    ,X       ; Calls to the address in X
JSR    [B,X]    ; Calls to the address stored at X + B
```

Indexed operands are useful in that they provide the ability to compute the subroutine address at run-time. The use of an Indirect Indexing mode is frequently used to call subroutines through a jump-table in memory.

Using Direct or Extended operands with the JSR instruction should be avoided in position-independent code unless the destination address is within non-relocatable code (such as a ROM routine).

See Also: **BSR, JMP, LBSR, PULS, RTS**

# LBCC

## Long Branch If Carry Clear

IF  $CC.C = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBCC <i>address</i> | RELATIVE        | 1024   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is clear (0), causes a relative branch. If the Carry flag is 1, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBCC instruction will branch if the source value was higher or the same as the original destination value. For this reason, 6809/6309 assemblers will accept LBHS as an alternate mnemonic for LBCC.

LBCC is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag. Also, the LBCC instruction will always branch following a CLR instruction and never branch following a COM instruction due to the way those instructions affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the LBCC instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BCC instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BCC**, **LBCS**, **LBGE**

# LBCS

## Long Branch If Carry Set

IF  $CC.C \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBCS <i>address</i> | RELATIVE        | 1025   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is set (1), causes a relative branch. If the Carry flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBCS instruction will branch if the source value was lower than the original destination value. For this reason, 6809/6309 assemblers will accept **LBLO** as an alternate mnemonic for LBCS.

LBCS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag. Also, the LBCS instruction will never branch following a CLR instruction and always branch following a COM instruction due to the way those instructions affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the LBCS instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BCS instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BCS**, **LBCC**, **LBLT**



# LBEQ

## Long Branch If Equal to Zero

IF  $CC.Z \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBEQ <i>address</i> | RELATIVE        | 1027   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero flag in the CC register and, if it is set (1), causes a relative branch. If the Z flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following almost any instruction that produces, tests or moves a value, the LBEQ instruction will branch if that value is equal to zero. In the case of an instruction that performs a subtract or compare, the LBEQ instruction will branch if the source value was equal to the original destination value.

LBEQ is generally not useful following a CLR instruction since the Z flag is always set.

The following instructions produce or move values, but do not affect the Z flag:

|      |      |      |       |       |
|------|------|------|-------|-------|
| ABX  | BAND | BEOR | BIAND | BIEOR |
| BOR  | BIOR | EXG  | LDBT  | LDMD  |
| LEAS | LEAU | PSH  | PUL   | STBT  |
| TFR  |      |      |       |       |

The branch address is calculated by adding the current value of the PC register (after the LBEQ instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BEQ instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BEQ**, **LBNE**

# LBGE

## Long Branch If Greater than or Equal to Zero

IF  $CC.N = CC.V$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBGE <i>address</i> | RELATIVE        | 102C   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) and Overflow (V) flags in the CC register and, if both are set OR both are clear, causes a relative branch. If the N and V flags do not have the same value then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the LBGE instruction will branch if the source value was greater than or equal to the original destination value.

The branch address is calculated by adding the current value of the PC register (after the LBGE instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BGE instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BGE, LBHS, LBLT**

# LBGT

## Long Branch If Greater Than Zero

IF (CC.N = CC.V) AND (CC.Z = 0) then PC' ← PC + IMM

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBGT <i>address</i> | RELATIVE        | 102E   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) flag in the CC register and, if it is clear AND the values of the Negative (N) and Overflow (V) flags are equal (both set OR both clear), causes a relative branch. If the N and V flags do not have the same value or if the Z flag is set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the LBGT instruction will branch if the source value was greater than the original destination value.

The branch address is calculated by adding the current value of the PC register (after the LBGT instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BGT instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BGT, LBHI, LBLE**

# LBHI

## Long Branch If Higher

IF (CC.Z = 0) AND (CC.C = 0) then PC' ← PC + IMM

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBHI <i>address</i> | RELATIVE        | 1022   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) and Carry (C) flags in the CC register and, if both are zero, causes a relative branch. If either the Z or C flags are set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBHI instruction will branch if the source value was higher than the original destination value.

LBHI is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the LBHI instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BHI instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BHI**, **LBGT**, **LBLS**

# LBHS

## Long Branch If Higher or Same

IF  $CC.C = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBHS <i>address</i> | RELATIVE        | 1024   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is clear (0), causes a relative branch. If the Carry flag is 1, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBHS instruction will branch if the source value was higher or the same as the original destination value.

LBHS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

LBHS is an alternate mnemonic for the LBCC instruction. Both produce the same object code.

The branch address is calculated by adding the current value of the PC register (after the LBHS instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BHS instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BHS**, **LBGE**, **LBLO**

# LBLE

## Long Branch If Less than or Equal to Zero

IF (CC.N  $\neq$  CC.V) OR (CC.Z = 1) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBLE <i>address</i> | RELATIVE        | 102F   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction performs a relative branch if the value of the Zero (Z) flag is 1, OR if the values of the Negative (N) and Overflow (V) flags are not equal. If the N and V flags have the same value and the Z flag is not set then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the LBLE instruction will branch if the source value was less than or equal to the original destination value.

The branch address is calculated by adding the current value of the PC register (after the LBLE instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BLE instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BLE**, **LBGT**, **LBL**

# LBLO

## Long Branch If Lower

IF  $CC.C \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBLO <i>address</i> | RELATIVE        | 1025   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Carry flag in the CC register and, if it is set (1), causes a relative branch. If the Carry flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBLO instruction will branch if the source value was lower than the original destination value.

LBLO is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

LBLO is an alternate mnemonic for the LBCS instruction. Both produce the same object code.

The branch address is calculated by adding the current value of the PC register (after the LBLO instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BLO instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BLO**, **LBHS**, **LBLT**

# LBS

## Long Branch If Lower or Same

IF (CC.Z  $\neq$  0) OR (CC.C  $\neq$  0) then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM        | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|--------------------|-----------------|--------|---------|------------|
| LBS <i>address</i> | RELATIVE        | 1023   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero (Z) and Carry (C) flags in the CC register and, if either are set, causes a relative branch. If both the Z and C flags are clear then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of unsigned binary values, the LBS instruction will branch if the source value was lower than or the same as the original destination value.

LBS is generally not useful following INC, DEC, LD, ST or TST instructions since none of those affect the Carry flag.

The branch address is calculated by adding the current value of the PC register (after the LBS instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BS instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BS**, **LBHI**, **LBLE**



# LBLT

## Long Branch If Less Than Zero

IF  $CC.N \neq CC.V$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBLT <i>address</i> | RELATIVE        | 102D   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction performs a relative branch if the values of the Negative (N) and Overflow (V) flags are not equal. If the N and V flags have the same value then the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following a subtract or compare of signed (twos-complement) values, the LBLT instruction will branch if the source value was less than the original destination value.

The branch address is calculated by adding the current value of the PC register (after the LBLT instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BLT instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BLT**, **LBGE**, **LBLO**

# LBMI

## Long Branch If Minus

IF  $CC.N \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBMI <i>address</i> | RELATIVE        | 102B   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) flag in the CC register and, if it is set (1), causes a relative branch. If the N flag is 0, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the LBMI instruction will branch if the resulting value is negative. It is generally preferable to use the LBLT instruction following such an operation because the sign bit may be invalid due to a twos-complement overflow.

The branch address is calculated by adding the current value of the PC register (after the LBMI instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BMI instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BMI**, **LBLT**, **LBPL**

# LBNE

## Long Branch If Not Equal to Zero

IF  $CC.Z = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBNE <i>address</i> | RELATIVE        | 1026   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Zero flag in the CC register and, if it is clear (0), causes a relative branch. If the Z flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following almost any instruction that produces, tests or moves a value, the LBNE instruction will branch if that value is not equal to zero. In the case of an instruction that performs a subtract or compare, the LBNE instruction will branch if the source value was not equal to the original destination value.

LBNE is generally not useful following a CLR instruction since the Z flag is always set.

The following instructions produce or move values, but do not affect the Z flag:

|      |      |      |       |       |
|------|------|------|-------|-------|
| ABX  | BAND | BEOR | BIAND | BIEOR |
| BOR  | BIOR | EXG  | LDBT  | LDMD  |
| LEAS | LEAU | PSH  | PUL   | STBT  |
| TFM  | TFR  |      |       |       |

The branch address is calculated by adding the current value of the PC register (after the LBNE instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BNE instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BNE**, **LBEQ**

# LBPL

## Long Branch If Plus

IF CC.N = 0 then PC'  $\leftarrow$  PC + IMM

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBPL <i>address</i> | RELATIVE        | 102A   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Negative (N) flag in the CC register and, if it is clear (0), causes a relative branch. If the N flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the LBPL instruction will branch if the resulting value is positive. It is generally preferable to use the LBGE instruction following such an operation because the sign bit may be invalid due to a twos-complement overflow.

The branch address is calculated by adding the current value of the PC register (after the LBPL instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BPL instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BPL**, **LBGE**, **LBMI**

# LBRA

## Long Branch Always

$$PC' \leftarrow PC + IMM$$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|---------------------|-----------------|--------|--------|------------|
| LBRA <i>address</i> | RELATIVE        | 16     | 5 / 4  | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction causes an unconditional relative branch. None of the Condition Code flags are affected.

The LBRA instruction is similar in function to the JMP instruction in that it always causes execution to be transferred to the effective address specified by the operand. The primary difference is that LBRA uses the Relative Addressing mode which allows the code to be position-independent.

The branch address is calculated by adding the current value of the PC register (after the LBRA instruction bytes have been fetched) with the 16-bit twos-complement value contained in the second and third bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BRA instruction can be used when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BRA**, **LBRN**, **JMP**

# LBRN

## Long Branch Never

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|---------------------|-----------------|--------|--------|------------|
| LBRN <i>address</i> | RELATIVE        | 1021   | 5      | 4          |

E F H I N Z V C

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

This instruction is essentially a no-operation; that is, the CPU never branches but merely advances the Program Counter to the next instruction in sequence. None of the Condition Code flags are affected.

The LBRN instruction provides a 4-byte no-op that consumes 5 bus cycles, whereas NOP is a single-byte instruction that consumes either 1 or 2 bus cycles. In addition, there is the BRN instruction which provides a 2-byte no-op that consumes 3 bus cycles.

Since the branch is never taken, the third and fourth bytes of the instruction do not serve any purpose and may contain any value. These bytes could contain program code or data that is accessed by some other instruction(s).

See Also: **BRN**, **LBRA**, **NOP**

# LBSR

## Long Branch to Subroutine

$$\begin{aligned} S' &\leftarrow S - 2 \\ (S : S+1) &\leftarrow PC \\ PC' &\leftarrow PC + IMM \end{aligned}$$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|---------------------|-----------------|--------|--------|------------|
| LBSR <i>address</i> | RELATIVE        | 17     | 9 / 7  | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction pushes the value of the PC register (after the LBSR instruction bytes have been fetched) onto the hardware stack and then performs an unconditional relative branch. None of the Condition Code flags are affected.

By pushing the PC value onto the stack, the called subroutine can "return" to this address after it has completed.

The LBSR instruction is similar in function to the JSR instruction. The primary difference is that LBSR uses the Relative Addressing mode which allows the code to be position-independent.

The branch address is calculated by adding the current value of the PC register (after the LBSR instruction bytes have been fetched) with the 16-bit twos-complement value contained in the second and third bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BSR instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BSR**, **JSR**, **PULS**, **RTS**

# LBVC

## Long Branch If Overflow Clear

IF  $CC.V = 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBVC <i>address</i> | RELATIVE        | 1028   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Overflow (V) flag in the CC register and, if it is clear (0), causes a relative branch. If the V flag is set, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the LBVC instruction will branch if there was no overflow.

The branch address is calculated by adding the current value of the PC register (after the LBVC instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BVC instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BVC**, **LBVS**



# LBVS

## Long Branch If Overflow Set

IF  $CC.V \neq 0$  then  $PC' \leftarrow PC + IMM$

| SOURCE FORM         | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---------------------|-----------------|--------|---------|------------|
| LBVS <i>address</i> | RELATIVE        | 1029   | 5 (6) * | 4          |

\*The 6809 requires 6 cycles only if the branch is taken.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction tests the Overflow (V) flag in the CC register and, if it is set (1), causes a relative branch. If the V flag is clear, the CPU continues executing the next instruction in sequence. None of the Condition Code flags are affected by this instruction.

When used following an operation on signed (twos-complement) binary values, the LBVS instruction will branch if an overflow occurred.

The branch address is calculated by adding the current value of the PC register (after the LBVS instruction bytes have been fetched) with the 16-bit twos-complement value contained in the third and fourth bytes of the instruction. Long branch instructions permit a relative jump to any location within the 64K address space. The smaller, faster BVS instruction can be used instead when the destination address is within -126 to +129 bytes of the address of the branch instruction.

See Also: **BVS**, **LBVC**

## LD (8 Bit)

### Load Data into 8-Bit Accumulator

$r' \leftarrow \text{IMM8} \mid (M)$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| LDA             | 86        | 2 | 2 | 96     | 4 / 3 | 2 | A6      | 4+ | 2+ | B6       | 5 / 4 | 3 |
| LDB             | C6        | 2 | 2 | D6     | 4 / 3 | 2 | E6      | 4+ | 2+ | F6       | 5 / 4 | 3 |
| LDE             | 1186      | 3 | 3 | 1196   | 5 / 4 | 3 | 11A6    | 5+ | 3+ | 11B6     | 6 / 5 | 4 |
| LDF             | 11C6      | 3 | 3 | 11D6   | 5 / 4 | 3 | 11E6    | 5+ | 3+ | 11F6     | 6 / 5 | 4 |

LDE and LDF are available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions load either an 8-bit immediate value or the contents of a memory byte into one of the 8-bit accumulators (A,B,E,F). The Condition Codes are affected as follows.

- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by these instructions.

See Also: **LD** (16-bit), **LDQ**

# LD (16 Bit)

## Load Data into 16-Bit Register

$r' \leftarrow \text{IMM16} \mid (M:M+1)$

| SOURCE FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| LDD          | CC        | 3 | 3 | DC     | 5 / 4 | 2 | EC      | 5+ | 2+ | FC       | 6 / 5 | 3 |
| LDS          | 10CE      | 4 | 4 | 10DE   | 6 / 5 | 3 | 10EE    | 6+ | 3+ | 10FE     | 7 / 6 | 4 |
| LDU          | CE        | 3 | 3 | DE     | 5 / 4 | 2 | EE      | 5+ | 2+ | FE       | 6 / 5 | 3 |
| LDW          | 1086      | 4 | 4 | 1096   | 6 / 5 | 3 | 10A6    | 6+ | 3+ | 10B6     | 7 / 6 | 4 |
| LDX          | 8E        | 3 | 3 | 9E     | 5 / 4 | 2 | AE      | 5+ | 2+ | BE       | 6 / 5 | 3 |
| LDY          | 108E      | 4 | 4 | 109E   | 6 / 5 | 3 | 10AE    | 6+ | 3+ | 10BE     | 7 / 6 | 4 |

LDW is available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions load either a 16-bit immediate value or the contents from a pair of memory bytes (in big-endian order) into one of the 16-bit accumulators (D,W) or one of the 16-bit Index registers (X,Y,U,S). The Condition Codes are affected as follows.

- N** The Negative flag is set equal to the new value of bit 15 of the register.
- Z** The Zero flag is set if the new register value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by these instructions.

See Also: **LD** (8-bit), **LDQ**, **LEA**

# LDBT

6309 ONLY

## Load Memory Bit into Register Bit

$r.dstBit' \leftarrow (DPM).srcBit$

| SOURCE FORM                  | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------------------|-----------------|--------|--------|------------|
| LDBT <i>r,sBit,dBit,addr</i> | DIRECT          | 1136   | 7 / 6  | 4          |

The LDBT instruction loads the value of a specified bit in memory into a specified bit of either the A, B or CC registers. None of the Condition Code flags are affected by the operation unless CC is specified as the register, in which case only the destination bit will be affected. The usefulness of the LDBT instruction is limited by the fact that only Direct Addressing is permitted.

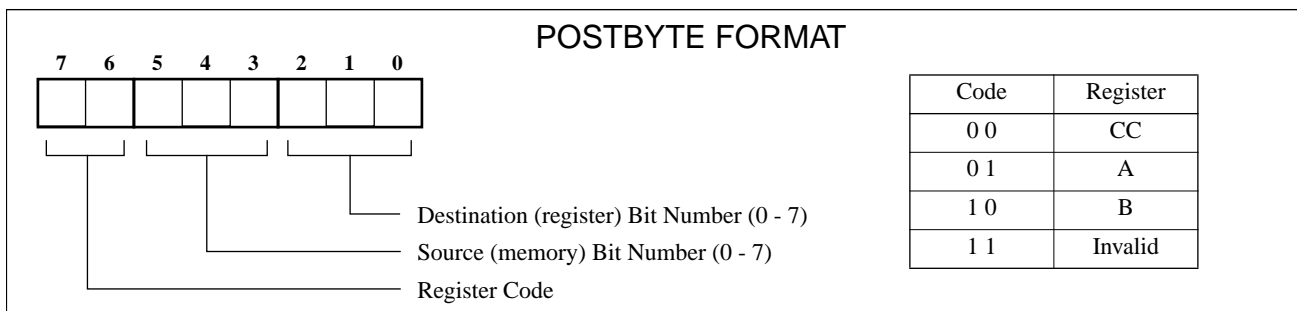


The figure above shows an example of the LDBT instruction where bit 1 of Accumulator A is Loaded with bit 5 of the byte in memory at address \$0040 (DP = 0).

The assembler syntax for this instruction can be confusing due to the ordering of the operands: *destination register, source bit, destination bit, source address*.

The object code format for the LDBT instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$36 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|



See Also: **BAND, BEOR, BIAND, BIEOR, BIOR, BOR, STBT**

# LDMD

6309 ONLY

## Load an Immediate Value into the MD Register

$MD.NM' \leftarrow IMM.0$

$MD.FM' \leftarrow IMM.1$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| LDMD #i8    | IMMEDIATE       | 113D   | 5      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   |   |   |   |

This instruction loads the two least-significant bits of the MD register (the *Native Mode* and *FIRQ Mode* control bits) with the two least-significant bits of the immediate operand. None of the Condition Code flags are affected.

The LDMD instruction provides the method by which the 6309 execution mode can be changed. Upon RESET, both the NM and FM mode bits are cleared. The execution mode may then be changed at any time by executing an LDMD instruction. See page 144 for more information about the 6309 execution modes.

Care should be taken when changing the value of the NM bit inside of an interrupt service routine because doing so can affect the behavior of an RTI instruction.

Bits 2 through 7 of the MD register are not affected by this instruction, so it cannot be used to alter the /0 and IL status bits.

The figure below shows the layout of the MD register:

|    |    |   |   |   |   |    |    |
|----|----|---|---|---|---|----|----|
| 7  | 6  | 5 | 4 | 3 | 2 | 1  | 0  |
| /0 | IL |   |   |   |   | FM | NM |

See Also: **BITMD**, **RTI**

# LDQ

6309 ONLY

## Load 32-bit Data into Accumulator Q

$$Q' \leftarrow \text{IMM32} \mid (M:M+3)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| LDQ             | CD        | 5 | 5 | 10DC   | 8 / 7 | 3 | 10EC    | 8+ | 3+ | 10FC     | 9 / 8 | 4 |

E F H I N Z V C

|  |  |  |  |   |   |   |  |
|--|--|--|--|---|---|---|--|
|  |  |  |  | ↑ | ↑ | 0 |  |
|--|--|--|--|---|---|---|--|

This instruction loads either a 32-bit immediate value or the contents of a quad-byte value from memory (in big-endian order) into the Q accumulator. The Condition Codes are affected as follows.

- N** The Negative flag is set equal to the new value of bit 31 of Accumulator Q.
- Z** The Zero flag is set if the new value of Accumulator Q is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by this instruction.

See Also: **LD** (8-bit), **LD** (16-bit)

# LEA

## Load Effective Address

$r' \leftarrow EA$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| LEAS        | INDEXED         | 32     | 4+     | 2+         |
| LEAU        | INDEXED         | 33     | 4+     | 2+         |
| LEAX        | INDEXED         | 30     | 4+     | 2+         |
| LEAY        | INDEXED         | 31     | 4+     | 2+         |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   | * |   |   |

\* The Z flag is updated by LEAX and LEAY only.

These instructions compute the effective address from an Indexed Addressing Mode operand and place that address into one of the Stack Pointers (S or U) or one of the Index Registers (X or Y).

The LEAS and LEAU instructions do not affect any of the Condition Code flags. The LEAX and LEAY instructions set the Z flag when the effective address is 0 and clear it otherwise. This permits X and Y to be used as 16-bit loop counters as well as providing compatibility with the INX and DEX instructions of the 6800 microprocessor.

LEA instructions differ from LD instructions in that the value loaded into the register is the address specified by the operand rather than the data pointed to by the address. LEA instructions might be used when you need to pass a parameter by-reference as opposed to by-value.

The LEA instructions can be quite versatile. For example, adding the contents of Accumulator B to Index Register Y and depositing the result in the User Stack pointer (U) can be accomplished with the single instruction:

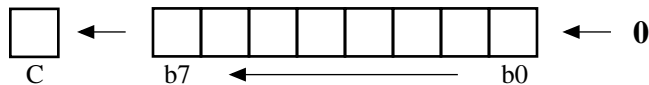
LEAU      B, Y

**NOTE:** The effective address of an auto-increment operand is the value prior to incrementing. Therefore, an instruction such as LEAX ,X+ will leave X unmodified. To achieve the expected results, you can use LEAX 1,X instead.

See Also: **ADDR**, **LD** (16-bit), **SUBR**

# LSL (8 Bit)

## Logical Shift Left of 8-Bit Accumulator or Memory Byte



| SOURCE FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| LSLA         | 48       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| LSLB         | 58       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| LSL          |          |       |   | 08     | 6 / 5 | 2 | 68      | 6+ | 2+ | 78       | 7 / 6 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ~ |   | ↕ | ↕ | ↕ | ↕ |

These instructions shift the contents of the A or B accumulator or a specified byte in memory to the left by one bit, clearing bit 0. Bit 7 is shifted into the Carry flag of the Condition Codes register.

- H** The affect on the Half-Carry flag is undefined for these instructions.
- N** The Negative flag is set equal to the new value of bit 7; previously bit 6.
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is set to the Exclusive-OR of the original values of bits 6 and 7.
- C** The Carry flag receives the value shifted out of bit 7.

The LSL instruction can be used for simple multiplication (a single left-shift multiplies the value by 2). Other uses include conversion of data from serial to parallel and vise-versa.

The 6309 does not provide variants of LSL to operate on the E and F accumulators. You can however achieve the same functionality using the ADDR instruction. The instructions ADDR E,E and ADDR F,F will perform the same left-shift operation on the E and F accumulators respectively.

The ASL and LSL mnemonics are duplicates. Both produce the same object code.

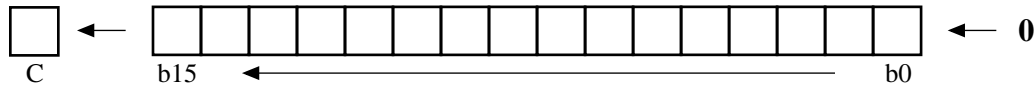
See Also: **LSLD**



# LSLD

6309 ONLY

## Logical Shift Left of Accumulator D



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| LSLD        | INHERENT        | 1048   | 3 / 2  | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

This instruction shifts the contents of Accumulator D to the left by one bit, clearing bit 0. Bit 15 is shifted into the Carry flag of the Condition Codes register.

- N** The Negative flag is set equal to the new value of bit 15; previously bit 14.
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is set to the Exclusive-OR of the original values of bits 14 and 15.
- C** The Carry flag receives the value shifted out of bit 15.

The LSL instruction can be used for simple multiplication (a single left-shift multiplies the value by 2). Other uses include conversion of data from serial to parallel and vice-versa.

The D accumulator is the only 16-bit register for which an LSL instruction has been provided. You can however achieve the same functionality for other 16-bit registers using the ADDR instruction. For example, ADDR W,W will perform the same left-shift operation on the W accumulator.

A left-shift of the 32-bit Q accumulator can be achieved as follows:

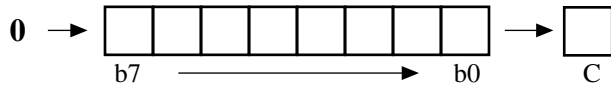
```
ADDR    W,W      ; Shift Low-word, Hi-bit into Carry
ROL     W,W      ; Shift Hi-word, Carry into Low-bit
```

The ASLD and LSLD mnemonics are duplicates. Both produce the same object code.

See Also: **LSL** (8-bit), **ROL** (16-bit)

# LSR (8 Bit)

## Logical Shift Right of 8-Bit Accumulator or Memory Byte



| SOURCE FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| LSRA         | 44       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| LSRB         | 54       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| LSR          |          |       |   | 04     | 6 / 5 | 2 | 64      | 6+ | 2+ | 74       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | 0 | ↓ |   | ↓ |

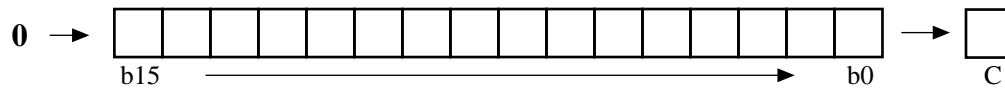
These instructions logically shift the contents of the A or B accumulator or a specified byte in memory to the right by one bit, clearing bit 7. Bit 0 is shifted into the Carry flag of the Condition Codes register.

- N** The Negative flag is cleared by these instructions.
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by these instructions.
- C** The Carry flag receives the value shifted out of bit 0.

The LSR instruction can be used in simple division routines on unsigned values (a single right-shift divides the value by 2).

The 6309 does not provide variants of LSR to operate on the E and F accumulators.

See Also: **LSR** (16-bit)

**Logical Shift Right of 16-Bit Accumulator**

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| LSRD        | INHERENT        | 1044   | 3 / 2  | 2          |
| LSRW        | INHERENT        | 1054   | 3 / 2  | 2          |

|          |          |          |          |          |          |          |          |
|----------|----------|----------|----------|----------|----------|----------|----------|
| <b>E</b> | <b>F</b> | <b>H</b> | <b>I</b> | <b>N</b> | <b>Z</b> | <b>V</b> | <b>C</b> |
|          |          |          |          | 0        | ↕        |          | ↕        |

This instruction shifts the contents of Accumulator D to the right by one bit. Bit 0 is shifted into the Carry flag of the Condition Codes register. The value of bit 15 is not changed.

- N** The Negative flag is cleared by these instructions.
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by this instruction.
- C** The Carry flag receives the value shifted out of bit 0.

These instructions can be used in simple division routines on unsigned values (a single right-shift divides the value by 2).

A logical right-shift of the 32-bit Q accumulator can be achieved as follows:

```

LSRD      ; Shift Hi-word, Low-bit into Carry
RORW      ; Shift Low-word, Carry into Hi-bit

```

See Also: **LSR** (8-bit), **ROR** (16-bit)

# MUL

## Unsigned Multiply of Accumulator A and Accumulator B

$ACCD' \leftarrow ACCA * ACCB$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|-------------|-----------------|--------|---------|------------|
| MUL         | INHERENT        | 3D     | 11 / 10 | 1          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   |   | ↕ |   | ↕ |

This instruction multiplies the unsigned 8-bit value in Accumulator A by the unsigned 8-bit value in Accumulator B. The 16-bit unsigned product is placed into Accumulator D. Only two Condition Code flags are affected:

**Z** The Zero flag is set if the 16-bit result is zero; cleared otherwise.

**C** The Carry flag is set equal to the new value of bit 7 in Accumulator B.

The Carry flag is set equal to bit 7 of the least-significant byte so that rounding of the most-significant byte can be accomplished by executing:

ADCA      #0

See Also: **ADCA**, **MULD**

# MULD

6309 ONLY

## Signed Multiply of Accumulator D and Memory Word

$$ACCQ' \leftarrow ACCD \times IMM16 | (M:M+1)$$

| SOURCE<br>FORMS | IMMEDIATE |    |   | DIRECT |         |   | INDEXED |     |    | EXTENDED |         |   |
|-----------------|-----------|----|---|--------|---------|---|---------|-----|----|----------|---------|---|
|                 | OP        | ~  | # | OP     | ~       | # | OP      | ~   | #  | OP       | ~       | # |
| MULD            | 118F      | 28 | 4 | 119F   | 30 / 29 | 3 | 11AF    | 30+ | 3+ | 11BF     | 31 / 30 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ |   |   |

This instruction multiplies the signed 16-bit value in Accumulator D by either a 16-bit immediate value or the contents of a double-byte value from memory. The signed 32-bit product is placed into Accumulator Q. Only two Condition Code flags are affected:

- N** The Negative flag is set if the twos complement result is negative; cleared otherwise.
- Z** The Zero flag is set if the 32-bit result is zero; cleared otherwise.

See Also: **MUL**

## NEG (accumulator)

### Negation (Twos-Complement) of Accumulator

$$r' \leftarrow 0 - r$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| NEGA        | INHERENT        | 40     | 2 / 1  | 1          |
| NEGB        | INHERENT        | 50     | 2 / 1  | 1          |
| NEGD        | INHERENT        | 1040   | 3 / 2  | 2          |

NEGD is available on 6309 only.

| E | F | H | I | N  | Z  | V  | C  |
|---|---|---|---|----|----|----|----|
|   |   |   |   | ↑↓ | ↑↓ | ↑↓ | ↑↓ |

Each of these instructions change the value of the specified accumulator to that of its twos-complement; that is the value which when added to the original value produces a sum of zero. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of the accumulators high-order bit.
- Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.
- V** The Overflow flag is set if the original value was  $80_{16}$  (8-bit) or  $8000_{16}$  (16-bit); cleared otherwise.
- C** The Carry flag is cleared if the original value was 0; set otherwise.

The operation performed by the NEG instruction can be expressed as:

$$\text{result} = 0 - \text{value}$$

The Carry flag represents a Borrow for this operation and is therefore always set unless the accumulator's original value was zero.

If the original value of the accumulator is  $80_{16}$  ( $8000_{16}$  for NEGD) then the Overflow flag (V) is set and the accumulator's value is not modified.

This instruction performs a twos-complement operation. A ones-complement can be achieved with the COM instruction.

The 6309 does not provide instructions for negating the E, F, W and Q accumulators. A 32-bit negation of Q can be achieved with the following instructions:

```
COMD
COMW
ADCR    0, W
ADCR    0, D
```

See Also: **COM**, **NEG** (memory)

# NEG (memory)

## Negate (Twos Complement) a Byte in Memory

$$(M)' \leftarrow 0 - (M)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| NEG             |           |   |   | 00     | 6 / 5 | 2 | 60      | 6+ | 2+ | 70       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

This instruction changes the value of a byte in memory to that of its twos-complement; that is the value which when added to the original value produces a sum of zero. The Condition Code flags are also modified as follows:

- N** The Negative flag is set equal to the new value of bit 7.
- Z** The Zero flag is set if the new value is zero; cleared otherwise.
- V** The Overflow flag is set if the original value was  $80_{16}$ ; cleared otherwise.
- C** The Carry flag is cleared if the original value was 0; set otherwise.

The operation performed by the NEG instruction can be expressed as:

$$\text{result} = 0 - \text{value}$$

The Carry flag represents a Borrow for this operation and is therefore always set unless the memory byte's original value was zero.

If the original value of the memory byte is  $80_{16}$  then the Overflow flag (V) is set and the byte's value is not modified.

This instruction performs a twos-complement operation. A ones-complement can be achieved with the COM instruction.

See Also: **COM**, **NEG** (accumulator)

# NOP

## No Operation

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| NOP         | INHERENT        | 12     | 2 / 1  | 1          |

E F H I N Z V C

|  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|--|

The NOP instruction advances the Program Counter by one byte without affecting any other registers or condition codes.

The NOP instruction provides a single-byte no-op that consumes two bus cycles (one cycle on a 6309 when NM=1). Some larger, more time-consuming instructions that can also be used as effective no-ops include:

|             |          |           |
|-------------|----------|-----------|
| BRN         | LBRN     |           |
| ANDCC #\$FF | ORCC #0  |           |
| PSHS #0     | PULS #0  |           |
| PSHU #0     | PULU #0  |           |
| EXG r,r     | TFR r,r  |           |
| LEAS ,S     | LEAS ,S+ | LEAS ,S++ |
| LEAU ,U     | LEAU ,U+ | LEAU ,U++ |

See Also: **BRN, EXG, LBRN, LEA, PSH, PUL, TFR**



### Logical OR of Immediate Value with Memory Byte

$(M)' \leftarrow (M) \text{ OR } \text{IMM}$

| SOURCE FORM | IMMEDIATE |   |   | DIRECT |   |   | INDEXED |    |    | EXTENDED |   |   |
|-------------|-----------|---|---|--------|---|---|---------|----|----|----------|---|---|
|             | OP        | ~ | # | OP     | ~ | # | OP      | ~  | #  | OP       | ~ | # |
| OIM #i8;EA  |           |   |   | 01     | 6 | 3 | 61      | 7+ | 3+ | 71       | 7 | 4 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 |   |

The OIM instruction logically ORs the contents of a byte in memory with an 8-bit immediate value. The resulting value is placed back into the designated memory location.

- N** The Negative flag is set equal to the new value of bit 7 of the memory byte.
- Z** The Zero flag is set if the new value of the memory byte is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

OIM is one of the instructions added to the 6309 which allow logical operations to be performed directly in memory instead of having to use an accumulator. It takes three separate instructions to perform the same operation on a 6809:

6809 (6 instruction bytes; 12 cycles):

```
LDA    #$C0
ORA     4,U
STA     4,U
```

6309 (3 instruction bytes; 8 cycles):

```
OIM     #$C0;4,U
```

Note that the assembler syntax used for the OIM operand is non-typical. Some assemblers may require a comma (,) rather than a semicolon (;) between the immediate operand and the address operand.

The object code format for the EIM instruction is:

| OPCODE | IMMED VALUE | ADDRESS / INDEX BYTE(S) |
|--------|-------------|-------------------------|
|--------|-------------|-------------------------|

See Also: **AIM**, **EIM**, **TIM**

## OR (8 Bit)

### Logically OR Accumulator with a Byte from Memory

$$r' \leftarrow r \text{ OR } \text{IMM8} \mid (M)$$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ORA             | 8A        | 2 | 2 | 9A     | 4 / 3 | 2 | AA      | 4+ | 2+ | BA       | 5 / 4 | 3 |
| ORB             | CA        | 2 | 2 | DA     | 4 / 3 | 2 | EA      | 4+ | 2+ | FA       | 5 / 4 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↓ | ↓ | 0 |   |

These instructions logically OR the contents of Accumulator A or B with either an 8-bit immediate value or the contents of a memory byte. The 8-bit result is then placed back in the specified accumulator.

**N** The Negative flag is set equal to the new value of bit 7 of the accumulator.

**Z** The Zero flag is set if the new value of the accumulator is zero; cleared otherwise.

**V** The Overflow flag is cleared by this instruction.

**C** The Carry flag is not affected by this instruction.

The OR instructions are commonly used for setting specific bits in an accumulator to '1' while leaving other bits unchanged. Consider the following examples:

```
ORA    #%00010000    ;Sets bit 4 in A
ORB    #$7F           ;Sets all bits in B except bit 7
```

See Also: **BIOR**, **BOR**, **OIM**, **ORCC**, **ORD**, **ORR**

# ORCC

## Logically OR the CC Register with an Immediate Value

$CC' \leftarrow CC \text{ OR } IMM8$

| SOURCE FORM     | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-----------------|-----------------|--------|--------|------------|
| ORCC <i>#i8</i> | IMMEDIATE       | 1A     | 3      | 2          |

This instruction logically ORs the contents of the Condition Codes register with the 8-bit immediate value specified in the operand. The result is placed back into the Condition Codes register.

The ORCC instruction provides a method to set specific flags in the Condition Codes register. All flags that correspond to '1' bits in the immediate operand are set, while those corresponding with '0's are left unchanged.

The bit numbers for each flag are shown below:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| E | F | H | I | N | Z | V | C |

One of the more common uses for the ORCC instruction is to set the IRQ and FIRQ Interrupt Masks (I and F) at the beginning of a routine that must run with interrupts disabled. This is accomplished by executing:

```
ORCC    #$50        ; Set bits 4 and 6 in CC
```

Some assemblers will accept a comma-delimited list of the bit names as an alternative to the immediate value. For instance, the example above might also be written as:

```
ORCC    I,F          ; Set bits 4 and 6 in CC
```

More examples:

```
ORCC    #1           ; Set the Carry flag
ORCC    #$80         ; Set the Entire flag
```

See Also: **ANDCC**, **OR** (8-bit), **ORD**, **ORR**

# ORD

6309 ONLY

## Logically OR Accumulator D with Word from Memory

$ACCD' \leftarrow ACCD \text{ OR } (M:M+1)$

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| ORD            | 108A      | 5 / 4 | 4 | 109A   | 7 / 5 | 3 | 10AA    | 7+ / 6+ | 3+ | 10BA     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The ORD instruction logically ORs the contents of Accumulator D with a double-byte value from memory. The 16-bit result is placed back into Accumulator D.

- N** The Negative flag is set equal to the new value of bit 15 of Accumulator D.
- Z** The Zero flag is set if the new value of the Accumulator D is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

The ORD instruction is commonly used for setting specific bits in the accumulator to '1' while leaving other bits unchanged.

When using an immediate operand, it is possible to optimize code by determining if the value will only affect half of the accumulator. For example:

ORD       #\$1E00

could be replaced with:

ORA       #\$1E

To ensure that the Negative (N) condition code is set correctly, this optimization must not be made if it would result in an ORB instruction that sets bit 7.

See Also: **BIOR**, **BOR**, **OIM**, **OR** (8-bit), **ORCC**, **ORR**

# ORR

**6309 ONLY**

## Logically OR Source Register with Destination Register

$$r1' \leftarrow r1 \text{ OR } r0$$

| SOURCE FORM      | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------|-----------------|--------|--------|------------|
| ORR <i>r0,r1</i> | IMMEDIATE       | 1035   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The ORR instruction logically ORs the contents of a source register with the contents of a destination register. The result is placed into the destination register.

- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

All of the 6309 registers except Q and MD can be specified as either the source or destination; however specifying the PC register as either the source or destination produces undefined results.

Although the ORR instruction is capable of altering the flow of program execution by specifying the PC register as the destination, you should avoid doing so because the pre-fetch capability of the 6309 can produce un-predictable results.

See “6309 Inter-Register Operations” on page 143 for details on how this instruction operates when registers of different sizes are specified.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. For details, see the description of the **TFR** instruction.

See Also: **OR** (8-bit), **ORD**

# PSH

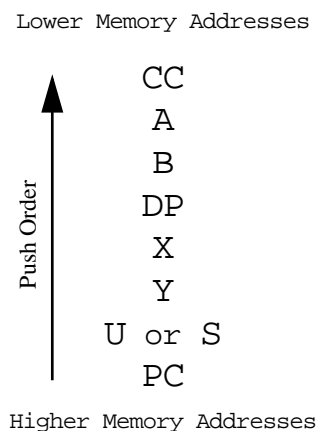
## Push Registers onto a Stack

| SOURCE FORM                                | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|--|-----------------|--------|---------|------------|
| PSHS <i>r0,r1,...rN</i><br>PSHS <i>#i8</i> | IMMEDIATE       | 34     | 5+ / 4+ | 2          |
| PSHU <i>r0,r1,...rN</i><br>PSHU <i>#i8</i> | IMMEDIATE       | 36     | 5+ / 4+ | 2          |

One additional cycle is used for each BYTE pushed.

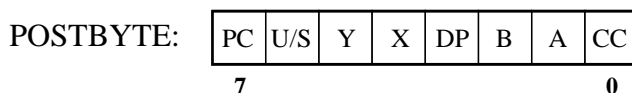
These instructions push the current values of none, one or multiple registers onto either the Hardware (PSHS) or User (PSHU) stack. None of the Condition Code flags are affected by these instructions.

Only the registers present in the 6809 architecture can be pushed by these instructions. Additionally, the stack pointer used by the instruction (S or U) cannot be pushed. Each register specified in the operand field is pushed onto the stack one at a time in the order shown in the figure below (the order you list them in the operand field is irrelevant).



For each 8-bit register specified, the stack pointer is decremented by one and the register's value is stored in the memory location pointed to by the stack pointer. For each 16-bit register specified, the stack pointer is decremented by one, the register's low-order byte is stored, the stack pointer is again decremented by one and the register's high-order byte is then stored.

The PSH instructions use a postbyte wherein each bit position corresponds to one of the registers which may be pushed. Bits that are set (1) specify the registers to be pushed.



See Also: **PSHSW**, **PSHUW**, **PUL**

# PSHSW

6309 ONLY

## Push Accumulator W onto the Hardware Stack

$$\begin{aligned} S' &\leftarrow S - 2 \\ (S:S+1)' &\leftarrow \text{ACCW} \end{aligned}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| PSHSW       | INHERENT        | 1038   | 6      | 2          |

This instruction pushes the contents of the W accumulator (E and F) onto the Hardware Stack (S). None of the Condition Code flags are affected by this instruction.

The PSHSW instruction first decrements hardware stack pointer (S) by one and stores the low-order byte (Accumulator F) at the address pointed to by S. The stack pointer is then decremented by one again, and the high-order byte (Accumulator E) is stored.

This instruction was included in the 6309 instruction set to supplement the PSHS instruction which does not support the W accumulator.

To push either half of the W accumulator onto the hardware stack, you could use the instructions STE , -S or STF , -S, however these instructions will set the Condition Code flags to reflect the pushed value.

See Also: **PSH**, **PSHUW**, **PULSW**, **PULUW**

# PSHUW

6309 ONLY

## Push Accumulator W onto the User Stack

$$\begin{aligned}U' &\leftarrow U - 2 \\(U:U+1)' &\leftarrow \text{ACCW}\end{aligned}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| PSHUW       | INHERENT        | 103A   | 6      | 2          |

This instruction pushes the contents of the W accumulator (E and F) onto the User Stack (U). None of the Condition Code flags are affected by this instruction.

The PSHUW instruction first decrements user stack pointer (U) by one and stores the low-order byte (accumulator F) at the address pointed to by U. The stack pointer is then decremented by one again, and the high-order byte (accumulator E) is stored.

This instruction was included in the 6309 instruction set to supplement the PSHU instruction which does not support the W accumulator.

To push either half of the W accumulator onto the user stack, you could use the instructions STE , -U or STF , -U, however these instructions will set the Condition Code flags to reflect the pushed value.

See Also: **PSH, PSHSW, PULSW, PULUW**



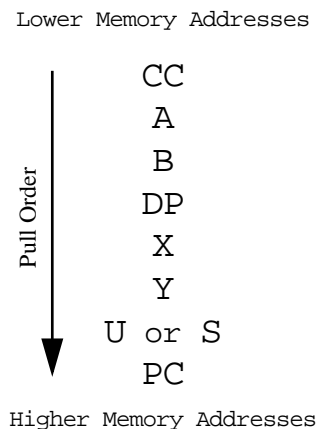
## Pull Registers from Stack

| SOURCE FORM                             | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|---|-----------------|--------|---------|------------|
| PULS $r0, r1, \dots, rN$<br>PULS $\#i8$ | IMMEDIATE       | 35     | 5+ / 4+ | 2          |
| PULU $r0, r1, \dots, rN$<br>PULU $\#i8$ | IMMEDIATE       | 37     | 5+ / 4+ | 2          |

One additional cycle is used for each BYTE pulled.

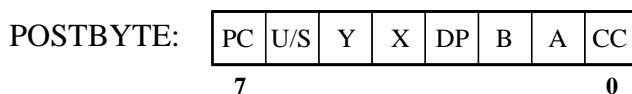
These instructions pull values for none, one or multiple registers from either the Hardware (PULS) or User (PULU) stack. None of the Condition Code flags are affected by these instructions unless the CC register is specified as one of the registers to pull.

Only the registers present in the 6809 architecture can be pulled by these instructions. The stack pointer used by the instruction (S or U) cannot be pulled. A value is pulled from the stack for each register specified in the operand field one at a time in the order shown below (the order you list them in the operand field is irrelevant).



For each 8-bit register specified, a byte is read from the memory location pointed to by the stack pointer and then the stack pointer is incremented by one. For each 16-bit register specified, the register's high-order byte is read from the address pointed to by the stack pointer and then the stack pointer is incremented by one. Next, the register's low-order byte is read and the stack pointer is again incremented by one.

The PUL instructions use a postbyte wherein each bit position corresponds to one of the registers which may be pulled. Bits that are set (1) specify the registers to be pulled.



**See Also: PSH, PULSW, PULUW**

# PULSW

6309 ONLY

## Pull Accumulator W from the Hardware Stack

$$\begin{aligned} \text{ACCW}' &\leftarrow (S:S+1) \\ S' &\leftarrow S + 2 \end{aligned}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| PULSW       | INHERENT        | 1039   | 6      | 2          |

This instruction pulls a value for the W accumulator (E and F) from the Hardware Stack (S). None of the Condition Code flags are affected by this instruction.

The PULSW instruction first loads the high-order byte (Accumulator E) with the value stored at the address pointed to by the hardware stack pointer (S) and increments the stack pointer by one. Next, the low-order byte (Accumulator F) is loaded and the stack pointer is again incremented by one.

This instruction was included in the 6309 instruction set to supplement the PULS instruction which does not support the W accumulator.

To pull either half of the W accumulator from the hardware stack, you could use the instructions `LDE ,S+` or `LDF ,S+`, however these instructions will set the Condition Code flags to reflect the pulled value.

See Also: **PSHSW, PSHUW, PUL, PULUW**

# PULUW

6309 ONLY

## Pull Accumulator W from the User Stack

$$\begin{aligned} \text{ACCW}' &\leftarrow (\text{U}:\text{U}+1) \\ \text{U}' &\leftarrow \text{U} + 2 \end{aligned}$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| PULUW       | INHERENT        | 103B   | 6      | 2          |

This instruction pulls a value for the W accumulator (E and F) from the User Stack (U). None of the Condition Code flags are affected by this instruction.

The PULUW instruction first loads the high-order byte (Accumulator E) with the value stored at the address pointed to by the user stack pointer (U) and increments the stack pointer by one. Next, the low-order byte (Accumulator F) is loaded and the stack pointer is again incremented by one.

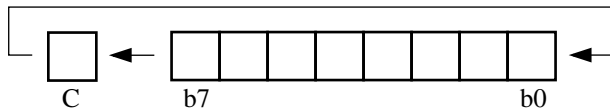
This instruction was included in the 6309 instruction set to supplement the PULU instruction which does not support the W accumulator.

To pull either half of the W accumulator from the user stack, you could use the instructions `LDE ,U+` or `LDF ,U+`, however these instructions will set the Condition Code flags to reflect the pulled value.

See Also: **PSHSW, PSHUW, PUL, PULSW**

# ROL (8 Bit)

## Rotate 8-Bit Accumulator or Memory Byte Left through Carry



| SOURCE<br>FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| ROLA            | 49       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ROLB            | 59       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ROL             |          |       |   | 09     | 6 / 5 | 2 | 69      | 6+ | 2+ | 79       | 7 / 6 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

These instructions rotate the contents of the A or B accumulator or a specified byte in memory to the left by one bit, through the Carry bit of the CC register (effectively a 9-bit rotation). Bit 0 receives the original value of the Carry flag, while the Carry flag receives the original value of bit 7.

- N** The Negative flag is set equal to the new value of bit 7.
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is set equal to the exclusive-OR of the original values of bits 6 and 7.
- C** The Carry flag receives the value shifted out of bit 7.

The ROL instructions can be used for subsequent bytes of a multi-byte shift to bring in the carry bit from previous shift or rotate instructions. Other uses include conversion of data from serial to parallel and vice-versa.

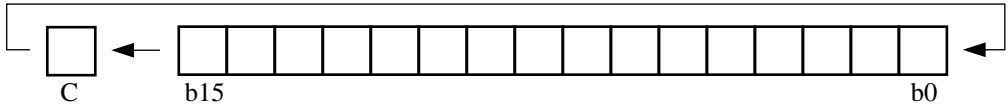
The 6309 does not provide variants of ROL to operate on the E and F accumulators. However, you can achieve the same functionality using the ADCR instruction. The instructions ADCR E,E and ADCR F,F will perform a left-rotate operation on the E and F accumulators respectively.

See Also: **ADCR**, **ROL** (16-bit)

# ROL (16 Bit)

6309 ONLY

## Rotate 16-Bit Accumulator Left through Carry



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| ROLD        | INHERENT        | 1049   | 3 / 2  | 2          |
| ROLW        | INHERENT        | 1059   | 3 / 2  | 2          |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

These instructions rotate the contents of the D or W accumulator to the left by one bit, through the Carry bit of the CC register (effectively a 17-bit rotation). Bit 0 receives the original value of the Carry flag, while the Carry flag receives the original value of bit 15.

- N** The Negative flag is set equal to the new value of bit 15.
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is set equal to the exclusive-OR of the original values of bits 14 and 15.
- C** The Carry flag receives the value shifted out of bit 15.

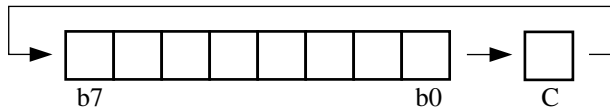
The ROL instructions can be used for subsequent words of a multi-byte shift to bring in the carry bit from a previous shift or rotate instruction. Other uses include conversion of data from serial to parallel and vise-versa.

A left rotate of the 32-bit Q accumulator can be achieved by executing ROLW immediately followed by ROLD.

See Also: **ROL** (8-bit)

## ROR(8 Bit)

### Rotate 8-Bit Accumulator or Memory Byte Right through Carry



| SOURCE FORMS | INHERENT |       |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|----------|-------|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP       | ~     | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| RORA         | 46       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| RORB         | 56       | 2 / 1 | 1 |        |       |   |         |    |    |          |       |   |
| ROR          |          |       |   | 06     | 6 / 5 | 2 | 66      | 6+ | 2+ | 76       | 7 / 6 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ |   | ↕ |

These instructions rotate the contents of the A or B accumulator or a specified byte in memory to the right by one bit, through the Carry bit of the CC register (effectively a 9-bit rotation). Bit 7 receives the original value of the Carry flag, while the Carry flag receives the original value of bit 0.

- N** The Negative flag is set equal to the new value of bit 7 (original value of Carry).
- Z** The Zero flag is set if the new 8-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by these instructions.
- C** The Carry flag receives the value shifted out of bit 0.

The ROR instructions can be used for subsequent bytes of a multi-byte shift to bring in the carry bit from previous shift or rotate instructions. Other uses include conversion of data from serial to parallel and vice-versa.

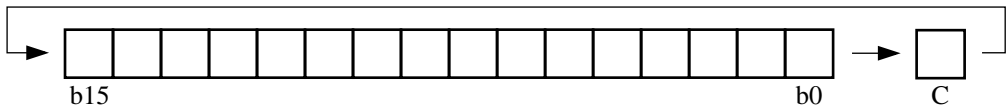
The 6309 does not provide variants of ROR to operate on the E and F accumulators.

See Also: **ROR** (16-bit)

# ROR (16 Bit)

6309 ONLY

## Rotate 16-Bit Accumulator Right through Carry



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| RORD        | INHERENT        | 1046   | 3 / 2  | 2          |
| RORW        | INHERENT        | 1056   | 3 / 2  | 2          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ |   | ↕ |

These instructions rotate the contents of the D or W accumulator to the right by one bit, through the Carry bit of the CC register (effectively a 17-bit rotation). Bit 15 receives the original value of the Carry flag, while the Carry flag receives the original value of bit 0.

- N** The Negative flag is set equal to the new value of bit 15 (original value of Carry).
- Z** The Zero flag is set if the new 16-bit value is zero; cleared otherwise.
- V** The Overflow flag is not affected by these instructions.
- C** The Carry flag receives the value shifted out of bit 0.

The ROR instructions can be used for subsequent words of a multi-byte shift to bring in the carry bit from a previous shift or rotate instruction. Other uses include conversion of data from serial to parallel and vise-versa.

A right rotate of the 32-bit Q accumulator can be achieved by executing RORD immediately followed by RORW.

See Also: **ROR** (8-bit)

# RTI

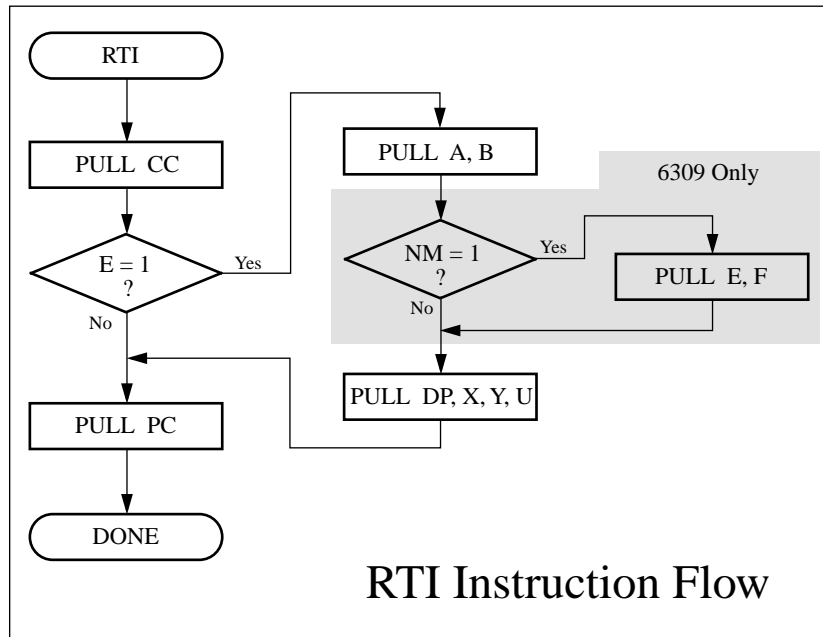
## Return from Interrupt

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES                       | BYTE COUNT |
|-------------|-----------------|--------|------------------------------|------------|
| RTI         | INHERENT        | 3B     | CC.E=0: 6<br>CC.E=1: 15 / 17 | 1          |

The RTI instruction restores the machine state which was stacked upon the invocation of an interrupt service routine.

The exact behavior of the RTI instruction depends on the state of the E flag in the stacked CC register and the state of the NM bit in the MD register.

The E flag will have been set or cleared at the time of the interrupt, based on the type of interrupt that occurred and the state of the FM bit in the MD register at that time.



Interrupt service routines should strive to use the RTI instruction for returning control to the interrupted task. All the logic for proper restoration of the machine state, based on the CPU's current execution mode, is built-in.

When an RTI instruction is executed, the state of the NM bit in the MD register must match the state it was in when the interrupt occurred, otherwise if the E flag was set, the wrong values will be restored to the DP, X, Y, U and PC registers. For this reason, interrupt service routines should avoid changing the NM bit unless they are prepared to deal with this situation.

Service routines which must examine or modify the stacked machine state can require a considerable amount of additional code to determine which registers have been preserved. In particular, the 6309 provides no instruction for testing the state of the NM bit in the MD register (see page 144 for the listing of a subroutine which can accomplish this).

See Also: **CWAI, RTS, SWI, SWI2, SWI3**



# RTS

## Return from Subroutine

$$PC' \leftarrow (S:S+1)$$
$$S' \leftarrow S + 2$$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| RTS         | INHERENT        | 39     | 5 / 4  | 1          |

E F H I N Z V C

|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|
|  |  |  |  |  |  |  |  |
|--|--|--|--|--|--|--|--|

This instruction pulls the double-byte value pointed to by the hardware stack pointer (S) and places it into the PC register. No condition code flags are affected. The effective result is the same as would be achieved using a PULS PC instruction.

RTS is typically used to exit from a subroutine that was called via a BSR or JSR instruction. Note, however, that a subroutine which preserves registers on entry by pushing them onto the stack, may opt to use a single PULS instruction to both restore the registers and return to the caller, as in:

```
ENTRY   PSHS      A,B,X      ; Preserve registers
        ...
        ...
        PULS      A,B,X,PC    ; Restore registers and return
```

See Also: **BSR, JSR, PULS, RTI**

## SBC (8 Bit)

### Subtract Memory Byte and Carry from Accumulator A or B

$$r' \leftarrow r - \text{IMM8} \mid (M) - C$$

| SOURCE FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| SBCA         | 82        | 2 | 2 | 92     | 4 / 3 | 2 | A2      | 4+ | 2+ | B2       | 5 / 4 | 3 |
| SBCB         | C2        | 2 | 2 | D2     | 4 / 3 | 2 | E2      | 4+ | 2+ | F2       | 5 / 4 | 3 |

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   | ~ |   | ↑ | ↑ | ↑ | ↑ |

These instructions subtract either an 8-bit immediate value or the contents of a memory byte, plus the value of the Carry flag from the A or B accumulator. The 8-bit result is placed back into the specified accumulator. Note that since subtraction is performed, the purpose of the Carry flag is to represent a Borrow.

- H** The affect on the Half-Carry flag is undefined for these instructions.
- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into bit-7 was needed; cleared otherwise.

The SBC instruction is most often used to perform subtraction of the subsequent bytes of a multi-byte subtraction. This allows the borrow from a previous SUB or SBC instruction to be included when doing subtraction for the next higher-order byte.

Since the 6809 and 6309 both provide 16-bit SUB instructions for the accumulators, it is not necessary to use the 8-bit SUB and SBC instructions to perform 16-bit subtraction.

See Also: **SBCD**, **SBCR**

# SBCD

6309 ONLY

## Subtract Memory Word and Carry from Accumulator D

$$ACCD' \leftarrow ACCD - IMM16 | (M:M+1) - C$$

| SOURCE<br>FORM | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|----------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| SBCD           | 1082      | 5 / 4 | 4 | 1092   | 7 / 5 | 3 | 10A2    | 7+ / 6+ | 3+ | 10B2     | 8 / 6 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

The SBCD instruction subtracts either a 16-bit immediate value or the contents of a double-byte value in memory, plus the value of the Carry flag from the D accumulator. The 16-bit result is placed back into Accumulator D. Note that since subtraction is performed, the purpose of the Carry flag is to represent a Borrow.

- H** The Half-Carry flag is not affected by the SBCD instruction.
- N** The Negative flag is set equal to the new value of bit 15 of Accumulator D.
- Z** The Zero flag is set if the new value of Accumulator D is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into bit 15 was needed; cleared otherwise.

The SBCD instruction is most often used to perform subtraction of subsequent words of a multi-byte subtraction. This allows the borrow from a previous SUB or SBC instruction to be included when doing subtraction for the next higher-order word.

The following instruction sequence is an example showing how 32-bit subtraction can be performed on a 6309 microprocessor:

```
LDQ      VAL1ADR      ; Q = 32-bit minuend
SUBW     VAL2ADR+2     ; Subtract lower half of subtrahend
SBCD     VAL2ADR       ; Subtract upper half of subtrahend
STQ      RESULT        ; Store difference
```

See Also: **SBC** (8-bit), **SBCR**

# SBCR

6309 ONLY

## Subtract Source Register and Carry from Destination Register

$$r1' \leftarrow r1 - r0 - C$$

| SOURCE FORM       | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------------|-----------------|--------|--------|------------|
| SBCR <i>r0,r1</i> | IMMEDIATE       | 1033   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | ↕ | ↕ |

The SBCR instruction subtracts the contents of a source register plus the value of the Carry flag from the contents of a destination register. The result is placed into the destination register.

- H** The Half-Carry flag is not affected by the SBCR instruction.
- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into the high-order bit was needed; cleared otherwise.

All of the 6309 registers except Q and MD can be specified as either the source or destination; however specifying the PC register as either the source or destination produces undefined results.

The SBCR instruction will perform either 8-bit or 16-bit subtraction according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See “6309 Inter-Register Operations” on page 143 for further details.

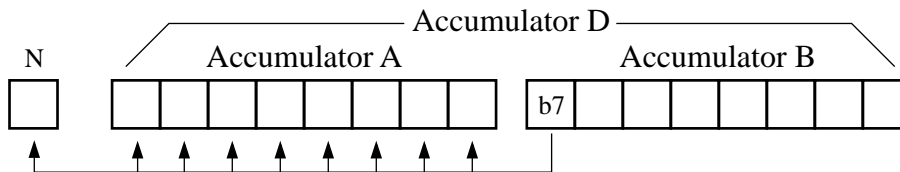
Although the SBCR instruction is capable of altering the flow of program execution by specifying the PC register as the destination, you should avoid doing so because the pre-fetch capability of the 6309 can produce un-predictable results.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. See the description of the **TFR** instruction for further details.

See Also: **SBC** (8-bit), **SBCD**

# SEX

## Sign Extend the 8-bit Value in B to a 16-bit Value in D



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| SEX         | INHERENT        | 1D     | 2 / 1  | 1          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ |   |   |

This instruction extends the 8-bit twos complement value in Accumulator B into a 16-bit twos complement value in Accumulator D. This is accomplished by copying the value of bit 7 (the sign bit) from Accumulator B into all 8 bits of Accumulator A.

- N** The Negative flag is also set equal the value of bit 7 in Accumulator B
- Z** The Zero flag is set if the new value of Accumulator D is zero (B was zero); cleared otherwise.
- V** The Overflow flag is not affected by this instruction.
- C** The Carry flag is not affected by this instruction.

The SEX instruction is used when a signed (twos complement) 8-bit value needs to be promoted to a full 16-bit value. For unsigned arithmetic, promoting an 8-bit value in Accumulator A to a 16-bit value in Accumulator D requires zero-extending the value by executing a CLRA instruction instead.

On a 6309, you can sign extend an 8-bit value in Accumulator A to a 32-bit value in Accumulator Q by executing the following sequence of instructions:

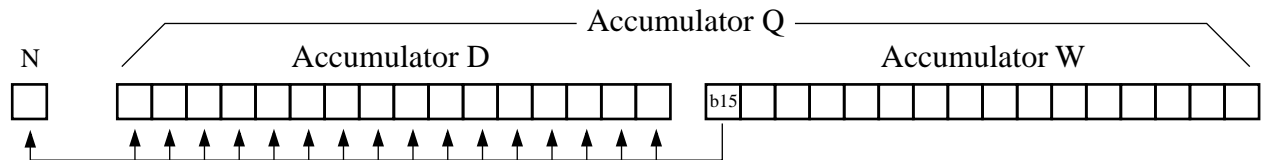
```
SEX          ; Sign extend A into D
TFR    D,W   ; Move D to W
SEXW        ; Sign extend W into Q
```

See Also: **SEXW**

# SEXW

6309 ONLY

## Sign Extend a 16-bit Value in W to a 32-bit Value in Q



| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| SEXW        | INHERENT        | 14     | 4      | 1          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ |   |   |

This instruction extends the 16-bit twos complement value in Accumulator W into a 32-bit twos complement value in Accumulator Q. This is accomplished by copying the value of bit 15 (the sign bit) from Accumulator W into all 16 bits of Accumulator D.

- N** The Negative flag is also set equal the value of bit 15 in Accumulator W
- Z** The Zero flag is set if the new value of Accumulator Q is zero (W was zero); cleared otherwise.
- V** The Overflow flag is not affected by this instruction.
- C** The Carry flag is not affected by this instruction.

The SEXW instruction is used when a signed (twos complement) 16-bit value needs to be promoted to a full 32-bit value. For unsigned arithmetic, promoting a 16-bit value in Accumulator W to a 32-bit value in Accumulator Q requires zero-extending the value by executing a CLRD instruction instead.

You can sign extend an 8-bit value in Accumulator A to a 32-bit value in Accumulator Q by executing the following sequence of instructions:

