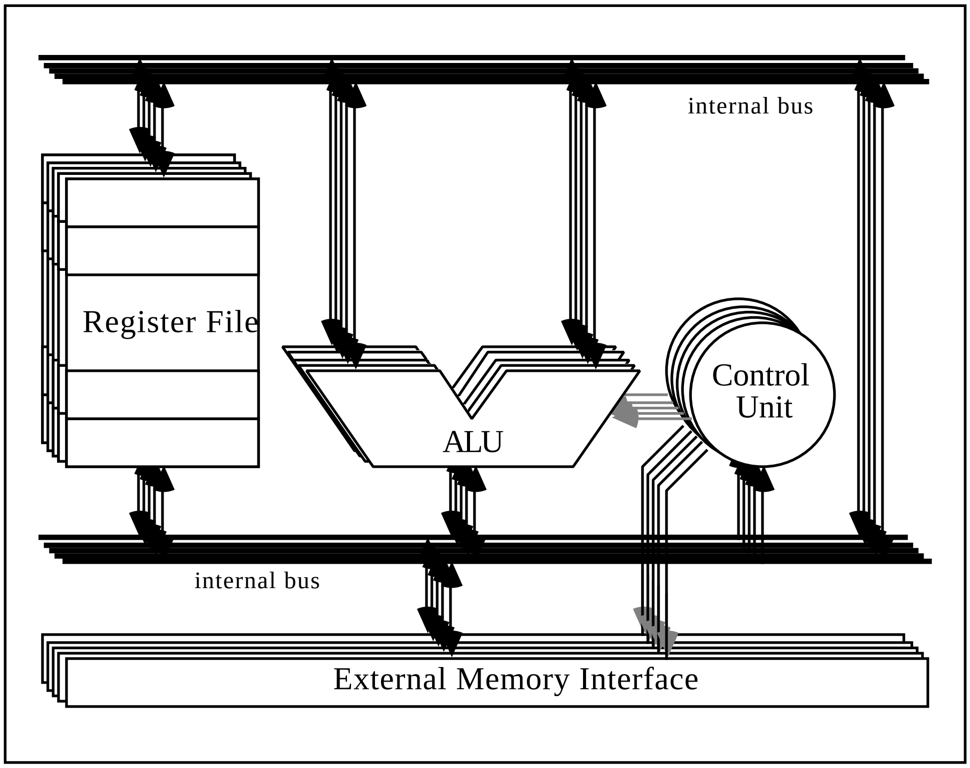
BURKE & BURKE



THE 6309 BOOK

Inside the 6309 Microprocessor

Second Edition

Chris Burke

This page intentionally left blank.

Foreword by Chris Burke

It's been 20 years since I wrote and self-published "The 6309 Book" as one of my many Color Computer projects for Burke & Burke. I've been wanting to put out a PDF version of this book for several years now, and it feels great to finally cross that goal off of my to-do list.

The recovery of "The 6309 Book" in this PDF format was quite an adventure! It began with a note I received from an old friend, Bob Swoger of the Glenside CoCo Club, asking if the club could scan an old copy of the book and distribute it online. I offered instead to reconstruct the original in PDF format.

I'd written the original manuscript in Claris Works and MacDraw, on a monochrome Mac SE/30. I'd used both the word processing and the database features of Claris Works: the page for each machine language instruction is actually a Claris Works database view.

To create the PDF version, I pulled an old lime green iMac G3 running System 9.1 out of storage. It still had the source files for the book, and the discontinued software to read and print the source files. I printed each chapter to a PDF file, saved to disk.

The old iMac I used wouldn't burn a CD, wouldn't read a thumb drive, didn't have a floppy, and used SCSI hard drives that I had no hardware to read. To get the PDF files for the chapters off of the iMac G3, I ran them through Binhex and Stuffit, then used an ancient TCP/IP program to send the archives up to an FTP site.

I downloaded the Stuffit files from the FTP site, only to find that modern versions of Stuffit couldn't read them. So.... I fired up SheepShaver, a PowerPC mac emulator that also runs System 9.1, used it to expand the Stuffit archives into a shared folder on my 27" iMac, and then assembled all of the pieces using Adobe Acrobat Pro under Mac OSX. The result was almost right; the last step was to replace some missing fonts with modern equivalents.

With that story of ancient means of production and recovery, I offer for your amusement and education "The 6309 Book," my twentieth-century assembly language primer for a marvelous 8/16 bit CPU used in aftermarket Color Computer upgrades.

Enjoy!

Chris Burke
4/22/2013

This page intentionally left blank.

TABLE OF CONTENTS

SECTION 1
PROCESSOR OVERVIEW

1.1 INTRODUCTION

Hitachi's HD63B09E microprocessor has been available for over five years. A drop-in replacement for the Motorola MC68B09E, the Hitachi processor features a x10 reduction in power consumption and additional power-saving features.

The HD63B09E is one member of a line of Hitachi microprocessors designed to replace the 6809 in low-power applications such as portable or battery-powered systems. Other processors in the "6309 family" include the HD63C09 (a 3 MHz version with internal clock generation) and the HD6309E (a 1 MHz version with external clocking).

Hitachi markets the 6309 family as exact replacements for the 6809, but as early as 1988 Japanese hobbyists discovered that the 6309 includes many advanced features. The advanced features of the 6309 were a well-kept secret in the United States and Europe until early in 1992, when a Japanese hobbyist named H. Kakugawa distributed a description on the Internet communication network under the title "A Memo on the Secret Features of 6309".

The advanced features of the 6309, when taken advantage of by software, make this processor considerably faster and more powerful than a 6809 in the same system. For example, the 6309 can copy information from peripherals or memory up to four times as fast as the 6809. The 6309 also includes new registers for greater flexibility in calculation, new machine language instructions for more efficient calculation, and new addressing modes for more powerful data manipulation.

This new second edition of The 6309 Book describes the advanced features of the 6309. It includes many corrections and additions, including information about several new 6309 addressing modes not described in previous editions.