```
SEX          ; Sign extend A into D
TFR    D,W   ; Move D to W
SEXW        ; Sign extend W into Q
```

See Also: **SEX**

# ST (8 Bit)

## Store 8-Bit Accumulator to Memory

$(M)' \leftarrow r$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| STA             |           |   |   | 97     | 4 / 3 | 2 | A7      | 4+ | 2+ | B7       | 5 / 4 | 3 |
| STB             |           |   |   | D7     | 4 / 3 | 2 | E7      | 4+ | 2+ | F7       | 5 / 4 | 3 |
| STE             |           |   |   | 1197   | 5 / 4 | 3 | 11A7    | 5+ | 3+ | 11B7     | 6 / 5 | 4 |
| STF             |           |   |   | 11D7   | 5 / 4 | 3 | 11E7    | 5+ | 3+ | 11F7     | 6 / 5 | 4 |

STE and STF are available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions store the contents of one of the 8-bit accumulators (A,B,E,F) into a byte in memory. The Condition Codes are affected as follows.

- N** The Negative flag is set equal to the value of bit 7 of the accumulator.
- Z** The Zero flag is set if the accumulator's value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by these instructions.

See Also: **ST** (16-bit), **STQ**

# ST (16 Bit)

## Store 16-Bit Register to Memory

$(M:M+1)' \leftarrow r$

| SOURCE FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| STD          |           |   |   | DD     | 5 / 4 | 2 | ED      | 5+ | 2+ | FD       | 6 / 5 | 3 |
| STS          |           |   |   | 10DF   | 6 / 5 | 3 | 10EF    | 6+ | 3+ | 10FF     | 7 / 6 | 4 |
| STU          |           |   |   | DF     | 5 / 4 | 2 | EF      | 5+ | 2+ | FF       | 6 / 5 | 3 |
| STW          |           |   |   | 1097   | 6 / 5 | 3 | 10A7    | 6+ | 3+ | 10B7     | 7 / 6 | 4 |
| STX          |           |   |   | 9F     | 5 / 4 | 2 | AF      | 5+ | 2+ | BF       | 6 / 5 | 3 |
| STY          |           |   |   | 109F   | 6 / 5 | 3 | 10AF    | 6+ | 3+ | 10BF     | 7 / 6 | 4 |

STW is available on 6309 only.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

These instructions store the contents of one of the 16-bit accumulators (D,W) or one of the 16-bit Index/Stack registers (X,Y,U,S) to a pair of memory bytes in big-endian order. The Condition Codes are affected as follows:

- N** The Negative flag is set equal to the value in bit 15 of the register.
- Z** The Zero flag is set if the register value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by these instructions.

See Also: **ST** (8-bit), **STQ**



# STBT

6309 ONLY

## Store value of a Register Bit into Memory

$(DPM).dstBit' \leftarrow r.srcBit$

| SOURCE FORM                  | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|------------------------------|-----------------|--------|--------|------------|
| STBT <i>r,sBit,dBit,addr</i> | DIRECT          | 1137   | 8 / 7  | 4          |

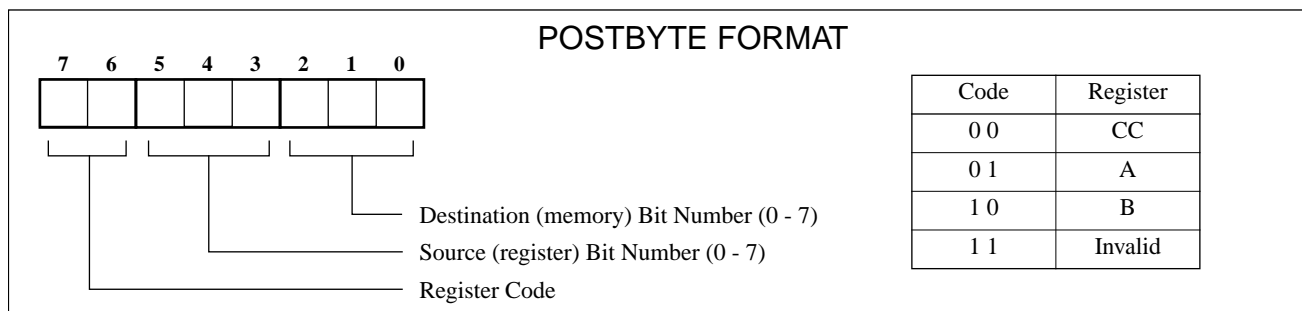
The STBT instruction stores the value of a specified bit in either the A, B or CC registers to a specified bit in memory. None of the Condition Code flags are affected by the operation. The usefulness of the STBT instruction is limited by the fact that only Direct Addressing is permitted.



The figure above shows an example of the STBT instruction where bit 5 from Accumulator A is stored into bit 1 of memory location \$0040 (DP = 0).

The object code format for the STBT instruction is:

|      |      |          |             |
|------|------|----------|-------------|
| \$11 | \$37 | POSTBYTE | ADDRESS LSB |
|------|------|----------|-------------|



See Also: **BAND, BEOR, BIAND, BIEOR, BIOR, BOR, LDBT**

# STQ

6309 ONLY

## Store Contents of Accumulator Q to Memory

$(M:M+3)' \leftarrow Q$

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| STQ             |           |   |   | 10DD   | 8 / 7 | 3 | 10ED    | 8+ | 3+ | 10FD     | 9 / 8 | 4 |

E F H I N Z V C

|  |  |  |  |   |   |   |  |
|--|--|--|--|---|---|---|--|
|  |  |  |  | ↑ | ↑ | 0 |  |
|--|--|--|--|---|---|---|--|

This instruction stores the contents of the Q accumulator into 4 sequential bytes of memory in big-endian order. The Condition Codes are affected as follows.

- N** The Negative flag is set equal to the value of bit 31 of Accumulator Q.
- Z** The Zero flag is set if the value of Accumulator Q is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by this instruction.

See Also: **ST** (8-bit), **ST** (16-bit)

# SUB (8 Bit)

## Subtract from value in 8-Bit Accumulator

$$r' \leftarrow r - \text{IMM8} \mid (M)$$

| SOURCE FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |    |    | EXTENDED |       |   |
|--------------|-----------|---|---|--------|-------|---|---------|----|----|----------|-------|---|
|              | OP        | ~ | # | OP     | ~     | # | OP      | ~  | #  | OP       | ~     | # |
| SUBA         | 80        | 2 | 2 | 90     | 4 / 3 | 2 | A0      | 4+ | 2+ | B0       | 5 / 4 | 3 |
| SUBB         | C0        | 2 | 2 | D0     | 4 / 3 | 2 | E0      | 4+ | 2+ | F0       | 5 / 4 | 3 |
| SUBE         | 1180      | 3 | 3 | 1190   | 5 / 4 | 3 | 11A0    | 5+ | 3+ | 11B0     | 6 / 5 | 4 |
| SUBF         | 11C0      | 3 | 3 | 11D0   | 5 / 4 | 3 | 11E0    | 5+ | 3+ | 11F0     | 6 / 5 | 4 |

SUBE and SUBF are available on 6309 only.

E F H I N Z V C

|  |  |   |  |   |   |   |   |
|--|--|---|--|---|---|---|---|
|  |  | ~ |  | ↑ | ↑ | ↑ | ↑ |
|--|--|---|--|---|---|---|---|

These instructions subtract either an 8-bit immediate value or the contents of a byte in memory from one of the 8-bit accumulators (A,B,E,F). The 8-bit result is placed back into the specified accumulator. Note that since subtraction is performed, the purpose of the Carry flag is to represent a Borrow.

- H** The value of Half-Carry flag is undefined after executing these instructions.
- N** The Negative flag is set equal to the new value of bit 7 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into bit 7 was needed; cleared otherwise.

The 8-bit SUB instructions are used for single-byte subtraction, and for subtraction of the least-significant byte in multi-byte subtractions. Since the 6809 and 6309 both provide 16-bit SUB instructions for the accumulators, it is not necessary to use the 8-bit SUB and SBC instructions to perform 16-bit subtraction.

See Also: **SUB** (16-bit), **SUBR**

# SUB (16 Bit)

## Subtract from value in 16-Bit Accumulator

$$r' \leftarrow r - \text{IMM16} \mid (M:M+1)$$

| SOURCE FORMS | IMMEDIATE |       |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|--------------|-----------|-------|---|--------|-------|---|---------|---------|----|----------|-------|---|
|              | OP        | ~     | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| SUBD         | 83        | 4 / 3 | 3 | 93     | 6 / 4 | 2 | A3      | 6+ / 5+ | 2+ | B3       | 7 / 5 | 3 |
| SUBW         | 1080      | 5 / 4 | 4 | 1090   | 7 / 5 | 3 | 10A0    | 7+ / 6+ | 3+ | 10B0     | 8 / 6 | 4 |

SUBW is available on 6309 only.

| E | F | H | I | N | Z | V | C |
|---|---|---|---|---|---|---|---|
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

These instructions subtract either a 16-bit immediate value or the contents of a double-byte value in memory from one of the 16-bit accumulators (D,W). The 16-bit result is placed back into the specified accumulator. Note that since subtraction is performed, the purpose of the Carry flag is to represent a Borrow.

- H** The Half-Carry flag is not affected by these instructions.
- N** The Negative flag is set equal to the new value of bit 15 of the accumulator.
- Z** The Zero flag is set if the new accumulator value is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow out of bit 7 was needed; cleared otherwise.

The 16-bit SUB instructions are used for 16-bit subtraction, and for subtraction of the least-significant word of multi-byte subtractions. See the description of the **SBCD** instruction for an example of how 32-bit subtraction can be performed on a 6309.

See Also: **SUB** (8-bit), **SUBR**

# SUBR

6309 ONLY

## Subtract Source Register from Destination Register

$$r1' \leftarrow r1 - r0$$

| SOURCE FORM   | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|---------------|-----------------|--------|--------|------------|
| SUBR $r0, r1$ | IMMEDIATE       | 1032   | 4      | 3          |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | ↑ | ↑ |

The SUBR instruction subtracts the value contained in the source register from the value contained in the destination register. The result is placed into the destination register. Note that since subtraction is performed, the purpose of the Carry flag is to represent a Borrow.

- H** The Half-Carry flag is not affected by the SUBR instruction.
- N** The Negative flag is set equal to the value of the result's high-order bit.
- Z** The Zero flag is set if the new value of the destination register is zero; cleared otherwise.
- V** The Overflow flag is set if an overflow occurred; cleared otherwise.
- C** The Carry flag is set if a borrow into the high-order bit was needed; cleared otherwise.

All of the 6309 registers except Q and MD can be specified as either the source or destination; however specifying the PC register as either the source or destination produces undefined results.

The SUBR instruction will perform either 8-bit or 16-bit subtraction according to the size of the destination register. When registers of different sizes are specified, the source will be promoted, demoted or substituted depending on the size of the destination and on which specific 8-bit register is involved. See "6309 Inter-Register Operations" on page 143 for further details.

Although the SUBR instruction is capable of altering the flow of program execution by specifying the PC register as the destination, you should avoid doing so because the pre-fetch capability of the 6309 can produce un-predictable results.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. See the description of the **TFR** instruction for further details.

See Also: **SUB** (8-bit), **SUB** (16-bit)

# SWI

## Software Interrupt

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES  | BYTE COUNT |
|-------------|-----------------|--------|---------|------------|
| SWI         | INHERENT        | 3F     | 19 / 21 | 1          |
| SWI2        | INHERENT        | 103F   | 20 / 22 | 2          |
| SWI3        | INHERENT        | 113F   | 20 / 22 | 2          |

The SWI, SWI2 and SWI3 instructions each invoke a Software Interrupt.

Each of these instructions first set the E flag in the CC register and then push the machine state onto the hardware stack (S).

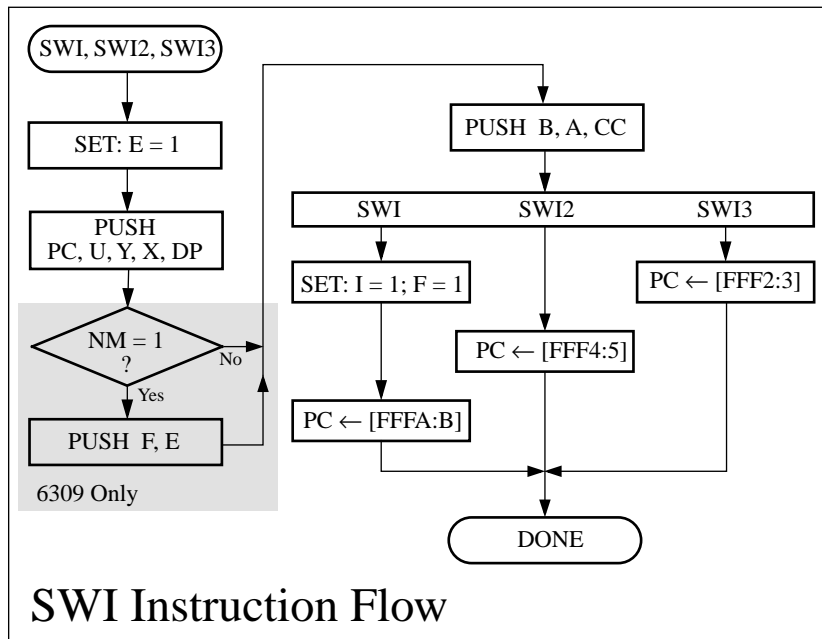
After stacking the machine state, the SWI instruction sets the I and F interrupt masks in the CC register. SWI2 and SWI3 do not modify the mask.

Finally, control is transferred to the interrupt service routine whose address is obtained from the vector which corresponds to the particular instruction.

The state of the NM bit in the MD register determines whether or not the E and F accumulators are included in the stacked machine state. Service routines should be written to work properly regardless of the current state of the NM bit. This is best accomplished by avoiding modification of the NM bit and using the RTI instruction to return control to the interrupted task. If an SWI service routine needs to examine or modify the stacked machine state, it may first need to determine the current state of the NM bit. See page 144 for the listing of a subroutine that will accomplish this task.

**NOTE:** When Motorola introduced the 6809, they designated SWI2 as an instruction reserved for the end user, and not to be used in packaged software. Under the OS9 operating system, SWI2 is used to invoke *Service Requests*.

See Also: **RTI**



# SYNC

## Synchronize with Interrupt

Halt Execution and Wait for Interrupt

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES              | BYTE COUNT |
|-------------|-----------------|--------|---------------------|------------|
| SYNC        | IMMEDIATE       | 13     | $\geq 4$ / $\geq 3$ | 1          |

The SYNC instruction allows software to synchronize itself with an external hardware event (interrupt). When executed, SYNC places the CPU's data and address busses into a high-impedance state, stops executing instructions and waits for an interrupt. None of the Condition Code flags are directly affected by this instruction.

When a signal is asserted on any one of the CPU's 3 interrupt lines (IRQ, FIRQ or NMI), the CPU clears the synchronizing state and resumes processing. If the interrupt type is not masked and the interrupt signal remains asserted for at least 3 cycles, then the CPU will stack the machine state accordingly and vector to the interrupt service routine. If the interrupt type is masked, or the interrupt signal was asserted for less than 3 cycles, then the CPU will simply resume execution at the following instruction without invoking the interrupt service routine.

Typically, SYNC is executed with interrupts masked so that the following instruction will be executed as quickly as possible after the synchronizing event occurs (no service routine overhead). Unlike CWAI, the SYNC instruction does not include the ability to set or clear the interrupt masks as part of its operation. A separate ORCC or ANDCC instruction would be needed to accomplish this.

SYNC may be useful for synchronizing with a video display or for performing fast data acquisition from an I/O device.

See Also: **ANDCC**, **CWAI**, **RTI**, **SYNC**

## Transfer Memory

| SOURCE FORM    | ADDRESSING MODE | OPCODE | CYCLES   | BYTE COUNT |
|----------------|-----------------|--------|----------|------------|
| TFM $r0+, r1+$ | IMMEDIATE       | 1138   | $6 + 3n$ | 3          |
| TFM $r0-, r1-$ | IMMEDIATE       | 1139   | $6 + 3n$ | 3          |
| TFM $r0+, r1$  | IMMEDIATE       | 113A   | $6 + 3n$ | 3          |
| TFM $r0, r1+$  | IMMEDIATE       | 113B   | $6 + 3n$ | 3          |

Three additional cycles are used for each BYTE transferred.

The TFM instructions transfer the number of bytes specified in the W accumulator from a source address pointed to by the X, Y, U, S or D registers to a destination address also pointed to by one of those registers. After each byte is transferred the source and destination registers may both be incremented by one, both decremented by one, only the source incremented, or only the destination incremented. Accumulator W is always decremented by one after each byte is transferred. The instruction completes when W is decremented to 0.

The forms which increment or decrement both addresses provide a block-move operation. Typically, the decrementing form is needed when the source block resides at a lower address than the destination block AND the two blocks may overlap each other.

The forms which increment only one of the addresses are useful for filling a block of memory with a particular byte value (destination increments), and for reading or writing a block of data from or to a memory-mapped I/O device. For the reasons described below, I/O transfers should always be performed with interrupts masked.

The Immediate operand for this instruction is a postbyte which uses the same format as that used by the TFR and EXG instructions. An Illegal Instruction exception will occur if the postbyte contains encodings for registers other than X, Y, U, S or D.

### IMPORTANT:

The TFM instructions are unique in that they are the only instructions that may be interrupted before they have completed. If an unmasked interrupt occurs while executing a TFM instruction, the CPU will interrupt the operation at a point where it has read a byte from the source address, but before it has incremented or decremented any registers or stored the byte at the destination address. The interrupt service routine will be invoked in the normal manner except for the fact that the PC value pushed onto the stack will still point to the TFM instruction. This causes the TFM instruction to be executed again when the service routine returns. Since the address registers were not updated prior to the invocation of the service routine, TFM will start by reading a byte from the previous source address for a second time.

It is also important to remember that in emulation mode (NM=0), the W register is not automatically preserved. If a service routine modifies W but does not explicitly preserve its original value, it could alter the actual number of bytes processed by a TFM instruction.



# TFR

## 6309 IMPLEMENTATION

### Transfer Register to Register

$r0 \rightarrow r1$

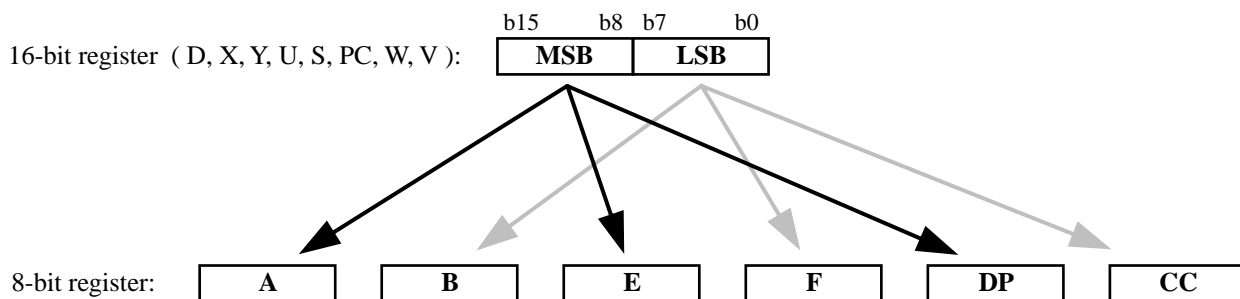
| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| TFR $r0,r1$ | IMMEDIATE       | 1F     | 6 / 4  | 2          |

TFR copies the contents of a source register into a destination register. None of the Condition Code flags are affected unless CC is specified as the destination register.

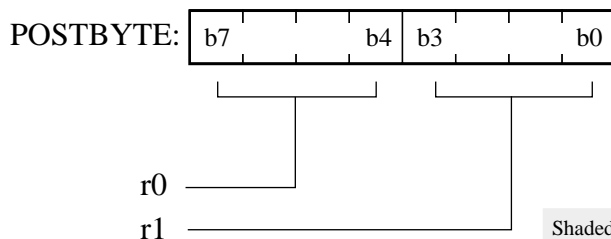
Any of the 6309 registers except Q and MD may be specified as either the source, destination or both. Specifying the same register for both the source and destination produces an instruction which, like NOP, has no effect.

The TFR instruction can be used to alter the flow of execution by specifying PC as the destination register.

When an 8-bit source register is transferred to a 16-bit destination register, the contents of the 8-bit register are placed into both halves of the 16-bit register. When a 16-bit source register is transferred to an 8-bit destination register, only the upper or the lower half of the 16-bit register is transferred. As illustrated in the diagram below, which half is transferred depends on which 8-bit register is specified as the destination.



The TFR instruction requires a postbyte in which the source and destination registers are encoded into the upper and lower nibbles respectively.



Shaded encodings are invalid on 6809 microprocessors

| Code | Register | Code | Register |
|------|----------|------|----------|
| 0000 | D        | 1000 | A        |
| 0001 | X        | 1001 | B        |
| 0010 | Y        | 1010 | CC       |
| 0011 | U        | 1011 | DP       |
| 0100 | S        | 1100 | 0        |
| 0101 | PC       | 1101 | 0        |
| 0110 | W        | 1110 | E        |
| 0111 | V        | 1111 | F        |

See Also: **EXG**, **TFR** (6809 implementation)

# TFR

## 6809 IMPLEMENTATION

### Transfer Register to Register

$r0 \rightarrow r1$

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| TFR $r0,r1$ | IMMEDIATE       | 1F     | 6      | 2          |

TFR copies the contents of a source register into a destination register. None of the Condition Code flags are affected unless CC is specified as the destination register.

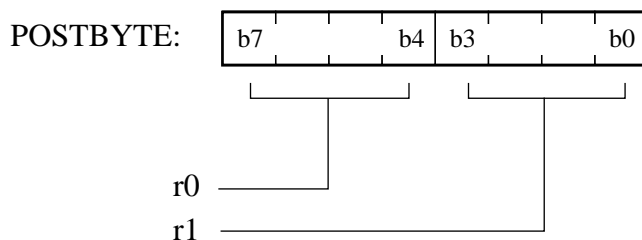
The TFR instruction can be used to alter the flow of execution by specifying PC as the destination register.

Any of the 6809 registers may be specified as either the source, destination or both. Specifying the same register for both the source and destination produces an instruction which, like NOP, has no effect.

The table below explains how the destination register is affected when the source and destination sizes are different. This behavior differs from the 6309 implementation.

| Operation          | 8-bit Register Used | Results                                       |
|--------------------|---------------------|---|
| $16 \rightarrow 8$ | Any                 | Destination = LSB from Source                 |
| $8 \rightarrow 16$ | A or B              | MSB of Destination = $FF_{16}$ ; LSB = Source |
| $8 \rightarrow 16$ | CC or DP            | Both MSB and LSB of Destination = Source      |

The TFR instruction requires a postbyte in which the source and destination registers are encoded into the upper and lower nibbles respectively.



| Code | Register       | Code | Register       |
|------|----------------|------|----------------|
| 0000 | D              | 1000 | A              |
| 0001 | X              | 1001 | B              |
| 0010 | Y              | 1010 | CC             |
| 0011 | U              | 1011 | DP             |
| 0100 | S              | 1100 | <i>invalid</i> |
| 0101 | PC             | 1101 | <i>invalid</i> |
| 0110 | <i>invalid</i> | 1110 | <i>invalid</i> |
| 0111 | <i>invalid</i> | 1111 | <i>invalid</i> |

If an invalid register encoding is used for the source, a constant value of  $FF_{16}$  or  $FFFF_{16}$  is transferred to the destination. If an invalid register encoding is used for the destination, then the instruction will have no effect. **The invalid register encodings have valid meanings when executed on 6309 processors, and should be avoided in code that needs to work the same way on both CPU's.**

See Also: **EXG**, **TFR** (6309 implementation)

# TIM

6309 ONLY

## Bit Test Immediate Value with Memory Byte

TEMP  $\leftarrow$  (M) AND IMM8

| SOURCE<br>FORM | IMMEDIATE |   |   | DIRECT |   |   | INDEXED |    |    | EXTENDED |   |   |
|----------------|-----------|---|---|--------|---|---|---------|----|----|----------|---|---|
|                | OP        | ~ | # | OP     | ~ | # | OP      | ~  | #  | OP       | ~ | # |
| TIM #i8;EA     |           |   |   | 0B     | 6 | 3 | 6B      | 7+ | 3+ | 7B       | 7 | 4 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

The TIM instruction logically ANDs the contents of a byte in memory with an 8-bit immediate value. The resulting value is tested and then discarded. The Condition Codes are updated to reflect the results of the test as follows:

- N** The Negative flag is set equal to bit 7 of the resulting value.
- Z** The Zero flag is set if the resulting value was zero; cleared otherwise.
- V** The Overflow flag is cleared by this instruction.
- C** The Carry flag is not affected by this instruction.

TIM can be used as a space-saving optimization for a pair of equivalent 6809 instructions, and to perform a bit test without having to utilize a register. However, it is slower than the 6809 equivalent:

6809: (4 instruction bytes; 7 cycles):

```
LDA    #$3F
BITA   4,U
```

6309: (3 instruction bytes; 8 cycles):

```
TIM    #$3F;4,U
```

Note that the assembler syntax used for the TIM operand is non-typical. Some assemblers may require a comma (,) rather than a semicolon (;) between the immediate operand and the address operand.

The object code format for the TIM instruction is:

|        |             |                         |
|--------|-------------|-------------------------|
| OPCODE | IMMED VALUE | ADDRESS / INDEX BYTE(S) |
|--------|-------------|-------------------------|

See Also: **AIM**, **AND**, **EIM**, **OIM**

# TST (accumulator)

## Test Value in Accumulator

TEMP  $\leftarrow$  r

| SOURCE FORM | ADDRESSING MODE | OPCODE | CYCLES | BYTE COUNT |
|-------------|-----------------|--------|--------|------------|
| TSTA        | INHERENT        | 4D     | 2 / 1  | 1          |
| TSTB        | INHERENT        | 5D     | 2 / 1  | 1          |
| TSTD        | INHERENT        | 104D   | 3 / 2  | 2          |
| TSTE        | INHERENT        | 114D   | 3 / 2  | 2          |
| TSTF        | INHERENT        | 115D   | 3 / 2  | 2          |
| TSTW        | INHERENT        | 105D   | 3 / 2  | 2          |

TSTD, TSTE, TSTF and TSTW are available on 6309 only.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↕ | ↕ | 0 |   |

The TST instructions test the value in an accumulator to setup the Condition Codes register with minimal status for that value. The accumulator itself is not modified by these instructions.

- N** The Negative flag is set equal to the value of the accumulator's high-order bit (sign bit).
- Z** The Zero flag is set if the accumulator's value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by these instructions.

For unsigned values, the only meaningful information provided is whether or not the value is zero. In this case, BEQ or BNE would typically follow such a test.

For signed (twos complement) values, the information provided is sufficient to allow any of the signed conditional branches (BGE, BGT, BLE, BLT) to be used as though the accumulator's value had been compared with zero. You can also use BMI and BPL to branch according to the sign of the value.

To determine the sign of a 16-bit or 32-bit value, you only need to test the high order byte. For example, TSTA is sufficient for determining the sign of a 32-bit twos complement value in accumulator Q. A full test of accumulator Q could be accomplished by storing it to a scratchpad RAM location (or ROM address). In a traditional stack environment, the instruction STQ -4, S may be acceptable.

See Also: **CMP**, **STQ**, **TST** (memory)

# TST (memory)

## Test Value in Memory Byte

TEMP ← (M)

| SOURCE<br>FORMS | IMMEDIATE |   |   | DIRECT |       |   | INDEXED |         |    | EXTENDED |       |   |
|-----------------|-----------|---|---|--------|-------|---|---------|---------|----|----------|-------|---|
|                 | OP        | ~ | # | OP     | ~     | # | OP      | ~       | #  | OP       | ~     | # |
| TST             |           |   |   | 0D     | 6 / 4 | 2 | 6D      | 6+ / 5+ | 2+ | 7D       | 7 / 5 | 3 |

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| E | F | H | I | N | Z | V | C |
|   |   |   |   | ↑ | ↑ | 0 |   |

The TST instructions test the value in a memory byte to setup the Condition Codes register with minimal status for that value. The memory byte is not modified.

- N** The Negative flag is set equal to bit 7 of the byte's value (sign bit).
- Z** The Zero flag is set if the byte's value is zero; cleared otherwise.
- V** The Overflow flag is always cleared.
- C** The Carry flag is not affected by this instruction.

For unsigned values, the only meaningful information provided is whether or not the value is zero. In this case, BEQ or BNE would typically follow such a test.

For signed (twos complement) values, the information provided is sufficient to allow any of the signed conditional branches (BGE, BGT, BLE, BLT) to be used as though the byte's value had been compared with zero. You could also use BMI and BPL to branch according to the sign of the value.

You can obtain the same information in fewer cycles by loading the byte into an 8-bit accumulator (LDA and LDB are fastest). For this reason it is usually preferable to avoid using TST on a memory byte if there is an available accumulator.

See Also: **CMP**, **LD** (8-bit), **TST** (accumulator)

# **Part II**

## **6309 Specifics**

# 6309 Inter-Register Operations

The 6309 microprocessor adds several new instructions which operate directly on a pair of register operands. The operations provided are addition, subtraction, bitwise AND, bitwise OR, bitwise Exclusive-OR, and comparison. There are two forms of addition and subtraction operations to allow for inclusion or exclusion of the Carry bit.

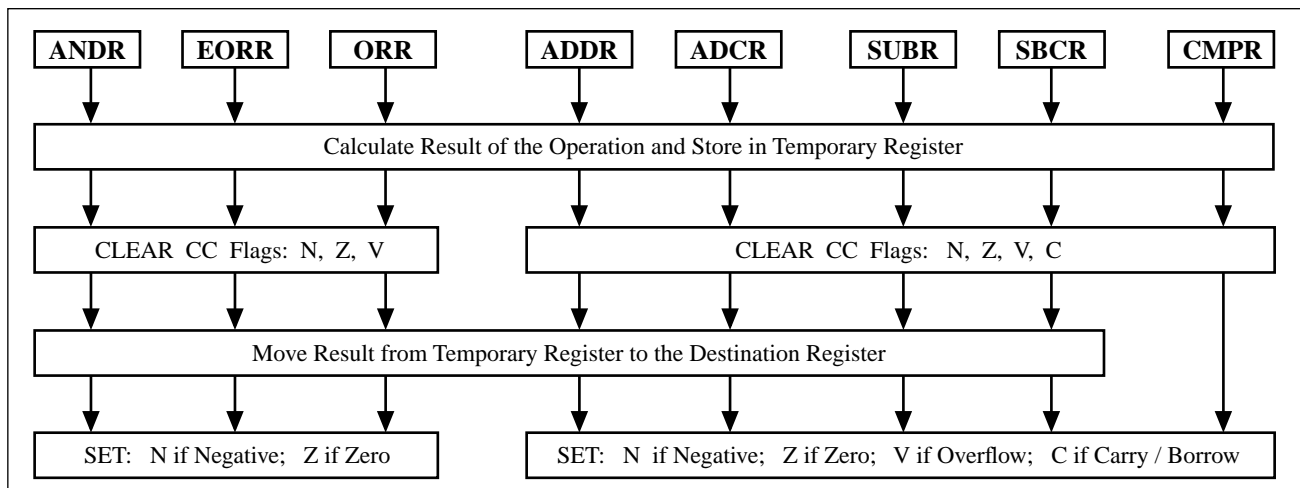
|      |      |      |      |
|------|------|------|------|
| ADCR | ADDR | ANDR | CMPR |
| EORR | ORR  | SBCR | SUBR |

Any of the 6309's registers except Q and MD may be used in the inter-register instructions as either the source operand, destination operand or both. Although the PC register can be used in these instructions, it is not advised. The pipelining performed by the 6309 is not properly synchronized for these instructions. This causes the actual PC value used in these operations to be unpredictable. This flaw affects only the new inter-register instructions in the 6309 instruction set. Using PC in a TFR or EXG instruction functions correctly, as on the 6809.

The inter-register instructions will perform either an 8-bit or 16-bit operation according to the size of the destination register. If the sizes of the source and destination registers differ then the source operand will either be promoted or demoted as shown in the table below.

| Destination Size | Source Register     | Actual Source Operand                    |
|------------------|---------------------|--|
| 8 bits           | Any 16-bit Register | Lower 8 bits of 16-bit Source            |
| 16 bits          | A or B              | Accumulator D                            |
| 16 bits          | E or F              | Accumulator W                            |
| 16 bits          | CC                  | Zero in upper 8 bits; CC in lower 8 bits |
| 16 bits          | DP                  | DP in upper 8 bits; Zero in lower 8 bits |

Using CC as the destination operand for instructions other than CMPR can be problematic. This is due to the fact that not only is the resulting value of the operation stored in CC, but so too are the status bits which reflect that result. The diagram below illustrates the order in which the internal processing steps occur.



# Determining the 6309 Execution Mode

The BITMD instruction cannot be used to test the state of the two execution mode bits (NM and FM). The state of NM can be determined programatically with the TESTNM subroutine listed below. Upon return, accumulator A will contain the value of the NM bit. All other registers are preserved. When run on a 6809 processor it will always return with A = 0.