Burke & Burke has written this book for programmers who are already familiar with the 6809 microprocessor. In it, we've concentrated on describing the differences between the two

6309. It includes many corrections and additions, including information about several new 6309 addressing modes not described in previous editions.

Burke & Burke has written this book for programmers who are already familiar with the 6809 microprocessor. In it, we've concentrated on describing the differences between the two processors and on providing application programming examples that take advantage of the 6309.

1.2 6309 Hardware Summary

The hardware of the 6309 includes an external memory / control interface, a control unit, an arithmetic logic unit (ALU), and a register file, as shown in Figure 1.1

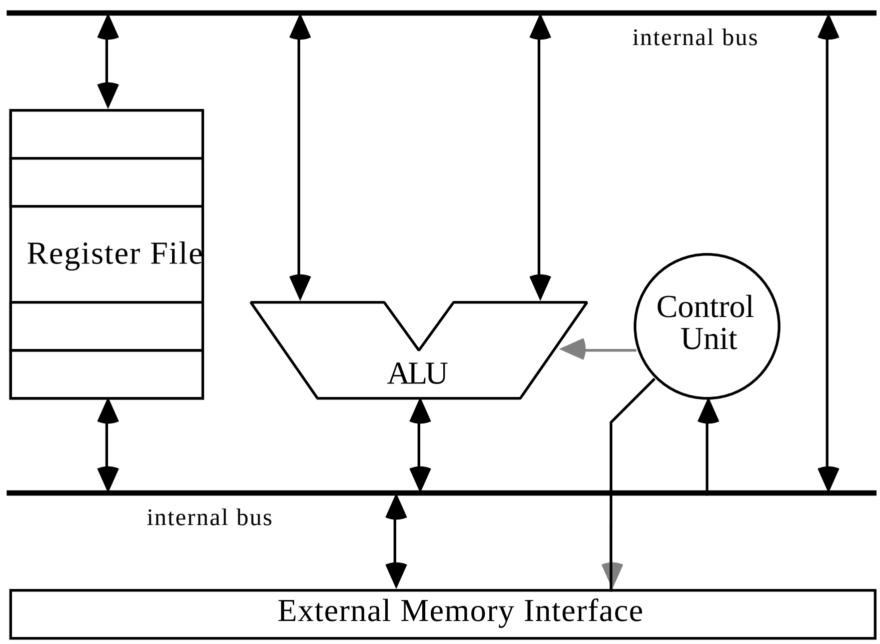


Figure 1.1 Hardware Architecture of the 6309

The external memory / control interface allows the 6309 to work as part of a complete computer system. The 6309 can perform calculations, but it has no on-chip peripherals or memory. The external interface has two parts: a memory interface, used to connect peripherals and memory (ROM / RAM) to the 6309, and a control interface, used to synchronize the 6309 with external events (RESET, interrupts).

synchronize the 6309 with external events (RESET, interrupts).

The control unit is responsible for fetching machine-language instructions from memory external to the 6309. It decodes these instructions, converting them into a sequence of internal signals that control the external interface, the ALU and the register file. The control unit also performs hardware-level interrupt processing (such as register stacking and dispatch).

The ALU performs calculations such as addition, subtraction, logical AND and logical OR. The shape of the ALU symbol suggests that it accepts two inputs and produces a single output; in truth, the ALU processes a combination of inputs (taken from the register file or external memory) and produces a combination of outputs (sent to the register file or external memory). The selection of which operands the ALU will process, and of which operation the ALU will perform, is determined by the control unit as it decodes each instruction. Some functions, such as multiplication and division, are more complex than the ALU can handle in a single operation. For these functions, the control unit performs a sequence of ALU operations, all in the process of decoding a single machine language instruction.

The register file is a list of places where the 6309 can store the results of calculations. In the 6309, the register file is divided into twelve independent registers, called CC, A, B, E, F, DP, X, Y, U, PC, SP, and V. Several of these registers are dedicated to special purposes; for example, the PC (program counter) register keeps track of the memory address from which the next machine language instruction byte will be fetched. Other registers are considered to be general-purpose; for example, both the A and B registers can be used in any arithmetic or logical calculation.

The register file contains both 8-bit and 16-bit registers. In addition, the control unit can combine certain 8-bit registers into a single register to perform 16-bit or 32-bit operations. These combined registers are called D (A and B), W (E and F), and Q (A, B, E, and F).

The lines between each of the elements in Figure 1.1 represent the flow of data and control within the 6309 processor. It is not necessary to understand the details of internal operation when programming the 6309, but a basic understanding of the functional blocks in Figure 1.1 will provide the programmer with valuable insight into the behavior of individual machine language instructions.

1.3 Software Features

control within the 6309 processor. It is not necessary to understand the details of internal operation when programming the 6309, but a basic understanding of the functional blocks in Figure 1.1 will provide the programmer with valuable insight into the behavior of individual machine language instructions.

1.3 Software Features

The 6309 provides two operating modes: Emulation Mode, and Native Mode. In Emulation Mode, the 6309 provides the same features as the 6809. These include:

- 10 addressing modes
  - 6800-compatible modes
  - Short address (direct) addressing of any 256 byte memory block
  - Long relative branches
  - Program counter (PC) relative
  - True indirect addressing
  - Indexed addressing with 0, 5, 8, or 16-bit constant offsets; 8 or 16-bit accumulator offsets; auto-increment / decrement by 1 or 2
- Improved stack manipulation
- Over 1400 instructions counting all addressing modes
- 8 x 8 unsigned hardware multiply
- 16-bit arithmetic

The 6309 adds new instructions, registers, and addressing modes - even when operating in Emulation Mode. These include:

- Two additional 8-bit registers (E, F), combinable to a 16-bit register (W)
- D and W registers combinable to a 32-bit register (Q)
- New instructions:
  - 16 x 16 bit hardware multiplication
  - 32 / 16 bit, and 16 / 8 bit, hardware division
  - Interruptible memory-to-memory block moves
  - Inter-register arithmetic and logical operations
  - Byte-oriented bit manipulation instructions
  - Single-bit arithmetic and logical operations
- New indexed addressing modes:
  - E-, F-, and W-offset from 6809 index register, with optional indirection
  - Auto-increment and auto-decrement by 2 from W register, with optional indirection
  - Zero-offset from W register
  - 16-bit offset from W register
- Automatic low power mode during SYNC and CWAI

  - Auto-increment and auto-decrement by 2 from W register, with optional indirection
  - Zero-offset from W register
  - 16-bit offset from W register
- Automatic low power mode during SYNC and CWAI
- Illegal Opcode trap interrupt
- Division by Zero trap interrupt
- 16-bit arithmetic

Native Mode further improves the operation of the processor in two ways:

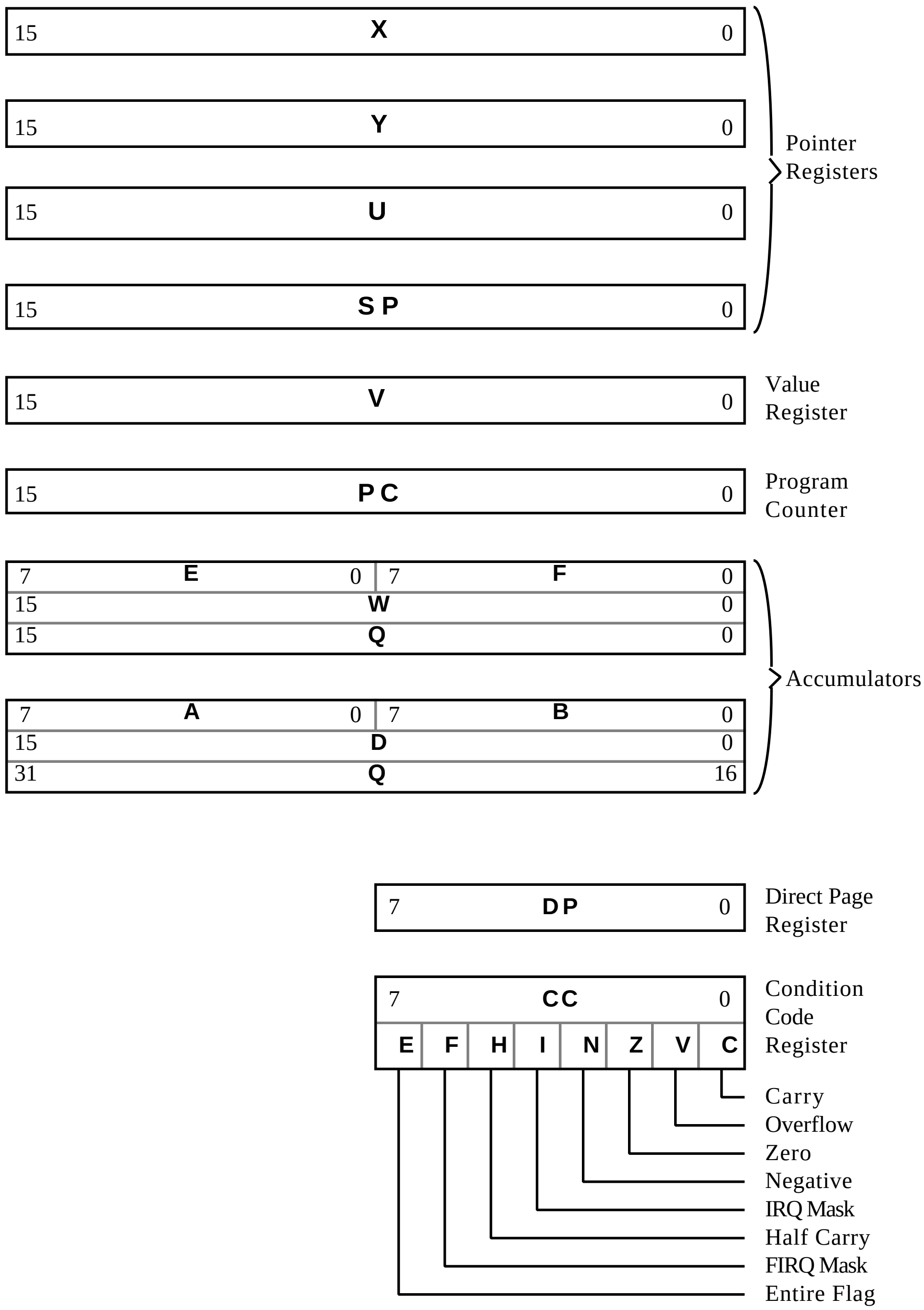
- Faster execution of many instructions (15% typical speed increase)
- W register saved and restored during interrupt processing

All of the new instructions and registers are available in both Native Mode and Emulation Mode.

1.4 Programming Model

The 6309 has twelve registers, called CC, A, B, E, F, DP, X, Y, U, SP, PC, and V. The Figure classifies these registers into several categories: the condition code register (CC), the direct page register (DP), accumulators (A, B, E, F, D, W, and Q), the program counter (PC), the value register (V) and pointer registers (W, X, Y, U, SP).

Note that the W register is both an accumulator and a pointer register. In truth, the W register is a hybrid accumulator / pointer register. This register combines many of the best features of each register type.



1.4.1 Condition Code Register

The condition code register defines the state of the processor. Each bit has an independent function.

The carry (C) bit represents the carry or borrow from an arithmetic operation. It's also used as an extra bit in shift and rotate instructions.

The overflow (V) bit indicates signed two's complement overflow, which occurs when the sign bit differs from the carry bit after an arithmetic operation.

The zero (Z) bit is set when the result of an operation is zero.

The negative (N) bit always contains the value of the most-significant bit of the result of the last operation. For two's complement arithmetic, the result is negative when the N bit is set.

The interrupt (I) bit and fast interrupt (F) bit can be set or cleared by a program. When set, the I bit disables IRQ* interrupts; the F bit disables FIRQ* interrupts when set.

The half-carry (H) bit is set when an arithmetic operation produces a carry from bit 3 to bit 4 of the result.

The entire (E) bit indicates which registers the 6309 stacked during the most recent interrupt. When set, all of the registers were stacked; when clear, only the program counter and condition code register were stacked. This bit effects the operation of the RTI instruction.

1.4.2 Direct Page Register

The value in the direct page (DP) register is used as the 8 most-significant bits of the 16-bit memory address when an instruction uses direct addressing mode.

1.4.3 Accumulators

6309 instructions manipulate 1, 8, or 16-bit data. Most of these instructions manipulate data in one of the accumulator registers.

The A and B registers are used to manipulate 8-bit data; they are called general-purpose registers, since almost any instruction can manipulate data in A or B. The E and F registers also manipulate 8-bit data, with somewhat less flexibility.

The D and W registers are 16-bit registers formed from pairs of 8-bit registers. D is a general-purpose accumulator; W can also be used for many 16-bit operations.

The A and B registers are used to manipulate 8-bit data; they are called general-purpose registers, since almost any instruction can manipulate data in A or B. The E and F registers also manipulate 8-bit data, with somewhat less flexibility.

The D and W registers are 16-bit registers formed from pairs of 8-bit registers. D is a general-purpose accumulator; W can also be used for many 16-bit operations.

Both the D and the W register function as pointer, rather than accumulator, registers under certain conditions. Several new 6309 addressing modes use W as in index register, while the 6309's block move instruction can use D to point at the data source or destination.

1.4.4 Program Counter

The program counter contains the address of the next instruction byte to be fetched by the processor. This register increments automatically each time the processor fetches an instruction byte.

Some instructions, such as JMP and RTS, modify the value of the program counter directly. This changes the address for instruction fetches, causing execution to resume at a new address.

1.4.5 Value Register

The value (V) register retains whatever value the program stores there, even across RESET.

The V register can also be used as an operand in register-to-register instructions such as EORR (exclusive-OR register-to-register) and ADDR (add register-to-register).

1.4.6 Pointer Registers

There are three kinds of pointer registers on the 6309.

The first group, index registers, includes the X and Y registers. These registers are most useful as index registers or loop counters, but they may also be used for other purposes.

The second group, stack pointer registers, includes the U and SP registers. The SP register

There are three kinds of pointer registers on the 6309.

The first group, index registers, includes the X and Y registers. These registers are most useful as index registers or loop counters, but they may also be used for other purposes.

The second group, stack pointer registers, includes the U and SP registers. The SP register points at an area of memory used to automatically store register values during interrupts, and return addresses during subroutine calls. This area is called the system stack. The U register points at another area of memory, which my be used as a temporary value stack or for other purposes.

The third group, special-purpose pointer registers, includes the D and W registers. The 6309 includes 8 new addressing mode variations which use W as an index register. One new 6309 instruction, TFM (block move), can use the D register as a source or destination data pointer.

1.5 Processor Modes

The 6309 contains a special register, called the mode (MD) register. This register cannot be used for calculation, but values stored in this register effect the operation of the processor.

We have already mentioned that the 6309 supports two operating modes: Emulation Mode, and Native Mode. The programmer can select between these modes through the least-significant bit of the MD register. When this bit is clear, the processor operates in Emulation Mode. When this bit is set, the processor operates in Native Mode.

Other bits of the MD register are described in the Applications section.

1.6 Processor Address Map

The 6309 does not have any on-chip peripherals, so no memory locations are reserved for this purpose. Computer designers using the 6309 can place peripherals at any address in the processor's 64K map.

The highest 16 addresses of the processor's map are reserved for interrupt vectors. Each of these vectors contains a two-byte value, selected by the computer or software designer, that is the address of a subroutine to be called whenever the associated interrupt occurs. The Table shows the type of interrupt assigned to each vector:

The 6309 does not have any on-chip peripherals, so no memory locations are reserved for this purpose. Computer designers using the 6309 can place peripherals at any address in the processor's 64K map.

The highest 16 addresses of the processor's map are reserved for interrupt vectors. Each of these vectors contains a two-byte value, selected by the computer or software designer, that is the address of a subroutine to be called whenever the associated interrupt occurs. The Table shows the type of interrupt assigned to each vector:

Address	Interrupt Type
$FFFE	Processor reset (RESET* line)
$FFFC	Non-maskable interrupt (NMI* line)
$FFFA	Software interrupt (SWI instruction)
$FFF8	Interrupt (IRQ* line)
$FFF6	Fast interrupt (FIRQ* line)
$FFF4	Second software interrupt (SWI2 instruction)
$FFF2	Third software interrupt (SWI3 instruction)
$FFF0	Illegal Opcode and Division by Zero Trap (exception)

SECTION 2
ADDRESSING MODE REFERENCE

2.1 INTRODUCTION

This section describes each of the addressing modes of the 6309 processor. Each description includes 8 fields:

Name	The name of the addressing mode (sometimes abbreviated) in a box at the upper left of the description.
Synopsis	A brief summary of what the instruction does. The Synopsis for each instruction is printed along the line at the top of its page.
Source Forms	The way the addressing mode is written in 6309 assembly language. This item uses symbols to represent variable portions of the addressing mode. Note that in 6309 source code, the addressing mode follows the instruction mnemonic on the same line, separated from it by one or more spaces.
EA Calculation	A symbolic, formal description of hthe addressing mode calculates the address of the instruction's operand.
Description	Text describing the addressing mode in more detail.
Comments	Additional information about special considerations when using this addressing mode.
Examples	A short assembly language program fragment illustrating the use of the addressing mode.
Post-Bytes	A definition of the machine-language (hexadecimal) encoding of the addressing mode as it must be stored in memory after the basic instruction.