```
TSTNM   PSHS    U,Y,X,DP,CC      ; Preserve Registers
        ORCC    #$D0             ; Mask interrupts and set E flag
        TFR     W,Y              ; Y=W (6309), Y=$FFFF (6809)
        LDA     #1               ; Set result for NM=1
        BSR     L1               ; Set return point for RTI when NM=1
        BEQ     L0               ; Skip next instruction if NM=0
        TFR     X,W              ; Restore W
L0       PULS    CC,DP,X,Y,U      ; Restore other registers
        TSTA                    ; Setup CC.Z to reflect result
        RTS
L1       BSR     L2               ; Set return point for RTI when NM=0
        CLRA                    ; Set result for NM=0
        RTS
L2       PSHS    U,Y,X,DP,D,CC    ; Push emulation mode machine state
        RTI                    ; Return to one of the two BSR calls
```

The state of the FM bit can only be determined when an actual FIRQ interrupt occurs. Upon FIRQ, the 6309 copies the value of the FM bit into the Entire (E) bit of the CC register. An FIRQ service routine can check the state of E upon entry:

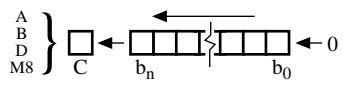
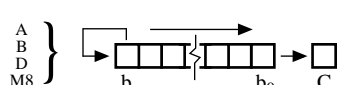
```
F_SRV   PSHS    A                ; Save A on the stack
        TFR     CC,A             ; Copy CC into A
        ANDA    #$80             ; Clear all flags except E
        STA     FMSTATE          ; Store for use by mainline code
        ...                     ; Clear interrupt source
        PULS    A                ; Restore A
        RTI                    ; Return
```



# **Part III**

## **Quick Reference**

# 6809 / 6309 Programming Aid

| Instr. | Forms   | Addressing Modes |     |   |        |     |   |                      |      |    |          |     |    | Description | 5 3 2 1 0 |                      |   |                   |   |   |   |   |
|--------|---------|------------------|-----|---|--------|-----|---|----------------------|------|----|----------|-----|----|-------------|-----------|----------------------|---|-------------------|---|---|---|---|
|        |         | Immediate        |     |   | Direct |     |   | Indexed <sup>1</sup> |      |    | Extended |     |    |             | Inherent  |                      |   | H                 | N | Z | V | C |
|        |         | Op               | ~   | # | Op     | ~   | # | Op                   | ~    | #  | Op       | ~   | #  |             | Op        | ~                    | #   |                   |   |   |   |   |
| ABX    | ABX     |                  |     |   |        |     |   |                      |      |    |          |     | 3A | 3/1         | 1         | X = X + B (unsigned) | -   | -                 | - | - | - |   |
| ADC    | ADCA    | 89               | 2   | 2 | 99     | 4/3 | 2 | A9                   | 4+   | 2+ | B9       | 5/4 | 3  |             |           |                      | A = A + M8 + C  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADCB    | C9               | 2   | 2 | D9     | 4/3 | 2 | E9                   | 4+   | 2+ | F9       | 5/4 | 3  |             |           |                      | B = B + M8 + C  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADCD    | 089              | 5/4 | 4 | 099    | 7/5 | 3 | 0A9                  | 7/6+ | 3+ | 0B9      | 8/6 | 4  |             |           |                      | D = D + M16 + C   | -                 | ↑ | ↑ | ↑ | ↑ |
|        | ADCR    | 031              | 4   | 3 |        |     |   |                      |      |    |          |     |    |             |           |                      | r1 = r1 + r0 + C  | -                 | ↑ | ↑ | ↑ | ↑ |
|        |         |                  |     |   |        |     |   |                      |      |    |          |     |    |             |           |                      | <i>See Note 2</i>   |                   |   |   |   |   |
| ADD    | ADDA    | 8B               | 2   | 2 | 9B     | 4/3 | 2 | AB                   | 4+   | 2+ | BB       | 5/4 | 3  |             |           |                      | A = A + M8  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDB    | CB               | 2   | 2 | DB     | 4/3 | 2 | EB                   | 4+   | 2+ | FB       | 5/4 | 3  |             |           |                      | B = B + M8  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDD    | C3               | 4/3 | 3 | D3     | 6/4 | 2 | E3                   | 6/5+ | 2+ | F3       | 7/5 | 3  |             |           |                      | D = D + M16   | -                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDE    | 18B              | 3   | 3 | 19B    | 5/4 | 3 | 1AB                  | 5+   | 3+ | 1BB      | 6/5 | 4  |             |           |                      | E = E + M8  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDF    | 1CB              | 3   | 3 | 1DB    | 5/4 | 3 | 1EB                  | 5+   | 3+ | 1FB      | 6/5 | 4  |             |           |                      | F = F + M8  | ↑                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDR    | 030              | 4   | 3 |        |     |   |                      |      |    |          |     |    |             |           |                      | r1 = r1 + r0  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | ADDW    | 08B              | 5/4 | 4 | 09B    | 7/5 | 3 | 0AB                  | 7/6+ | 3+ | 0BB      | 8/6 | 4  |             |           |                      | W = W + M16   | -                 | ↑ | ↑ | ↑ | ↑ |
|        |         |                  |     |   |        |     |   |                      |      |    |          |     |    |             |           |                      | <i>See Note 2</i>   |                   |   |   |   |   |
| AIM    | #I8, EA |                  |     |   | 02     | 6   | 3 | 62                   | 7+   | 3+ | 72       | 7   | 4  |             |           |                      | M8 = M8 & I8  | -                 | ↑ | ↑ | 0 | - |
| AND    | ANDA    | 84               | 2   | 2 | 94     | 4/3 | 2 | A4                   | 4+   | 2+ | B4       | 5/4 | 3  |             |           |                      | A = A & M8  | -                 | ↑ | ↑ | 0 | - |
|        | ANDB    | C4               | 2   | 2 | D4     | 4/3 | 2 | E4                   | 4+   | 2+ | F4       | 5/4 | 3  |             |           |                      | B = B & M8  | -                 | ↑ | ↑ | 0 | - |
|        | ANDCC   | 1C               | 3   | 2 |        |     |   |                      |      |    |          |     |    |             |           |                      | CC = CC & I8  | <i>See Note 7</i> |   |   |   |   |
|        | ANDD    | 084              | 5/4 | 4 | 094    | 7/5 | 3 | 0A4                  | 7/6+ | 3+ | 0B4      | 8/6 | 4  |             |           |                      | D = D & M16   | -                 | ↑ | ↑ | 0 | - |
|        | ANDR    | 034              | 4   | 3 |        |     |   |                      |      |    |          |     |    |             |           |                      | r1 = r1 & r0  | -                 | ↑ | ↑ | 0 | - |
|        |         |                  |     |   |        |     |   |                      |      |    |          |     |    |             |           |                      | <i>See Note 2</i>   |                   |   |   |   |   |
| ASL    | ASLA    |                  |     |   |        |     |   |                      |      |    |          |     |    | 48          | 2/1       | 1                    |  | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | ASLB    |                  |     |   |        |     |   |                      |      |    |          |     |    | 58          | 2/1       | 1                    |   | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | ASLD    |                  |     |   |        |     |   |                      |      |    |          |     |    | 048         | 3/2       | 2                    |   | -                 | ↑ | ↑ | ↑ | ↑ |
|        | ASL     |                  |     |   | 08     | 6/5 | 2 | 68                   | 6+   | 2+ | 78       | 7/6 | 3  |             |           |                      |   | 8                 | ↑ | ↑ | ↑ | ↑ |
| ASR    | ASRA    |                  |     |   |        |     |   |                      |      |    |          |     |    | 47          | 2/1       | 1                    |  | 8                 | ↑ | ↑ | - | ↑ |
|        | ASRB    |                  |     |   |        |     |   |                      |      |    |          |     |    | 57          | 2/1       | 1                    |   | 8                 | ↑ | ↑ | - | ↑ |
|        | ASRD    |                  |     |   |        |     |   |                      |      |    |          |     |    | 047         | 3/2       | 2                    |   | -                 | ↑ | ↑ | - | ↑ |
|        | ASR     |                  |     |   | 07     | 6/5 | 2 | 67                   | 6+   | 2+ | 77       | 7/6 | 3  |             |           |                      |   | 8                 | ↑ | ↑ | - | ↑ |
|        |         |                  |     |   |        |     |   |                      |      |    |          |     |    |             |           |                      |   |                   |   |   | - | ↑ |
| BAND   | BAND    |                  |     |   | 130    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = M8.srcBit & R.dstBit   | <i>See Note 3</i> |   |   |   |   |
|        | BIAND   |                  |     |   | 131    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = $\overline{\text{M8.srcBit}}$ & R.dstBit                                 | <i>See Note 3</i> |   |   |   |   |
| BEOR   | BEOR    |                  |     |   | 134    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = M8.srcBit XOR R.dstBit   | <i>See Note 3</i> |   |   |   |   |
|        | BIEOR   |                  |     |   | 135    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = $\overline{\text{M8.srcBit}}$ XOR R.dstBit                               | <i>See Note 3</i> |   |   |   |   |
| BIT    | BITA    | 85               | 2   | 2 | 95     | 4/3 | 2 | A5                   | 4+   | 2+ | B5       | 5/4 | 3  |             |           |                      | Bit Test A (A & M8)   | -                 | ↑ | ↑ | 0 | - |
|        | BITB    | C5               | 2   | 2 | D5     | 4/3 | 2 | E5                   | 4+   | 2+ | F5       | 5/4 | 3  |             |           |                      | Bit Test B (B & M8)   | -                 | ↑ | ↑ | 0 | - |
|        | BITD    | 085              | 5/4 | 4 | 095    | 7/5 | 3 | 0A5                  | 7/6+ | 3+ | 0B5      | 8/6 | 4  |             |           |                      | Bit Test D (D & M16)  | -                 | ↑ | ↑ | 0 | - |
|        | BITMD   | 13C              | 4   | 3 |        |     |   |                      |      |    |          |     |    |             |           |                      | Bit Test MD (MD & I8) bits 6 and 7 only   | -                 | - | ↑ | - | - |
| BOR    | BOR     |                  |     |   | 132    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = M8.srcBit   R.dstBit   | <i>See Note 3</i> |   |   |   |   |
|        | BIOR    |                  |     |   | 133    | 7/6 | 4 |                      |      |    |          |     |    |             |           |                      | R.dstBit = $\overline{\text{M8.srcBit}}$   R.dstBit                                 | <i>See Note 3</i> |   |   |   |   |
| CLR    | CLRA    |                  |     |   |        |     |   |                      |      |    |          |     |    | 4F          | 2/1       | 1                    | A = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLRB    |                  |     |   |        |     |   |                      |      |    |          |     |    | 5F          | 2/1       | 1                    | B = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLRD    |                  |     |   |        |     |   |                      |      |    |          |     |    | 04F         | 3/2       | 2                    | D = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLRE    |                  |     |   |        |     |   |                      |      |    |          |     |    | 14F         | 3/2       | 2                    | E = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLRF    |                  |     |   |        |     |   |                      |      |    |          |     |    | 15F         | 3/2       | 2                    | F = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLRW    |                  |     |   |        |     |   |                      |      |    |          |     |    | 05F         | 3/2       | 2                    | W = 0   | -                 | 0 | 1 | 0 | 0 |
|        | CLR     |                  |     |   | 0F     | 6/5 | 2 | 6F                   | 6+   | 2+ | 7F       | 7/6 | 3  |             |           |                      | M8 = 0  | -                 | 0 | 1 | 0 | 0 |
| CMP    | CMPA    | 81               | 2   | 2 | 91     | 4/3 | 2 | A1                   | 4+   | 2+ | B1       | 5/4 | 3  |             |           |                      | Compare M8 from A   | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPB    | C1               | 2   | 2 | D1     | 4/3 | 2 | E1                   | 4+   | 2+ | F1       | 5/4 | 3  |             |           |                      | Compare M8 from B   | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPD    | 083              | 5/4 | 4 | 093    | 7/5 | 3 | 0A3                  | 7/6+ | 3+ | 0B3      | 8/6 | 4  |             |           |                      | Compare M16 from D  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPE    | 181              | 3   | 3 | 191    | 5/4 | 3 | 1A1                  | 5+   | 3+ | 1B1      | 6/5 | 4  |             |           |                      | Compare M8 from E   | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPF    | 1C1              | 3   | 3 | 1D1    | 5/4 | 3 | 1E1                  | 5+   | 3+ | 1F1      | 6/5 | 4  |             |           |                      | Compare M8 from F   | 8                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPR    | 037              | 4   | 3 |        |     |   |                      |      |    |          |     |    |             |           |                      | Compare r0 from r1  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPS    | 18C              | 5/4 | 4 | 19C    | 7/5 | 3 | 1AC                  | 7/6+ | 3+ | 1BC      | 8/6 | 4  |             |           |                      | Compare M16 from S  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPU    | 183              | 5/4 | 4 | 193    | 7/5 | 3 | 1A3                  | 7/6+ | 3+ | 1B3      | 8/6 | 4  |             |           |                      | Compare M16 from U  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPW    | 081              | 5/4 | 4 | 091    | 7/5 | 3 | 0A1                  | 7/6+ | 3+ | 0B1      | 8/6 | 4  |             |           |                      | Compare M16 from W  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPX    | 8C               | 4/3 | 3 | 9C     | 6/4 | 2 | AC                   | 6/5+ | 2+ | BC       | 7/5 | 3  |             |           |                      | Compare M16 from X  | -                 | ↑ | ↑ | ↑ | ↑ |
|        | CMPY    | 08C              | 5/4 | 4 | 09C    | 7/5 | 3 | 0AC                  | 7/6+ | 3+ | 0BC      | 8/6 | 4  |             |           |                      | Compare M16 from Y  | -                 | ↑ | ↑ | ↑ | ↑ |

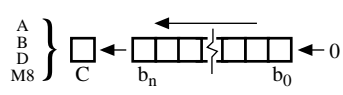
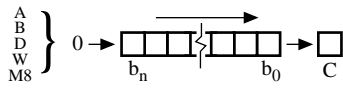
Legend:

- Op** Hex Operation Code (Leading '1' not shown for two-byte opcodes)
- ~** Number of MPU Cycles (6809 emulation / native)
- #** Number of Program Bytes
- I8** 8-bit Immediate value
- I16** 16-bit Immediate value
- M8** 8-bit value in Memory (may also include Immediate values)
- M16** 16-bit value in Memory (may also include Immediate values)

- EA** Effective Address
- C** Value of Carry flag in CC
- r0** First register (source) operand
- r1** Second register (destination) operand
- ↑** Status flag Set if TRUE, Cleared otherwise
- Status flag Not Affected by operation

Instructions in shaded rows are not available on 6809 microprocessors

# 6809 / 6309 Programming Aid continued

| Instr. | Forms   | Addressing Modes   |  |  |  |   |   |  |  |  |  |  |   | Description                          | 5 3 2 1 0                              |                            |  |  |  |  |   |  |
|--------|---|--|--|--|--|---|---|--|--|--|--|--|---|--------------------------------------|--|----------------------------|--|--|--|--|---|--|
|        |   | Immediate  |  |  | Direct   |   |   | Indexed <sup>1</sup>   |  |  | Extended   |  |   |                                      | Inherent                               |                            |  | H  | N  | Z  | V   | C  |
|        |   | Op   | ~  | #  | Op   | ~   | #   | Op   | ~  | #  | Op   | ~  | #   |                                      | Op                                     | ~                          | #  |  |  |  |   |  |
| COM    | COMA<br>COMB<br>COMD<br>COME<br>COMF<br>COMW<br>COM                               |  |  |  |  |   |   |  |  |  |  |  |   | 43<br>53<br>043<br>143<br>153<br>053 | 2/1<br>2/1<br>3/2<br>3/2<br>3/2<br>3/2 | 1<br>1<br>2<br>2<br>2<br>2 | A = $\overline{A}$<br>B = $\overline{B}$<br>D = $\overline{D}$<br>E = $\overline{E}$<br>F = $\overline{F}$<br>W = $\overline{W}$<br>M8 = $\overline{M8}$ | -<br>-<br>-<br>-<br>-<br>-                               | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | 0<br>0<br>0<br>0<br>0<br>0                          | 1<br>1<br>1<br>1<br>1<br>1                           |
| CWAI   |   | 3C   | 22/20  | 2  |  |   |   |  |  |  |  |  |   |                                      |  |                            | CC = CC & I8 ; Wait for interrupt  | See Note 7   |  |  |   |  |
| DAA    |   |  |  |  |  |   |   |  |  |  |  |  |   | 19                                   | 2/1                                    | 1                          | Decimal Adjust A   | -  | $\uparrow$   | $\uparrow$   | 8   | $\uparrow$   |
| DEC    | DECA<br>DECB<br>DECD<br>DECE<br>DECF<br>DECW<br>DEC                               |  |  |  |  |   |   |  |  |  |  |  |   | 4A<br>5A<br>04A<br>14A<br>15A<br>05A | 2/1<br>2/1<br>3/2<br>3/2<br>3/2<br>3/2 | 1<br>1<br>2<br>2<br>2<br>2 | A = A - 1<br>B = B - 1<br>D = D - 1<br>E = E - 1<br>F = F - 1<br>W = W - 1<br>M8 = M8 - 1  | -<br>-<br>-<br>-<br>-<br>-                               | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | -<br>-<br>-<br>-<br>-<br>-                          |  |
| DIV    | DIVD<br>DIVQ  | 18D<br>18E   | 25<br>34   | 3<br>4   | 19D<br>19E   | 27/26<br>36/35  | 3<br>3  | 1AD<br>1AE   | 27+<br>36+   | 3+<br>3+   | 1BD<br>1BE   | 28/27<br>37/36   | 4<br>4  |                                      |  |                            | B = D ÷ M8; A = modulo See Note 12<br>W = Q ÷ M16; D = modulo See Note 12  | -<br>-   | $\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$                            | 9<br>9   |
| EIM    | #I8, EA   |  |  |  | 05   | 6   | 3   | 65   | 7+   | 3+   | 75   | 7  | 4   |                                      |  |                            | M8 = M8 xor I8   | -  | $\uparrow$   | $\uparrow$   | 0   | -  |
| EOR    | EORA<br>EORB<br>EORD<br>EORR  | 88<br>C8<br>088<br>036   | 2<br>2<br>5/4<br>4                                       | 2<br>2<br>4<br>3   | 98<br>D8<br>098  | 4/3<br>4/3<br>7/5   | 2<br>2<br>3   | A8<br>E8<br>0A8  | 4+<br>4+<br>7/6+   | 2+<br>2+<br>3+   | B8<br>F8<br>0B8  | 5/4<br>5/4<br>8/6  | 3<br>3<br>4   |                                      |  |                            | A = A ⊕ M8<br>B = B ⊕ M8<br>D = D ⊕ M16<br>r1 = r0 ⊕ r1 See Note 2   | -<br>-<br>-<br>-   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | 0<br>0<br>0<br>0                                    | -<br>-<br>-<br>-                                     |
| EXG    | r0, r1  | 1E   | 8/5  | 2  |  |   |   |  |  |  |  |  |   |                                      |  |                            | r0 ↔ r1 See Note 2   | -  | -  | -  | -   | -  |
| INC    | INCA<br>INCB<br>INCD<br>INCE<br>INCF<br>INCW<br>INC                               |  |  |  |  |   |   |  |  |  |  |  |   | 4C<br>5C<br>04C<br>14C<br>15C<br>05C | 2/1<br>2/1<br>3/2<br>3/2<br>3/2<br>3/2 | 1<br>1<br>2<br>2<br>2<br>2 | A = A + 1<br>B = B + 1<br>D = D + 1<br>E = E + 1<br>F = F + 1<br>W = W + 1<br>M8 = M8 + 1  | -<br>-<br>-<br>-<br>-<br>-                               | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | -<br>-<br>-<br>-<br>-<br>-                          |  |
| JMP    |   |  |  |  | 0E   | 3/2   | 2   | 6E   | 3+   | 2+   | 7E   | 4/3  | 3   |                                      |  |                            | PC = Effective Address   | -  | -  | -  | -   | -  |
| JSR    |   |  |  |  | 9D   | 7/6   | 2   | AD   | 7/6+   | 2+   | BD   | 8/7  | 3   |                                      |  |                            | Jump to Subroutine   | -  | -  | -  | -   | -  |
| LD     | LDA<br>LDB<br>LDD<br>LDE<br>LDF<br>LDMD<br>LDQ<br>LDS<br>LDU<br>LDW<br>LDX<br>LDY | 86<br>C6<br>CC<br>186<br>1C6<br>13D<br>CD<br>0CE<br>CE<br>086<br>8E<br>08E | 2<br>2<br>3<br>3<br>3<br>5<br>5<br>4<br>3<br>4<br>3<br>4 | 2<br>2<br>3<br>3<br>3<br>3<br>5<br>4<br>3<br>4<br>3<br>4 | 96<br>D6<br>DC<br>196<br>1D6<br>0DC<br>0DE<br>DE<br>096<br>9E<br>09E | 4/3<br>4/3<br>5/4<br>5/4<br>5/4<br>8/7<br>6/5<br>5/4<br>6/5<br>5/4<br>6/5 | 2<br>2<br>2<br>3<br>3<br>3<br>3<br>2<br>2<br>3<br>3 | A6<br>E6<br>EC<br>1A6<br>1E6<br>0EC<br>0EE<br>EE<br>0A6<br>AE<br>0AE | 4+<br>4+<br>5+<br>5+<br>5+<br>8+<br>6+<br>5+<br>6+<br>5+<br>6+ | 2+<br>2+<br>2+<br>3+<br>3+<br>3+<br>3+<br>2+<br>2+<br>2+<br>3+ | B6<br>F6<br>FC<br>1B6<br>1F6<br>0FC<br>0FE<br>FE<br>0B6<br>BE<br>0BE | 5/4<br>5/4<br>6/5<br>6/5<br>6/5<br>9/8<br>7/6<br>6/5<br>7/6<br>6/5 | 3<br>3<br>3<br>4<br>4<br>4<br>4<br>3<br>3<br>3<br>4 |                                      |  |                            | A = M8<br>B = M8<br>D = M16<br>E = M8<br>F = M8<br>MD = I8<br>Q = M32<br>S = M16<br>U = M16<br>W = M16<br>X = M16<br>Y = M16                             | -<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>- | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$ | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$ | 0<br>0<br>0<br>0<br>0<br>0<br>0<br>0<br>0<br>0<br>0 | -<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-<br>-  |
| LDBT   |   |  |  |  | 136  | 7/6   | 4   |  |  |  |  |  |   |                                      |  |                            | R.dstBit = M8.srcBit   | See Note 3   |  |  |   |  |
| LEA    | LEAS<br>LEAU<br>LEAX<br>LEAY  |  |  |  |  |   |   | 32<br>33<br>30<br>31   | 4+<br>4+<br>4+<br>4+   | 2+<br>2+<br>2+<br>2+   |  |  |   |                                      |  |                            | S = Effective Address<br>U = Effective Address<br>X = Effective Address<br>Y = Effective Address   | -<br>-<br>-<br>-   | -<br>-<br>$\uparrow$<br>$\uparrow$   | -<br>-<br>$\uparrow$<br>$\uparrow$   | -<br>-<br>$\uparrow$<br>$\uparrow$                  | -<br>-<br>$\uparrow$<br>$\uparrow$                   |
| LSL    | LSLA<br>LSLB<br>LSLD<br>LSL   |  |  |  |  |   |   |  |  |  |  |  |   | 48<br>58<br>048                      | 2/1<br>2/1<br>3/2                      | 1<br>1<br>2                |   | 8<br>8<br>8  | $\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$   | $\uparrow$<br>$\uparrow$<br>$\uparrow$              | $\uparrow$<br>$\uparrow$<br>$\uparrow$               |
| LSR    | LSRA<br>LSRB<br>LSRD<br>LSRW<br>LSR   |  |  |  |  |   |   |  |  |  |  |  |   | 44<br>54<br>044<br>054               | 2/1<br>2/1<br>3/2<br>3/2               | 1<br>1<br>2<br>2           |   | -<br>-<br>-<br>-   | 0<br>0<br>0<br>0   | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$   | -<br>-<br>$\uparrow$<br>$\uparrow$                  | $\uparrow$<br>$\uparrow$<br>$\uparrow$<br>$\uparrow$ |

# 6809 / 6309 Programming Aid continued

| Instr. | Forms   | Addressing Modes |      |   |        |       |   |                      |      |    |          |       |   | Description | 5 3 2 1 0 |   |   |            |   |   |   |   |   |
|--------|---------|------------------|------|---|--------|-------|---|----------------------|------|----|----------|-------|---|-------------|-----------|---|---|------------|---|---|---|---|---|
|        |         | Immediate        |      |   | Direct |       |   | Indexed <sup>1</sup> |      |    | Extended |       |   |             | Inherent  |   |   | H          | N | Z | V | C |   |
|        |         | Op               | ~    | # | Op     | ~     | # | Op                   | ~    | #  | Op       | ~     | # |             | Op        | ~ | #   |            |   |   |   |   |   |
| MUL    | MUL     |                  |      |   |        |       |   |                      |      |    |          |       |   | 3D          | 11/10     | 1 | D = A * B (unsigned)  | -          | - | ↑ | - | 9 |   |
|        | MULD    | 18F              | 28   | 4 | 19F    | 30/29 | 3 | 1AF                  | 30+  | 3+ | 1BF      | 31/30 | 4 |             |           |   | Q = D * M16 (signed)  | -          | ↑ | 6 | - | 0 |   |
| NEG    | NEGA    |                  |      |   |        |       |   |                      |      |    |          |       |   | 40          | 2/1       | 1 | A = $\overline{A}$ + 1  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | NEGB    |                  |      |   |        |       |   |                      |      |    |          |       |   | 50          | 2/1       | 1 | B = $\overline{B}$ + 1  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | NEGD    |                  |      |   |        |       |   |                      |      |    |          |       |   | 040         | 3/2       | 2 | D = $\overline{D}$ + 1  | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | NEG     |                  |      |   | 00     | 6/5   | 2 | 60                   | 6+   | 2+ | 70       | 7/6   | 3 |             |           |   | M8 = $\overline{M8}$ + 1  | 8          | ↑ | ↑ | ↑ | ↑ |   |
| NOP    |         |                  |      |   |        |       |   |                      |      |    |          |       |   | 12          | 2/1       | 1 | No Operation  | -          | - | - | - | - |   |