2.2 SOURCE FORMS FIELD NOTATION

The Source Forms field of each description uses symbols to describe how a programmer would code the addressing mode in assembly language.

The Source Forms field of each description uses symbols to describe how a programmer would code the addressing mode in assembly language.

Upper-case letters and punctuation marks represent literal parts of the assembly language addressing mode.

Lower-case letters represent variable portions of the instruction. The programmer must replace lower-case letters occuring in Source Forms section with appropriate addressing mode details. Each lower-case symbol identifies the general type of an operand according to the following table:

Symbol	Meaning
`r`	Any 1- character register name valid for this addressing mode (e.g. X)
`ea, n`	Any expression, which may include labels, numeric constants, and calculations so long as the value of the expression can be completely determined during assembly.

2.3 EA CALCULATION FIELD NOTATION

The EA Calculation item of each description tells how the addressing mode calculates the address of the operand.

Each 6309 addresing mode performs a different calculation when executed, but all of the operations follow a general pattern: whatever the operation, it produces an operand address `m` from some combination of values stored in internal registers and external memory. The EA Calculation field of each description uses symbols to describe these calculations. The Table explains the meaning of each symbol:

Symbol	Meaning
`<-`	Assignment; the entity on the left of this symbol is assigned (”takes on”) the value of the expression on right of this symbol.
`‘`	Placed after a register name, idicates “after decrementing”.
`(signed)`	Placed before a register name, this symbol indicates that the 6309 treats the value of the register as a signed number when converting it from 5 or 8 bits to 16 bits (e.g. `m<-(signed)B` means “m takes on the signed value of B”).
`+, -`	Addition or subtraction of the expressions on the left and right sides of the symbol.
`(…)`	The value of the memory location(s) accessed by using the enclosed expression as an address.

	sides of the symbol.
`(…)`	The value of the memory location(s) accessed by using the enclosed expression as an address.
`sizeof(…)`	The size, in bytes, of the item in the parenthesis (e.g. `sizeof(basic instruction)` is the size of the instruction's Encoding field described in Section, and `sizeof(post-bytes)` is the size of the addressing mode post-bytes described in Section 2).
`r`	The value of a register, identified with the same symbol in the Source Forms item of the instruction description.
`m`	The calculated address of the operand.
`n, ea`	The value of an expression identified with the same symbol in the Source Forms field.
`;`	Marks the end of one calculation and the beginning of another, performed by the same addressing mode.

2.4 POST-BYTES FIELD

The Post-Bytes field gives a rule for encoding the addressing mode as a series of 8-bit hexadecimal values (6309 machine language). If you're using an assembler program, you'll probably never need to know the encoding of individual addressing modes. The information in this field is provided for debugging, hand-coding, and special-purpose applications.

When hand-coding an entire instruction, first code the basic instruction as described in the Encoding field of the instruction's description (Section 3). If the Encoding field does not include the notation "`(+ post-bytes)`", the instruction is completely coded using the information provided in Section 3. If the Encoding field does include "`(+ post-bytes)`", you must encode additional bytes as described in Section 2 for the desired addressing mode. For example, to encode the instruction:

```
LDA     3,X
```

we first find the description of the 8-bit LD instruction in Section 3. The description lists several possible encodings; we select the encoding for register A with indexed address mode:

```
$A6 (+ post-bytes)
```

Since the encoding indicates post-bytes, we refer to the description of the register indexed addressing mode in Section 2. Here again the description lists several possible encodings; we select the encoding for register X with 5-bit offset, and plug in the desired offset of +3 to indexed addressing mode in Section 2. Here again the description lists several possible encodings; we select the encoding for register X with 5-bit offset, and plug in the desired offset of +3 to get post-bytes of:

```
$03
```

Now we combine the basic instruction encoding and the post-byte encoding to obtain the complete encoding for this instruction:

```
$86 $03
```

Refer to Section 3, Instruction Reference, for additional information.

Direct

Least-Significant Byte of Literal Address

Source Forms:

```
ea
<ea
```

EA Calculation: m <- DP:ea

Description:

The direct addressing mode takes a 1-byte value from the post-byte(s), advancing the program counter past the post-bytes after instruction execution.

The value is used as the 8 least-significant bits of the 16-bit memory address of the operand. The contents of the DP (Direct Page) register are used as the 8 most-significant address bits.

Comments:

The address ea in the source form may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Direct addressing accesses 1-, 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
STB     <$7E    ;Store B at address $7E
JSR     <QUICK  ;Call subroutine at address ‘QUICK’
```

Post-Bytes:

$JJ

Where JJ is the least-significant byte of the address of the operand.

Extd. Indirect

Literal Address of Pointer to Operand

Source Forms:

```
[ea]        ;indexed form
```

EA Calculation: m <- (ea)

Description: The extended indirect addressing mode takes a 2-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution. The value is used as the 16-bit memory address of a 16-bit pointer to the operand.

Comments: The address ea in the source form may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Examples:

```
JMP     [$FFFE] ;Jump to address at $FFFE (RESET vector)
```

Post-Bytes:

```
$9FJJKK
```

Where JJ is the most-significant byte, and KK is the least-significant byte, of the address of a pointer to the operand.

Extended

Source Forms: `>ea`

EA Calculation: m <- ea

Description: The extended addressing mode takes a 2-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution. The value is used as the 16-bit memory address of the operand.

Comments: The address ea in the source form may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Examples:

```
LDD     >$1720  ;Load D with value stored at address $1720
JMP     >START  ;Jump to address ‘START’
```

Post-Bytes: $JJKK

Where JJ is the most-significant byte, and KK is the least-significant byte, of the address of the operand.

Immediate (8-

Source Forms: `#n`

EA Calculation: m <- PC + sizeof(basic instruction)

Description: The immediate addressing mode takes a 1-byte operand from the post-byte(s), advancing the program counter past the post-bytes after instruction execution.

The interpretation of the operand byte (e.g. signed, unsigned, bit field, or other) depends on the instruction but does not effect the encoding of the operand.

Comments: The operand n in the source form may be any expression that produces an 8-bit result or any expression preceded by the symbol < (forcing an 8-bit value); a list of register names (which the assembler converts into an 8-bit field).

Regardless of how the operand is represented in source form, the encoding of the instruction includes only the actual operand value, and not any of the calculation done to initially obtain it.

Examples:

```
LDA     #$20    ;Load A with the value $20 (32 decimal)
PSHS    Y,X,B   ;Push Y, X, and B (equiv. to #$34 operand)
```

Post-Bytes: `$JJ`

Where JJ is the operand value.

Source Forms: #n

EA Calculation: m <- PC + sizeof(basic instruction)

Description: The immediate addressing mode takes a 2-byte operand from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The interpretation of the operand word (e.g. signed, unsigned, bit field, or other) depends on the instruction but does not effect the encoding of the operand.

Comments: The operand n in the source form may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

Regardless of how the operand is represented in source form, the encoding of the instruction includes only the actual operand value, and not any of the calculation done to initially obtain it.

Examples:

```
LDD     #$1720  ;Load D with the value $1720
LDX     #65432  ;Load X with the value 65,432
```

Post-Bytes: $JJKK

Where JJ is the most-significant byte, and KK is the least-significant byte, of the operand value.

Immediate *(32-*

Source Forms: #n

EA Calculation: m <- PC + sizeof(basic instruction)

Description: The immediate addressing mode takes a 4-byte operand from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The interpretation of the operand long word (e.g. signed, unsigned, bit field, or other) depends on the instruction but does not effect the encoding of the operand.

Comments: The operand n in the source form may be any expression that produces a 32-bit result or any expression preceded by the symbol >> (forcing a 32-bit value).

Regardless of how the operand is represented in source form, the encoding of the instruction includes only the actual operand value, and not any of the calculation done to initially obtain it.

Examples:

```
LDQ     #$17204598  ;Load Q with the value $17204598
```

Post-Bytes: `$WWJJKKZZ`

Where WW is the most-significant byte, and ZZ is the least-significant byte, of the operand value.

Inherent

No Additional Operands

Source Forms:

EA Calculation:

Description: When using Inherent addressing, the instruction itself implies its operands.

This addressing mode has no additional source coding, effective address calculation, or machine language encoding.

Comments:

Examples:

```
ABX         ;Add B to X (inherent addressing)
CLRD        ;Clear D register (inherent addressing)
```

Post-Bytes:

Post-Inc Indirect

Register Points to Pointer to Operand

Source Forms: `[,r++]        ;indexed form`

EA Calculation: m <- (r); r' <- r+2

Description: The post-increment indirect addressing mode uses the value stored in register `r`, as the address of a 16-bit pointer to the operand.

After accessing the operand, but before completing execution of the instruction, this addressing mode adds $0002 to register `r`.

Comments: Post-increment indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 16-bit data, since it leaves register `r` pointing at the "next" data item.

Examples:

```
LDD     [,X++] ;Load D from pointer at X, advancing X by 2
LDY     [,W++] ;Load Y from pointer at W, advancing W by 2
```

Post-Bytes:

```
$M1        (M = address register; 9=X, B=Y, D=U, F=S)
$D0        (register W)
```

Post-Inc. ***(8-Bit)***

Register Points to Operand

Source Forms: `,r+` `;indexed form`

EA Calculation: m <- r; r' <- r+1

Description: The post-increment indexed addressing mode uses the value stored in register `r`, as the address of the operand.

After accessing the operand, but before completing execution of the instruction, this addressing mode adds $0001 to register `r`.

Comments: Post-increment indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 8-bit data, since it leaves register `r` pointing at the “next” data item.

Examples:

```
LDA     ,X+    ;Load A from address in X, advancing X by 1
```

Post-Bytes:

```
$M0       (M = address register; 8=X, A=Y, C=U, E=S)
```

Post-Inc. ***(16-Bit)***

Source Forms: `,r++` `;indexed form`

EA Calculation: m <- r; r’ <- r+2

Description: The post-increment indexed addressing mode uses the value stored in register `r`, as the address of the operand.

After accessing the operand, but before completing execution of the instruction, this addressing mode adds $0002 to register `r`.

Comments: Post-increment indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 16-bit data, since it leaves register `r` pointing at the “next” data item.

Examples:

```
LDD     ,X++   ;Load D from address in X, advancing X by 2
LDY     ,W++   ;Load Y from address in W, advancing W by 2
```

Post-Bytes:

```
$M1       (M = address register; 8=X, A=Y, C=U, E=S)
$CF       (register W)
```

Pre-Dec Indirect

Source Forms:

```
[,--r]        ;indexed form
```

EA Calculation: r’ <- r-2; m <- (r’)

Description: The pre-decrement indirect addressing mode subtracts $0002 from the value stored in register `r`, and then uses the value as the address of a 16-bit pointer to the operand.

Comments: Pre-decrement indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 16-bit data, since it leaves register `r` pointing at the “next” data item.

Examples:

```
LDD     [,--X] ;Double-decrement X, load D from pointer at X
LDU     [,--W] ;Double-decrement W, load U from pointer at W
```

Post-Bytes:

```
$M3       (M = address register; 9=X, B=Y, D=U, F=S)
$F0       (register W)
```

Pre-Dec. ***(8-Bit)***

Register Points to Operand

Source Forms: `,-r` `;indexed form`

EA Calculation: r’ <- r-1; m <- r’

Description: The pre-decrement indexed addressing mode subtracts $0001 from the value in register `r`, and then uses the decremented value as the address of the operand.

Comments: Pre-decrement indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 8-bit data, since it leaves register `r` pointing at the “next” data item.

Examples:

```
LDA     ,-X    ;Decrement X, load A from address in X
```

Post-Bytes:

```
$M2       (M = address register; 8=X, A=Y, C=U, E=S)
```

Pre-Dec. *(16-Bit)*

Source Forms: `,--r ;indexed form`

EA Calculation: r' <- r-2; m <- r'

Description: The pre-decrement indexed addressing mode subtracts $0002 from the value in register `r`, and then uses the decremented value as the address of the operand.

Comments: Pre-decrement indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction. It is most useful when accessing a block of contiguous 16-bit data, since it leaves register `r` pointing at the "next" data item.

Examples:

```
LDD     ,--X   ;Double-decrement X, load D from address in X
LDY     ,--W   ;Double-decrement W, load Y from address in W
```

Post-Bytes:

```
$M3       (M = address register; 8=X, A=Y, C=U, E=S)
$EF       (register W)
```

Reg. Indexed

Register Points to Operand

Source Forms: `,r ;indexed form`

EA Calculation: m <- r

Description: The register indexed addressing mode uses the value stored in register `r`, as the address of the operand.

Comments: Register indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

The assembler automatically selects this addressing mode when the offset value is zero and no < or > symbol appears in the source form.

Examples:

```
LDX     ,X    ;Load X from address stored in register X
LDU     ,W    ;Load U from address stored in register W
```

Post-Bytes:

```
$M4       (M = address register; 8=X, A=Y, C=U, E=S)
$8F       (register W)
```

Reg. Indexed (A)

A + Register Points to Operand

Source Forms: `A,r ;indexed form`

EA Calculation: m <- r + (signed)A

Description: The A accumulator indexed addressing mode takes a 1-byte value from the A register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: A accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     A,X     ;Load X from address X+A
```

Post-Bytes:

```
$M6        (M = address register; 8=X, A=Y, C=U, E=S)
```

Reg. Indexed (B)

B + Register Points to Operand

Source Forms:

```
B,r        ;indexed form
```

EA Calculation: m <- r + (signed)B

Description: The B accumulator indexed addressing mode takes a 1-byte value from the B register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: B accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     B,X     ;Load X from address X+B
```

Post-Bytes:

```
$M5        (M = address register; 8=X, A=Y, C=U, E=S)
```

Reg. Indexed (D)

D + Register Points to Operand

Source Forms: `D,r ;indexed form`

EA Calculation: m <- r + D

Description: The D accumulator indexed addressing mode takes a 2-byte value from the D register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: D accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     D,X     ;Load X from address X+D
```

Post-Bytes: `$MB        (M = address register; 8=X, A=Y, C=U, E=S)`

Reg. Indexed (E)

E + Register Points to Operand

Source Forms: `E,r        ;indexed form`

EA Calculation: m <- r + (signed)E

Description: The E accumulator indexed addressing mode takes a 1-byte value from the E register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: E accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     E,X     ;Load X from address X+E
```

Post-Bytes: `$M7        (M = address register; 8=X, A=Y, C=U, E=S)`

Reg. Indexed (F)

F + Register Points to Operand

Source Forms: `F,r ;indexed form`

EA Calculation: m <- r + (signed)F

Description: The F accumulator indexed addressing mode takes a 1-byte value from the F register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: F accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     F,X     ;Load X from address X+F
```

Post-Bytes:

```
$MA        (M = address register; 8=X, A=Y, C=U, E=S)
```

Reg. Indexed (W)

W + Register Points to Operand

Source Forms: `W,r ;indexed form`

EA Calculation: m <- r + W

Description: The W accumulator indexed addressing mode takes a 2-byte value from the W register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: W accumulator indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     W,X     ;Load X from address X+W
```

Post-Bytes: `$ME        (M = address register; 8=X, A=Y, C=U, E=S)`

Source Forms:

```
ea,r        ;indexed form
<ea,r       ;indexed form
```

EA Calculation: m <- r + (signed)ea

Description: The register indexed addressing mode takes a 1-byte value from the post-byte, advancing the program counter past the post-byte after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: The source form address ea may be any expression that produces a 5-bit result, or any expression preceded by the symbol `< (forcing an 8-bit value).`

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Register indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

The assembler automatically selects a 5-bit or 8-bit offsets based on the offset value.

Examples:

```
LDX     -5,X   ;Load X from address X-5
```

Post-Bytes: $MM (MM = address register and offset; 00+=X, 20+=Y, 40+=U, 60+=S)

The 5-bit signed offset value is stored in the 5 least-significant bits of MM. For example, MM = 3F indicates a

Reg. Indexed (8-

Offset + Register Points to Operand

Source Forms:

```
ea,r        ;indexed form
<ea,r       ;indexed form
```

EA Calculation: m <- r + (signed)ea

Description: The register indexed addressing mode takes a 1-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: The source form address ea may be any expression that produces an 8-bit result, or any expression preceded by the symbol < (forcing an 8-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Register indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

The assembler automatically selects a 5-bit or 8-bit offsets based on the offset value.

Examples:

```
LDX     32,X    ;Load X from address X+32
```

Post-Bytes:

```
$M8JJ       (M = address register; 8=X, A=Y, C=U, E=S)
```

JJ is the signed 8-bit offset to the operand.

Reg. Indexed *(16-*

Offset + Register Points to Operand

Source Forms: `>ea,r ;indexed form`

EA Calculation: m <- r + ea

Description: The register indexed addressing mode takes a 2-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of the operand.

Comments: The source form address `ea` may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Register indexed addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     $1000,X  ;Load X from address X+$1000
LDD     $1000,W  ;Load D from address W+$1000
```

Post-Bytes:

```
$M9JJKK     (M = address register; 8=X, A=Y, C=U, E=S)
$AFJJKK     (register W)
```

JJ is the most-significant byte, and KK is the least-significant byte, of the signed 16-bit offset to the operand.