| OIM    | #I8, EA |                  |      |   | 01     | 6     | 3 | 61                   | 7+   | 3+ | 71       | 7     | 4 |             |           |   | M8 = M8   I8  | -          | ↑ | ↑ | 0 | - |   |
| OR     | ORA     | 8A               | 2    | 2 | 9A     | 4/3   | 2 | AA                   | 4+   | 2+ | BA       | 5/4   | 3 |             |           |   | A = A   M8  | -          | ↑ | ↑ | 0 | - |   |
|        | ORB     | CA               | 2    | 2 | DA     | 4/3   | 2 | EA                   | 4+   | 2+ | FA       | 5/4   | 3 |             |           |   | B = B   M8  | -          | ↑ | ↑ | 0 | - |   |
|        | ORCC    | 1A               | 3    | 2 |        |       |   |                      |      |    |          |       |   |             |           |   | CC = CC   I8  | See Note 7 |   |   |   |   |   |
|        | ORD     | 08A              | 5/4  | 4 | 09A    | 7/5   | 3 | 0AA                  | 7/6+ | 3+ | 0BA      | 8/6   | 4 |             |           |   | D = D   M16   | -          | ↑ | ↑ | 0 | - |   |
|        | ORR     | 035              | 4    | 3 |        |       |   |                      |      |    |          |       |   |             |           |   | r1 = r1   r0  | -          | ↑ | ↑ | 0 | - |   |
|        |         |                  |      |   |        |       |   |                      |      |    |          |       |   |             |           |   | See Note 2  |            |   |   |   |   |   |
| PSH    | PSHS    | 34               | 5/4+ | 2 |        |       |   |                      |      |    |          |       |   |             |           |   | Push registers onto S stack   | See Note 4 | - | - | - | - | - |
|        | PSHU    | 36               | 5/4+ | 2 |        |       |   |                      |      |    |          |       |   |             |           |   | Push registers onto U stack   | See Note 4 | - | - | - | - | - |
|        | PSHSW   |                  |      |   |        |       |   |                      |      |    |          |       |   | 038         | 6         | 2 | Push W onto S stack   | -          | - | - | - | - |   |
|        | PSHUW   |                  |      |   |        |       |   |                      |      |    |          |       |   | 03A         | 6         | 2 | Push W onto U stack   | -          | - | - | - | - |   |
| PUL    | PULS    | 35               | 5/4+ | 2 |        |       |   |                      |      |    |          |       |   |             |           |   | Pull registers from S stack   | See Note 4 | - | - | - | - | - |
|        | PULU    | 37               | 5/4+ | 2 |        |       |   |                      |      |    |          |       |   |             |           |   | Pull registers from U stack   | See Note 4 | - | - | - | - | - |
|        | PULSW   |                  |      |   |        |       |   |                      |      |    |          |       |   | 039         | 6         | 2 | Pull W from S stack   | -          | - | - | - | - |   |
|        | PULUW   |                  |      |   |        |       |   |                      |      |    |          |       |   | 03B         | 6         | 2 | Pull W from U stack   | -          | - | - | - | - |   |
| ROL    | ROLA    |                  |      |   |        |       |   |                      |      |    |          |       |   | 49          | 2/1       | 1 | <div><div><div>A</div><div>B</div><div>D</div><div>W</div><div>M8</div></div><div><div><div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div></div><div>C</div><div><div>b<sub>n</sub></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div>b<sub>0</sub></div></div></div></div> | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | ROLB    |                  |      |   |        |       |   |                      |      |    |          |       |   | 59          | 2/1       | 1 |   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | ROLD    |                  |      |   |        |       |   |                      |      |    |          |       |   | 049         | 3/2       | 2 |   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | ROLW    |                  |      |   |        |       |   |                      |      |    |          |       |   | 059         | 3/2       | 2 |   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | ROL     |                  |      |   | 09     | 6/5   | 2 | 69                   | 6+   | 2+ | 79       | 7/6   | 3 |             |           |   |   | -          | ↑ | ↑ | ↑ | ↑ |   |
| ROR    | RORA    |                  |      |   |        |       |   |                      |      |    |          |       |   | 46          | 2/1       | 1 | <div><div><div>A</div><div>B</div><div>D</div><div>W</div><div>M8</div></div><div><div><div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div></div><div>C</div><div><div>b<sub>n</sub></div><div></div><div></div><div></div><div></div></div><div><div></div><div></div><div></div><div></div><div></div></div><div>b<sub>0</sub></div></div></div></div> | -          | ↑ | ↑ | - | ↑ |   |
|        | RORB    |                  |      |   |        |       |   |                      |      |    |          |       |   | 56          | 2/1       | 1 |   | -          | ↑ | ↑ | - | ↑ |   |
|        | RORD    |                  |      |   |        |       |   |                      |      |    |          |       |   | 046         | 3/2       | 2 |   | -          | ↑ | ↑ | - | ↑ |   |
|        | RORW    |                  |      |   |        |       |   |                      |      |    |          |       |   | 056         | 3/2       | 2 |   | -          | ↑ | ↑ | - | ↑ |   |
|        | ROR     |                  |      |   | 06     | 6/5   | 2 | 66                   | 6+   | 2+ | 76       | 7/6   | 3 |             |           |   |   | -          | ↑ | ↑ | - | ↑ |   |
| RTI    |         |                  |      |   |        |       |   |                      |      |    |          |       |   | 3B          | 15/17     | 1 | Return from Interrupt (when CC.E = 1)   | See Note 7 |   |   |   |   |   |
|        |         |                  |      |   |        |       |   |                      |      |    |          |       |   | 3B          | 6         | 1 | Return from Interrupt (when CC.E = 0)   | See Note 7 |   |   |   |   |   |
| RTS    |         |                  |      |   |        |       |   |                      |      |    |          |       |   | 39          | 5/4       | 1 | Return from Subroutine  | -          | - | - | - | - |   |
| SBC    | SBCA    | 82               | 2    | 2 | 92     | 4/3   | 2 | A2                   | 4+   | 2+ | B2       | 5/4   | 3 |             |           |   | A = A - M8 - C  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SBCB    | C2               | 2    | 2 | D2     | 4/3   | 2 | E2                   | 4+   | 2+ | F2       | 5/4   | 3 |             |           |   | B = B - M8 - C  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SBCD    | 082              | 5/4  | 4 | 092    | 7/5   | 3 | 0A2                  | 7/6+ | 3+ | 0B2      | 8/6   | 4 |             |           |   | D = D - M16 - C   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | SBCR    | 033              | 4    | 3 |        |       |   |                      |      |    |          |       |   |             |           |   | r1 = r1 - r0 - C  | -          | ↑ | ↑ | ↑ | ↑ |   |
|        |         |                  |      |   |        |       |   |                      |      |    |          |       |   |             |           |   | See Note 2  |            |   |   |   |   |   |
| SEX    | SEX     |                  |      |   |        |       |   |                      |      |    |          |       |   | 1D          | 2/1       | 1 | Sign Extend B into A  | -          | ↑ | ↑ | - | - |   |
|        | SEXW    |                  |      |   |        |       |   |                      |      |    |          |       |   | 14          | 4         | 1 | Sign Extend W into D  | -          | ↑ | ↑ | - | - |   |
| ST     | STA     |                  |      |   | 97     | 4/3   | 2 | A7                   | 4+   | 2+ | B7       | 5/4   | 3 |             |           |   | M8 = A  | -          | ↑ | ↑ | 0 | - |   |
|        | STB     |                  |      |   | D7     | 4/3   | 2 | E7                   | 4+   | 2+ | F7       | 5/4   | 3 |             |           |   | M8 = B  | -          | ↑ | ↑ | 0 | - |   |
|        | STD     |                  |      |   | DD     | 5/4   | 2 | ED                   | 5+   | 2+ | FD       | 6/5   | 3 |             |           |   | M16 = D   | -          | ↑ | ↑ | 0 | - |   |
|        | STE     |                  |      |   | 197    | 5/4   | 3 | 1A7                  | 5+   | 3+ | 1B7      | 6/5   | 4 |             |           |   | M8 = E  | -          | ↑ | ↑ | 0 | - |   |
|        | STF     |                  |      |   | 1D7    | 5/4   | 3 | 1E7                  | 5+   | 3+ | 1F7      | 6/5   | 4 |             |           |   | M8 = F  | -          | ↑ | ↑ | 0 | - |   |
|        | STQ     |                  |      |   | 0DD    | 8/7   | 3 | 0ED                  | 8+   | 3+ | 0FD      | 9/8   | 4 |             |           |   | M32 = Q   | -          | ↑ | ↑ | 0 | - |   |
|        | STS     |                  |      |   | 0DF    | 6/5   | 3 | 0EF                  | 6+   | 3+ | 0FF      | 7/6   | 4 |             |           |   | M16 = S   | -          | ↑ | ↑ | 0 | - |   |
|        | STU     |                  |      |   | DF     | 5/4   | 2 | EF                   | 5+   | 2+ | FF       | 6/5   | 3 |             |           |   | M16 = U   | -          | ↑ | ↑ | 0 | - |   |
|        | STW     |                  |      |   | 097    | 6/5   | 3 | 0A7                  | 6+   | 3+ | 0B7      | 7/6   | 4 |             |           |   | M16 = W   | -          | ↑ | ↑ | 0 | - |   |
|        | STX     |                  |      |   | 9F     | 5/4   | 2 | AF                   | 5+   | 2+ | BF       | 6/5   | 3 |             |           |   | M16 = X   | -          | ↑ | ↑ | 0 | - |   |
|        | STY     |                  |      |   | 09F    | 6/5   | 3 | 0AF                  | 6+   | 3+ | 0BF      | 7/6   | 4 |             |           |   | M16 = Y   | -          | ↑ | ↑ | 0 | - |   |
| STBT   |         |                  |      |   | 137    | 8/7   | 4 |                      |      |    |          |       |   |             |           |   | M8.dstBit = R.srcBit  | -          | ↑ | ↑ | - | - |   |
| SUB    | SUBA    | 80               | 2    | 2 | 90     | 4/3   | 2 | A0                   | 4+   | 2+ | B0       | 5/4   | 3 |             |           |   | A = A - M8  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBB    | C0               | 2    | 2 | D0     | 4/3   | 2 | E0                   | 4+   | 2+ | F0       | 5/4   | 3 |             |           |   | B = B - M8  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBD    | 83               | 4/3  | 3 | 93     | 6/4   | 2 | A3                   | 6/5+ | 2+ | B3       | 7/5   | 3 |             |           |   | D = D - M16   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBE    | 180              | 3    | 3 | 190    | 5/4   | 3 | 1A0                  | 5+   | 3+ | 1B0      | 6/5   | 4 |             |           |   | E = E - M8  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBF    | 1C0              | 3    | 3 | 1D0    | 5/4   | 3 | 1E0                  | 5+   | 3+ | 1F0      | 6/5   | 4 |             |           |   | F = F - M8  | 8          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBR    | 032              | 4    | 3 |        |       |   |                      |      |    |          |       |   |             |           |   | r1 = r1 - r0  | -          | ↑ | ↑ | ↑ | ↑ |   |
|        | SUBW    | 080              | 5/4  | 4 | 090    | 7/5   | 3 | 0A0                  | 7/6+ | 3+ | 0B0      | 8/6   | 4 |             |           |   | W = W - M16   | -          | ↑ | ↑ | ↑ | ↑ |   |
|        |         |                  |      |   |        |       |   |                      |      |    |          |       |   |             |           |   | See Note 2  |            |   |   |   |   |   |

# 6809 / 6309 Programming Aid continued

| Instr. | Forms               | Addressing Modes |      |   |        |     |   |                      |      |    |          |     |                  | Description                   | 5 3 2 1 0   |  |   |                    |   |   |   |   |   |
|--------|---------------------|------------------|------|---|--------|-----|---|----------------------|------|----|----------|-----|------------------|-------------------------------|-------------|--|---|--------------------|---|---|---|---|---|
|        |                     | Immediate        |      |   | Direct |     |   | Indexed <sup>1</sup> |      |    | Extended |     |                  |                               | Inherent    |  |   | H                  | N | Z | V | C |   |
|        |                     | Op               | ~    | # | Op     | ~   | # | Op                   | ~    | #  | Op       | ~   | #                |                               | Op          | ~  | #   |                    |   |   |   |   |   |
| SWI    | SWI<br>SWI2<br>SWI3 |                  |      |   |        |     |   |                      |      |    |          |     | 3F<br>03F<br>13F | 19 / 21<br>20 / 22<br>20 / 22 | 1<br>2<br>2 | Software Interrupt 1<br>Software Interrupt 2<br>Software Interrupt 3 | <i>See Note 5</i><br><i>See Note 5</i><br><i>See Note 5</i> | -                  | - | - | - | - |   |
| SYNC   |                     |                  |      |   |        |     |   |                      |      |    |          |     | 13               | ≥4/≥3                         | 1           | Synchronize to Interrupt   |   | -                  | - | - | - | - |   |
| TFM    | r0+, r1+            | 138              | 6+3n | 3 |        |     |   |                      |      |    |          |     |                  |                               |             |  | Block Move Incrementing                                     | <i>See Note 10</i> | - | - | 1 | - | - |
|        | r0-, r1-            | 139              | 6+3n | 3 |        |     |   |                      |      |    |          |     |                  |                               |             |  | Block Move Decrementing                                     | <i>See Note 10</i> | - | - | 1 | - | - |
|        | r0+, r1             | 13A              | 6+3n | 3 |        |     |   |                      |      |    |          |     |                  |                               |             |  | Block Write to address                                      | <i>See Note 10</i> | - | - | 1 | - | - |
|        | r0, r1+             | 13B              | 6+3n | 3 |        |     |   |                      |      |    |          |     |                  |                               |             |  | Block Read from address                                     | <i>See Note 10</i> | - | - | 1 | - | - |
| TFR    | r0, r1              | 1F               | 6/4  | 2 |        |     |   |                      |      |    |          |     |                  |                               |             |  | r1 = r0   | <i>See Note 2</i>  | - | - | - | - | - |
| TIM    | #I8, EA             |                  |      |   | 0B     | 6   | 3 | 6B                   | 7+   | 3+ | 7B       | 7   | 4                |                               |             |  | Bit Test Memory (I8 & M8)                                   |                    | - | ↑ | ↓ | 0 | - |
| TST    | TSTA                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 4D                            | 2/1         | 1  | Test A  |                    | - | ↑ | ↑ | 0 | - |
|        | TSTB                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 5D                            | 2/1         | 1  | Test B  |                    | - | ↑ | ↑ | 0 | - |
|        | TSTD                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 04D                           | 3/2         | 2  | Test D  |                    | - | ↑ | ↑ | 0 | - |
|        | TSTE                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 14D                           | 3/2         | 2  | Test E  |                    | - | ↑ | ↑ | 0 | - |
|        | TSTF                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 15D                           | 3/2         | 2  | Test F  |                    | - | ↑ | ↑ | 0 | - |
|        | TSTW                |                  |      |   |        |     |   |                      |      |    |          |     |                  | 05D                           | 3/2         | 2  | Test W  |                    | - | ↑ | ↑ | 0 | - |
|        | TST                 |                  |      |   | 0D     | 6/4 | 2 | 6D                   | 6/5+ | 2+ | 7D       | 7/5 | 3                |                               |             |  | Test M8   |                    | - | ↑ | ↑ | 0 | - |

| Instr. | Forms | Relative Addressing |       |   | Description                        |
|--------|-------|---------------------|-------|---|------------------------------------|
|        |       | Op                  | ~     | # |                                    |
| BCC    | BCC   | 24                  | 3     | 2 | Branch If C = 0                    |
|        | LBCC  | 024                 | 5 (6) | 4 | Long Branch If C = 0 (11)          |
| BCS    | BCS   | 25                  | 3     | 2 | Branch If C = 1                    |
|        | LBCS  | 025                 | 5 (6) | 4 | Long Branch If C = 1 (11)          |
| BEQ    | BEQ   | 27                  | 3     | 2 | Branch If Z = 1                    |
|        | LBEQ  | 027                 | 5 (6) | 4 | Long Branch If Z = 1 (11)          |
| BGE    | BGE   | 2C                  | 3     | 2 | Branch If ≥ 0                      |
|        | LBGE  | 02C                 | 5 (6) | 4 | Long Branch If ≥ 0 (11)            |
| BGT    | BGT   | 2E                  | 3     | 2 | Branch If > 0                      |
|        | LBGT  | 02E                 | 5 (6) | 4 | Long Branch If > 0 (11)            |
| BHI    | BHI   | 22                  | 3     | 2 | Branch If higher                   |
|        | LBHI  | 022                 | 5 (6) | 4 | Long Branch If higher (11)         |
| BHS    | BHS   | 24                  | 3     | 2 | Branch If higher or same           |
|        | LBHS  | 024                 | 5 (6) | 4 | Long Branch If higher or same (11) |
| BLE    | BLE   | 2F                  | 3     | 2 | Branch If ≤ 0                      |
|        | LBLE  | 02F                 | 5 (6) | 4 | Long Branch If ≤ 0 (11)            |
| BLO    | BLO   | 25                  | 3     | 2 | Branch If lower                    |
|        | LBLO  | 025                 | 5 (6) | 4 | Long Branch If lower (11)          |
| BLS    | BLS   | 23                  | 3     | 2 | Branch If lower or same            |
|        | LBLS  | 023                 | 5 (6) | 4 | Long Branch If lower or same (11)  |

| Instr. | Forms | Relative Addressing |       |   | Description                 |
|--------|-------|---------------------|-------|---|-----------------------------|
|        |       | Op                  | ~     | # |                             |
| BLT    | BLT   | 2D                  | 3     | 2 | Branch If < 0               |
|        | LBLT  | 02D                 | 5 (6) | 4 | Long Branch If < 0 (11)     |
| BMI    | BMI   | 2B                  | 3     | 2 | Branch If N = 1             |
|        | LBMI  | 02B                 | 5 (6) | 4 | Long Branch If N = 1 (11)   |
| BNE    | BNE   | 26                  | 3     | 2 | Branch If Z = 0             |
|        | LBNE  | 026                 | 5 (6) | 4 | Long Branch If Z = 0 (11)   |
| BPL    | BPL   | 2A                  | 3     | 2 | Branch If N = 0             |
|        | LBPL  | 02A                 | 5 (6) | 4 | Long Branch If N = 0 (11)   |
| BRA    | BRA   | 20                  | 3     | 2 | Branch unconditionally      |
|        | LBRA  | 16                  | 5/4   | 3 | Long Branch unconditionally |
| BRN    | BRN   | 21                  | 3     | 2 | Branch never (no-op)        |
|        | LBRN  | 021                 | 5     | 4 | Long Branch never (no-op)   |
| BSR    | BSR   | 8D                  | 7/6   | 2 | Branch to subroutine        |
|        | LBSR  | 17                  | 9/7   | 3 | Long Branch to subroutine   |
| BVC    | BVC   | 28                  | 3     | 2 | Branch If V = 0             |
|        | LBVC  | 028                 | 5 (6) | 4 | Long Branch If V = 0 (11)   |
| BVS    | BVS   | 29                  | 3     | 2 | Branch If V = 1             |
|        | LBVS  | 029                 | 5 (6) | 4 | Long Branch If V = 1 (11)   |

## Notes:

- The **Indexed** column provides base values for the MPU cycles and byte counts. To obtain totals, add the values from the **Indexed Addressing Mode Table** on page 150.
- r0 and r1 may be any pair of 8-bit registers, or any pair of 16-bit registers. Mixing registers of different sizes (TFR, EXG) behaves differently on the 6309 than on the 6809. The ZERO register (6309 only) may be used in combination with any other register. Undefined register codes produce a value of FF or FFFF on the 6809.
- The bit manipulation instructions (other than STBT) do not affect the CC register unless it is specified as the target register, in which case only the destination bit may be affected. Target registers for the bit manipulation instructions are limited to A, B and CC.
- The PSH and PUL instructions require one additional cycle for each **byte** pushed or pulled.
- SWI sets the I and F flags in CC. SWI2 and SWI3 do not affect I and F.
- The MULD instruction sets the Z flag in CC when the high-order word (D) is zero, even if the low-order word (W) is non-zero.
- The CC register is set as a direct result of the instruction.
- Value of the Condition Code bit is undefined.
- Special cases: For MUL, Carry set only if bit 7 is 1. For DIVD and DIVQ, Carry is set only if bit 0 is 1
- Source and destination registers for the TFM instruction are limited to X, Y, U, S and D. The W register always specifies the byte count. TFM is the only instruction that can be interrupted before it completes.
- Conditional long branches require a 6th cycle if the branch is taken (6809 only).
- The DIV instructions perform signed division. DIVD executes in 1 fewer cycle if a two's-complement overflow occurs. If a Range error occurs then the destination registers are not modified, the instruction executes in fewer cycles (13 fewer for DIVD, 21 fewer for DIVQ), the V flag is set and the N, Z and C flags are cleared.

## Indexed Addressing Mode Table

| Type  | Forms                            | Non Indirect   |                 |       |     | Indirect                         |                 |       |     |
|---|----------------------------------|----------------|-----------------|-------|-----|----------------------------------|-----------------|-------|-----|
|   |                                  | Assembler Form | Postbyte Opcode | + ~   | + # | Assembler Form                   | Postbyte Opcode | + ~   | + # |
| Constant Offset From R<br>(twos complement offset)    | No offset                        | ,R             | 1RR00100        | 0     | 0   | [ ,R]                            | 1RR10100        | 3     | 0   |
|   | 5 bit offset (-16 to +15)        | n,R            | 0RRnnnnn        | 1     | 0   | <i>not available - use 8-bit</i> |                 |       |     |
|   | 8 bit offset (-128 to +127)      | n,R            | 1RR01000        | 1     | 1   | [n,R]                            | 1RR11000        | 4     | 1   |
|   | 16 bit offset (-32768 to +32767) | n,R            | 1RR01001        | 4 / 3 | 2   | [n,R]                            | 1RR11001        | 7 / 6 | 2   |
| Constant Offset From W<br>(twos complement offset)    | No offset                        | ,W             | 10001111        | 0     | 0   | [ ,W]                            | 10010000        | 3     | 0   |
|   | 16 bit offset                    | n,W            | 10101111        | 2     | 2   | [n,W]                            | 10110000        | 5     | 2   |
| Accumulator Offset From R<br>(twos complement offset) | A - Accumulator offset           | A,R            | 1RR00110        | 1     | 0   | [A,R]                            | 1RR10110        | 4     | 0   |
|   | B - Accumulator offset           | B,R            | 1RR00101        | 1     | 0   | [B,R]                            | 1RR10101        | 4     | 0   |
|   | D - Accumulator offset           | D,R            | 1RR01011        | 4 / 2 | 0   | [D,R]                            | 1RR11011        | 7 / 5 | 0   |
|   | E - Accumulator offset           | E,R            | 1RR00111        | 1     | 0   | [E,R]                            | 1RR10111        | 4     | 0   |
|   | F - Accumulator offset           | F,R            | 1RR01010        | 1     | 0   | [F,R]                            | 1RR11010        | 4     | 0   |
|   | W - Accumulator offset           | W,R            | 1RR01110        | 1     | 0   | [W,R]                            | 1RR11110        | 4     | 0   |
| Auto Increment/Decrement of R                         | Post-Increment by 1              | ,R+            | 1RR00000        | 2 / 1 | 0   | <i>not allowed</i>               |                 |       |     |
|   | Post-Increment by 2              | ,R++           | 1RR00001        | 3 / 2 | 0   | [ ,R++]                          | 1RR10001        | 6 / 5 | 0   |
|   | Pre-Decrement by 1               | , -R           | 1RR00010        | 2 / 1 | 0   | <i>not allowed</i>               |                 |       |     |
|   | Pre-Decrement by 2               | , --R          | 1RR00011        | 3 / 2 | 0   | [ , --R]                         | 1RR10011        | 6 / 5 | 0   |
| Auto Increment/Decrement of W                         | Post-Increment by 2              | ,W++           | 11001111        | 1     | 0   | [ ,W++]                          | 11010000        | 4     | 0   |
|   | Pre-Decrement by 2               | , --W          | 11101111        | 1     | 0   | [ , --W]                         | 11110000        | 4     | 0   |
| Constant Offset From PC<br>(twos complement offset)   | 8 bit offset (-128 to +127)      | n,PCR          | 1XX01100        | 1     | 1   | [n,PCR]                          | 1XX11100        | 4     | 1   |
|   | 16 bit offset (-32768 to +32767) | n,PCR          | 1XX01101        | 5 / 3 | 2   | [n,PCR]                          | 1XX11101        | 8 / 6 | 2   |
| Extended Indirect                                     | 16 bit address                   |                |                 |       |     | [n]                              | 10011111        | 5 / 4 | 2   |

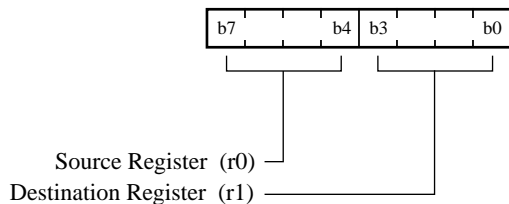
| RR | Register |
|----|----------|
| 00 | X        |
| 01 | Y        |
| 10 | U        |
| 11 | S        |

XX = Don't Care

+ and + these columns indicate the additional  
~ # number of MPU cycles and program bytes  
for the particular variation.

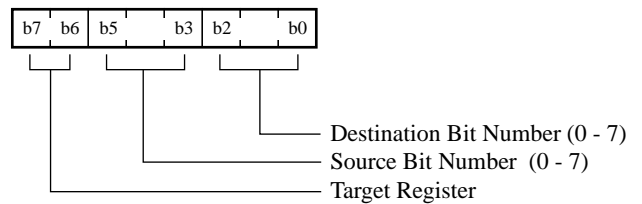
Indexing modes in shaded rows are not available on 6809 microprocessors.

### Inter-Register Postbyte



| Code | Register | Code | Register |
|------|----------|------|----------|
| 0000 | D        | 1000 | A        |
| 0001 | X        | 1001 | B        |
| 0010 | Y        | 1010 | CC       |
| 0011 | U        | 1011 | DP       |
| 0100 | S        | 1100 | 0        |
| 0101 | PC       | 1101 | 0        |
| 0110 | W        | 1110 | E        |
| 0111 | V        | 1111 | F        |

### Bit-Manipulation Postbyte (6309 only)



| Code | Register |
|------|----------|
| 00   | CC       |
| 01   | A        |
| 10   | B        |
| 11   | Invalid  |

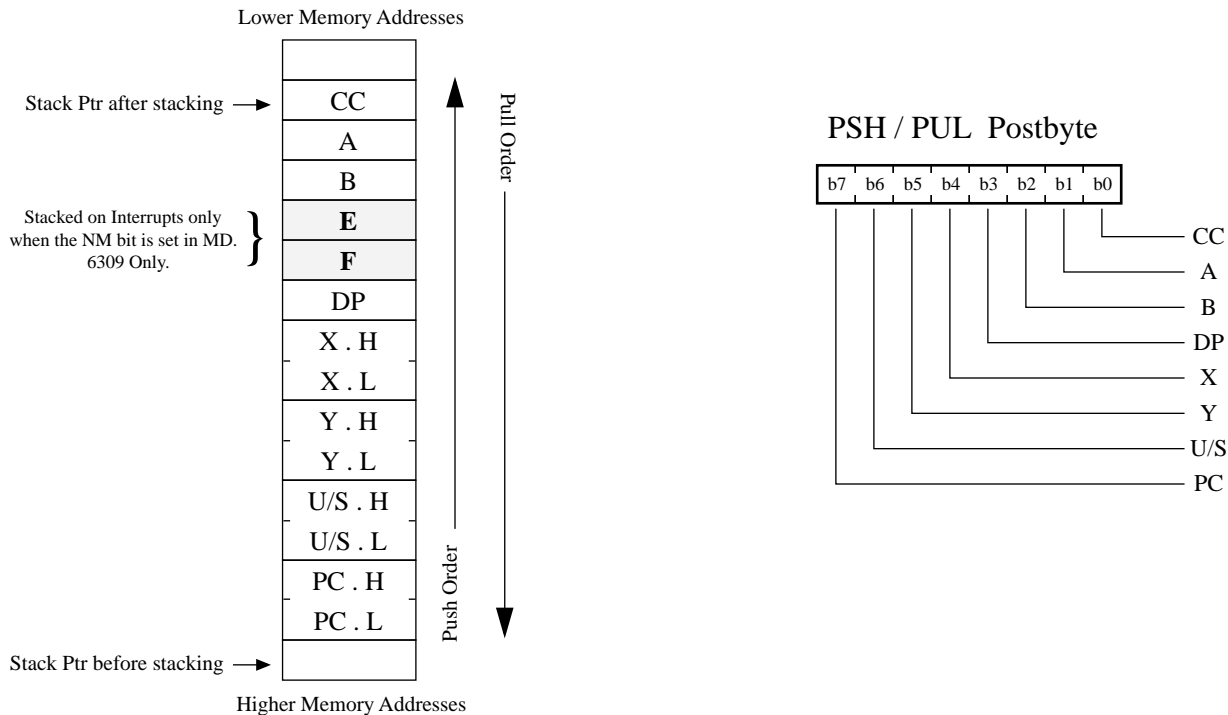
On 6809 microprocessors, the shaded Register Codes produce a value of FF or FFFF (all bits set).

## Programming Model

|  |  |               |  |   |    |               |   |                             |   |    |    |
|--|--|---------------|--|---|----|---------------|---|-----------------------------|---|----|----|
| Accumulator A  |  | Accumulator B |  | Accumulator E   |    | Accumulator F |   |                             |   |    |    |
| Accumulator D  |  |               |  | Accumulator W   |    |               |   |                             |   |    |    |
| Accumulator Q  |  |               |  |   |    |               |   |                             |   |    |    |
| <b>CC Register Bits</b><br>E Entire register state stacked<br>F FIRQ interrupt masked<br>H Half-Carry<br>I IRQ interrupt masked<br>N Negative result (twos complement)<br>Z Zero result<br>V Overflow<br>C Carry (or borrow) |  |               |  | Index Register X  |    |               |   |                             |   |    |    |
|  |  |               |  | Index Register Y  |    |               |   |                             |   |    |    |
|  |  |               |  | User Stack Pointer U  |    |               |   |                             |   |    |    |
|  |  |               |  | System Stack Pointer S  |    |               |   |                             |   |    |    |
|  |  |               |  | Program Counter PC  |    |               |   |                             |   |    |    |
|  |  |               |  | Transfer Value Register V   |    |               |   |                             |   |    |    |
|  |  |               |  | Zero Register 0   |    |               |   |                             |   |    |    |
|  |  |               |  | <b>MD Register Bits</b><br>/0 Divide-by-zero Exception<br>IL Illegal Instruction Exception<br>FM FIRQ uses IRQ stacking method (Entire state)<br>NM Native Mode (reduced cycles, W stacked on interrupts) |    |               |   | Condition Codes Register CC |   |    |    |
| Mode Register MD   |  |               |  |   |    |               |   |                             |   |    |    |
| Direct Page Register DP  |  |               |  |   |    |               |   |                             |   |    |    |
|  |  |               |  | E   | F  | H             | I | N                           | Z | V  | C  |
|  |  |               |  | /0  | IL |               |   |                             |   | FM | NM |
|  |  |               |  | b7  |    |               |   | b0                          |   |    |    |

The /0 and IL bits of the MD register can only be read once after an error exception occurs. They are reset to 0 after executing a BITMD instruction. The FM and NM bits of the MD register are write-only. Using the BITMD instruction to test these bits always produces zero.

## Register Stacking Order



When the FM bit in the MD register is set, the Entire register set is stacked upon an FIRQ interrupt, otherwise only CC and PC are stacked.

The Transfer Value register V is never stacked upon interrupts. No instructions are provided to directly push or pull the V register.

The E and F accumulators are stacked upon interrupts only if the NM bit is set in the MD register.

The PSHS, PULS, PSHU and PULU instructions do not permit the E and F accumulators (W) to be specified. These registers can be pushed and pulled together using the PSHSW, PSHUW, PULSW and PULUW instructions, or individually using the Auto Increment/Decrement Indexing modes with STE, STF, LDE, LDF (although these will have an effect on the Condition Codes).

# 6809 / 6309 Opcode Map

|         | DIRECT<br>\$0_ | \$1_   | REL<br>\$2_ | \$3_      | A / D / E<br>\$4_ | B / W / F<br>\$5_ | INDEX<br>\$6_ | EXTND<br>\$7_ | IMMED<br>\$8_ | DIRECT<br>\$9_ | INDEX<br>\$A_ | EXTND<br>\$B_ | IMMED<br>\$C_ | DIRECT<br>\$D_ | INDEX<br>\$E_ | EXTND<br>\$F_ |
|---------|----------------|--------|-------------|-----------|-------------------|-------------------|---------------|---------------|---------------|----------------|---------------|---------------|---------------|----------------|---------------|---------------|
| _0      | NEG            | PAGE 2 | BRA         | LEAX      | NEGA              | NEGB              | NEG           | NEG           | SUBA          | SUBA           | SUBA          | SUBA          | SUBB          | SUBB           | SUBB          | SUBB          |
| \$10 _0 |                |        |             | ADDR      | NEGD              |                   |               |               | SUBW          | SUBW           | SUBW          | SUBW          |               |                |               |               |
| \$11 _0 |                |        |             | BAND      |                   |                   |               |               | SUBE          | SUBE           | SUBE          | SUBE          | SUBF          | SUBF           | SUBF          | SUBF          |
| _1      | OIM            | PAGE 3 | BRN         | LEAY      |                   |                   | OIM           | OIM           | CMPA          | CMPA           | CMPA          | CMPA          | CMPB          | CMPB           | CMPB          | CMPB          |
| \$10 _1 |                |        | LB RN       | ADCR      |                   |                   |               |               | CMPW          | CMPW           | CMPW          | CMPW          |               |                |               |               |
| \$11 _1 |                |        |             | BIAND     |                   |                   |               |               | CMPE          | CMPE           | CMPE          | CMPE          | CMPF          | CMPF           | CMPF          | CMPF          |
| _2      | AIM            | NOP    | BHI         | LEAS      |                   |                   | AIM           | AIM           | SBCA          | SBCA           | SBCA          | SBCA          | SBCB          | SBCB           | SBCB          | SBCB          |
| \$10 _2 |                |        | LBHI        | SUBR      |                   |                   |               |               | SBCD          | SBCD           | SBCD          | SBCD          |               |                |               |               |
| \$11 _2 |                |        |             | BOR       |                   |                   |               |               |               |                |               |               |               |                |               |               |
| _3      | COM            | SYNC   | BLS         | LEAU      | COMA              | COMB              | COM           | COM           | SUBD          | SUBD           | SUBD          | SUBD          | ADDD          | ADDD           | ADDD          | ADDD          |
| \$10 _3 |                |        | LBLS        | SBCR      | COMD              | COMW              |               |               | CMPD          | CMPD           | CMPD          | CMPD          |               |                |               |               |
| \$11 _3 |                |        |             | BIOR      | COME              | COMF              |               |               | CMPU          | CMPU           | CMPU          | CMPU          |               |                |               |               |
| _4      | LSR            | SEXW   | BHS/CC      | PSHS      | LSRA              | LSRB              | LSR           | LSR           | ANDA          | ANDA           | ANDA          | ANDA          | ANDB          | ANDB           | ANDB          | ANDB          |
| \$10 _4 |                |        | LBHS/CC     | ANDR      | LSRD              | LSRW              |               |               | ANDD          | ANDD           | ANDD          | ANDD          |               |                |               |               |
| \$11 _4 |                |        |             | BEOR      |                   |                   |               |               |               |                |               |               |               |                |               |               |
| _5      | EIM            |        | BLO/CS      | PULS      |                   |                   | EIM           | EIM           | BITA          | BITA           | BITA          | BITA          | BITB          | BITB           | BITB          | BITB          |
| \$10 _5 |                |        | LBLO/CS     | ORR       |                   |                   |               |               | BITD          | BITD           | BITD          | BITD          |               |                |               |               |
| \$11 _5 |                |        |             | BIEOR     |                   |                   |               |               |               |                |               |               |               |                |               |               |
| _6      | ROR            | LBRA   | BNE         | PSHU      | RORA              | RORB              | ROR           | ROR           | LDA           | LDA            | LDA           | LDA           | LDB           | LDB            | LDB           | LDB           |
| \$10 _6 |                |        | LBNE        | EORR      | RORD              | RORW              |               |               | LDW           | LDW            | LDW           | LDW           |               |                |               |               |
| \$11 _6 |                |        |             | LDBT      |                   |                   |               |               | LDE           | LDE            | LDE           | LDE           | LDF           | LDF            | LDF           | LDF           |
| _7      | ASR            | LBSR   | BEQ         | PULU      | ASRA              | ASRB              | ASR           | ASR           |               | STA            | STA           | STA           |               | STB            | STB           | STB           |
| \$10 _7 |                |        | LBEQ        | CMPR      | ASRD              |                   |               |               |               | STW            | STW           | STW           |               |                |               |               |
| \$11 _7 |                |        |             | STBT      |                   |                   |               |               |               | STE            | STE           | STE           |               | STF            | STF           | STF           |
| _8      | LSL            |        | BVC         |           | LSLA              | LSLB              | LSL           | LSL           | EORA          | EORA           | EORA          | EORA          | EORB          | EORB           | EORB          | EORB          |
| \$10 _8 |                |        | LBVC        | PSHSW     | LSLD              |                   |               |               | EORD          | EORD           | EORD          | EORD          |               |                |               |               |
| \$11 _8 |                |        |             | TFM r+,r+ |                   |                   |               |               |               |                |               |               |               |                |               |               |
| _9      | ROL            | DAA    | BVS         | RTS       | ROLA              | ROLB              | ROL           | ROL           | ADCA          | ADCA           | ADCA          | ADCA          | ADCB          | ADCB           | ADCB          | ADCB          |
| \$10 _9 |                |        | LBVS        | PULSW     | ROLD              | ROLW              |               |               | ADCD          | ADCD           | ADCD          | ADCD          |               |                |               |               |
| \$11 _9 |                |        |             | TFM r-,r- |                   |                   |               |               |               |                |               |               |               |                |               |               |
| _A      | DEC            | ORCC   | BPL         | ABX       | DECA              | DECB              | DEC           | DEC           | ORA           | ORA            | ORA           | ORA           | ORB           | ORB            | ORB           | ORB           |
| \$10 _A |                |        | LPBL        | PSHUW     | DECD              | DECW              |               |               | ORD           | ORD            | ORD           | ORD           |               |                |               |               |
| \$11 _A |                |        |             | TFM r+,r  | DECE              | DECF              |               |               |               |                |               |               |               |                |               |               |
| _B      | TIM            |        | BMI         | RTI       |                   |                   | TIM           | TIM           | ADDA          | ADDA           | ADDA          | ADDA          | ADDB          | ADDB           | ADDB          | ADDB          |
| \$10 _B |                |        | LBMI        | PULUW     |                   |                   |               |               | ADDW          | ADDW           | ADDW          | ADDW          |               |                |               |               |
| \$11 _B |                |        |             | TFM r,r+  |                   |                   |               |               | ADDE          | ADDE           | ADDE          | ADDE          | ADDF          | ADDF           | ADDF          | ADDF          |
| _C      | INC            | ANDCC  | BGE         | CWAI      | INCA              | INCB              | INC           | INC           | CMPX          | CMPX           | CMPX          | CMPX          | LDD           | LDD            | LDD           | LDD           |
| \$10 _C |                |        | LBGE        |           | INCD              | INCW              |               |               | CMPY          | CMPY           | CMPY          | CMPY          |               | LDQ            | LDQ           | LDQ           |
| \$11 _C |                |        |             | BITMD     | INCE              | INCF              |               |               | CMPS          | CMPS           | CMPS          | CMPS          |               |                |               |               |
| _D      | TST            | SEX    | BLT         | MUL       | TSTA              | TSTB              | TST           | TST           | BSR           | JSR            | JSR           | JSR           | LDQ           | STD            | STD           | STD           |
| \$10 _D |                |        | LBLT        |           | TSTD              | TSTW              |               |               |               |                |               |               |               | STQ            | STQ           | STQ           |
| \$11 _D |                |        |             | LDMD      | TSTE              | TSTF              |               |               | DIVD          | DIVD           | DIVD          | DIVD          |               |                |               |               |
| _E      | JMP            | EXG    | BGT         |           |                   |                   | JMP           | JMP           | LDX           | LDX            | LDX           | LDX           | LDU           | LDU            | LDU           | LDU           |
| \$10 _E |                |        | LBGT        |           |                   |                   |               |               | LDY           | LDY            | LDY           | LDY           | LDS           | LDS            | LDS           | LDS           |
| \$11 _E |                |        |             |           |                   |                   |               |               | DIVQ          | DIVQ           | DIVQ          | DIVQ          |               |                |               |               |
| _F      | CLR            | TFR    | BLE         | SWI       | CLRA              | CLRB              | CLR           | CLR           |               | STX            | STX           | STX           |               | STU            | STU           | STU           |
| \$10 _F |                |        | LBLE        | SWI2      | CLRD              | CLR W             |               |               |               | STY            | STY           | STY           |               | STS            | STS           | STS           |
| \$11 _F |                |        |             | SWI3      | CLRE              | CLRF              |               |               | MULD          | MULD           | MULD          | MULD          |               |                |               |               |

Shaded Instructions are available on 6309 microprocessors only

Undefined opcodes generate an Illegal Instruction exception on the 6309 only.



## 6809 Undefined Opcode Behavior

Unlike the 6309 microprocessor, the 6809 does not trap illegal instructions. This section describes the behavior of the 6809 when it executes an undefined opcode. In most cases, the CPU behaves as if it had executed the instruction whose opcode value is either one less or one more than that of the undefined opcode. The Opcode Map and notes shown below describe the specific behavior of each undefined opcode. The same behavior will result when an undefined opcode is preceded by a Page 2 (\$10) or Page 3 (\$11) selector, except that 1 additional MPU cycle is consumed.

|    | DIRECT<br>\$0_           | \$1_             | REL<br>\$2_       | \$3_               | ACC. A<br>\$4_             | ACC. B<br>\$5_             | INDEX<br>\$6_            | EXTND<br>\$7_            | IMMED<br>\$8_ | DIRECT<br>\$9_ | INDEX<br>\$A_ | EXTND<br>\$B_ | IMMED<br>\$C_    | DIRECT<br>\$D_ | INDEX<br>\$E_ | EXTND<br>\$F_ |
|----|--------------------------|------------------|-------------------|--------------------|----------------------------|----------------------------|--------------------------|--------------------------|---------------|----------------|---------------|---------------|------------------|----------------|---------------|---------------|
| _0 |                          |                  | LBRA <sup>4</sup> |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _1 | NEG                      |                  |                   |                    | NEGA                       | NEGB                       | NEG                      | NEG                      |               |                |               |               |                  |                |               |               |
| _2 | NEG/<br>COM <sup>1</sup> |                  |                   |                    | NEGA/<br>COMA <sup>1</sup> | NEGB/<br>COMB <sup>1</sup> | NEG/<br>COM <sup>1</sup> | NEG/<br>COM <sup>1</sup> |               |                |               |               |                  |                |               |               |
| _3 |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _4 |                          | HCF <sup>2</sup> |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _5 | LSR                      | HCF <sup>2</sup> |                   |                    | LSRA                       | LSRB                       | LSR                      | LSR                      |               |                |               |               |                  |                |               |               |
| _6 |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _7 |                          |                  |                   |                    |                            |                            |                          |                          | 6             |                |               |               | 6                |                |               |               |
| _8 |                          | 3                |                   | ANDCC <sup>5</sup> |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _9 |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _A |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _B | DEC                      | NOP              |                   |                    | DECA                       | DECB                       | DEC                      | DEC                      |               |                |               |               |                  |                |               |               |
| _C |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               |                  |                |               |               |
| _D |                          |                  |                   |                    |                            |                            |                          |                          |               |                |               |               | HCF <sup>2</sup> |                |               |               |
| _E |                          |                  |                   | RESET <sup>7</sup> | CLRA                       | CLRB                       |                          |                          |               |                |               |               |                  |                |               |               |
| _F |                          |                  |                   |                    |                            |                            |                          |                          | 8             |                |               |               | 8                |                |               |               |

- Undefined opcodes in row 2 execute as a **NEG** instruction when the Carry bit in CC is 0, and as a **COM** instruction when the Carry bit is 1.
- Opcodes \$14, \$15 and \$CD all cause the CPU to stop functioning normally. One or more of these may be the **HCF** (Halt and Catch Fire) instruction. The HCF instruction was provided for manufacturing test purposes. Its causes the CPU to halt execution and enter a mode in which the Address lines are incrementally strobed.
- Opcode \$18 affects only the Condition Codes register (CC). The value in the Overflow bit (V) is shifted into the Zero bit (Z) while the value in the IRQ Mask bit (I) is shifted into the Half Carry bit (H). All other bits in the CC register are cleared. Execution of this opcode takes 3 MPU cycles.
- The 6809 will execute opcode \$20 as an LBRA when it is preceded by a Page 2 selector (\$10). The 6309 considers this an illegal instruction.
- Opcode \$38 behaves just like the **ANDCC** instruction (\$1C), except for the fact that it uses 1 additional MPU cycle (for a total of 4).
- Opcodes \$87 and \$C7 read and discard an 8-bit Immediate operand which follows the opcode. The value of the immediate byte is irrelevant. The Negative bit (N) in the CC register is always set, while the Zero (Z) and Overflow (V) bits are always cleared. No other bits in the Condition Codes register are affected. Each of these opcodes execute in 2 MPU cycles.
- Opcode \$3E is similar to the **SWI** instruction. It stacks the Entire register state, sets the I and F bits in the Condition Codes register and then loads the PC register with an address obtained from the **RESET** vector (\$FFFE:F). This could potentially be used as a fourth Software Interrupt instruction, so long as the code invoked by the Reset vector is able to differentiate between a software reset and a hardware reset. It does NOT set the Entire bit (E) in the CC register prior to stacking the register state. This could cause an **RTI** instruction for a Reset handler to fail to operate as expected. This opcode uses the same number of MPU cycles as **SWI** (15).
- Opcodes \$8F and \$CF are STX Immediate and STU Immediate respectively. These instructions are partially functional. Two bytes of immediate data follow the opcode. The first immediate byte is read and discarded by the instruction. The lower half (LSB) of the X or U register is then written into the second immediate byte. The Negative bit (N) in the CC register is always set, while the Zero (Z) and Overflow (V) bits are always cleared. No other bits in the Condition Codes register are affected. Each of these opcodes execute in 3 MPU cycles.

### NOTE:

This information was obtained through experimentation and may not be completely accurate. No information about how the 6809 operates when undefined opcodes are executed was ever published by Motorola.