Reg. Indirect

Register Points to Pointer to Operand

Source Forms: `[,r]` ;indexed form

EA Calculation: m <- (r)

Description: The register indirect addressing mode uses the value stored in register `r`, as the address of a 16-bit pointer to the operand.

Comments: Register indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

The assembler automatically selects this addressing mode when the offset value is zero and no < or > symbol appears in the source form.

Examples:

```
LDX     [,X]   ;Load X from pointer at address stored in X
LDS     [,W]   ;Load S from pointer at address stored in W
```

Post-Bytes:

```
$M4       (M = address register; 9=X, B=Y, D=U, F=S)
$90       (register W)
```

Reg. Indirect (A)

A + Register Points to Pointer to Operand

Source Forms: `[A,r]      ;indexed form`

EA Calculation: m <- (r + (signed)A)

Description: The A accumulator indirect addressing mode takes a 1-byte value from the A register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: A accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [A,X]     ;Load X from address stored at address X+A
```

Post-Bytes:

```
$M6         (M = address register; 9=X, B=Y, D=U, F=S)
```

Reg. Indirect (B)

B + Register Points to Pointer to Operand

Source Forms: `[B,r]      ;indexed form`

EA Calculation: m <- (r + (signed)B)

Description: The B accumulator indexed addressing mode takes a 1-byte value from the B register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: B accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [B,X]   ;Load X from address stored at address X+B
```

Post-Bytes: `$M5        (M = address register; 9=X, B=Y, D=U, F=S)`

Reg. Indirect (D)

D + Register Points to Pointer to Operand

Source Forms: `[D,r]      ;indexed form`

EA Calculation: m <- (r + D)

Description: The D accumulator indexed addressing mode takes a 2-byte value from the D register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: D accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [D,X]   ;Load X from address stored at address X+D
```

Post-Bytes:

```
$MB         (M = address register; 9=X, B=Y, D=U, F=S)
```

Reg. Indirect (E)

E + Register Points to Pointer to Operand

Source Forms: `[E,r]      ;indexed form`

EA Calculation: m <- (r + (signed)E)

Description: The E accumulator indirect addressing mode takes a 1-byte value from the E register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: E accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [E,X]     ;Load X from address stored at address X+E
```

Post-Bytes:

```
$M7         (M = address register; 9=X, B=Y, D=U, F=S)
```

Reg. Indirect (F)

F + Register Points to Pointer to Operand

Source Forms: `[F,r]      ;indexed form`

EA Calculation: m <- (r + (signed)F)

Description: The F accumulator indirect addressing mode takes a 1-byte value from the F register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: A accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [F,X]     ;Load X from address stored at address X+F
```

Post-Bytes:

```
$MA         (M = address register; 9=X, B=Y, D=U, F=S)
```

Reg. Indirect (W)

W + Register Points to Pointer to Operand

Source Forms: `[W,r]      ;indexed form`

EA Calculation: m <- (r + W)

Description: The W accumulator indexed addressing mode takes a 2-byte value from the W register, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: W accumulator indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [D,X]   ;Load X from address stored at address X+D
```

Post-Bytes:

```
$ME         (M = address register; 9=X, B=Y, D=U, F=S)
```

Reg. Indirect (8- — Offset + Reg. Points to Pointer to Operand

Source Forms:

```
[ea,r]      ;indexed form
[<ea,r]     ;indexed form
```

EA Calculation: m <- (r + (signed)ea)

Description: The register indirect addressing mode takes a 1-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: The source form address `ea` may be any expression that produces an 8-bit result, or any expression preceded by the symbol < (forcing an 8-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Register indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [32,X]  ;Load X from address stored in 32,X
```

Post-Bytes:

```
$M8JJ       (M = address register; 9=X, B=Y, D=U, F=S)

JJ is the signed 8-bit offset to the 16-bit address of the
operand.
```

Source Forms: `[>ea,r]      ;indexed form`

EA Calculation: m <- (r + ea)

Description: The register indirect addressing mode takes a 2-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in register `r`, to calculate the address of a 16-bit pointer to the operand.

Comments: The source form address `ea` may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Register indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [$1000,X]  ;Load X from address stored in $1000,X
LDQ     [$1000,W]  ;Load Q from address stored in $1000,W
```

Post-Bytes:

```
$M9JJKK     (M = address register; 9=X, B=Y, D=U, F=S)
$B0JJKK     (register W)
```

JJ is the most-significant byte, and KK is the least-significant byte, of the signed 16-bit offset to the 16-bit

Source Forms: `[ea,PCR]     ;indexed form`

EA Calculation: m <- (PC + sizeof(basic instruction) + (signed)ea + 1)

Description: The relative indirect addressing mode takes a 1-byte value from the post-byte(s), advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in the PC register, to calculate the address of a 16-bit pointer to the operand.

An offset value of $00 would indicate that the address of the operand is stored in the word at the memory location immediately following the complete instruction.

Comments: The source form address `ea` may be any expression that produces an 8-bit result or any expression preceded by the symbol < (forcing an 8-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Relative indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [PNTR,PCR]  ;Load X from address stored in PNTR,PCR
```

Post-Bytes: `$9CJJ`

`Where JJ is the signed 8-bit offset to the 16-bit address of the operand.`

Source Forms: `[>ea,PCR]    ;indexed form`

EA Calculation: m <- (PC + sizeof(basic instruction) + ea + 2)

Description: The relative indirect addressing mode takes a 2-byte value from the post-bytes, advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in the PC register, to calculate the address of a 16-bit pointer to the operand.

An offset value of $0000 would indicate that the address of the operand is stored in the word at the memory location immediately following the complete instruction.

Comments: The source form address `ea` may be any expression that produces an 8-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Relative indirect addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LDX     [>PNTR,PCR] ;Load X from address stored in PNTR,PCR
```

Post-Bytes: `$9DJJKK`

Where JJ is the most-significant byte, and KK is the least-significant byte, of the signed 16-bit offset to the 16-bit address of the operand.

Relative (8-Bit)

Source Forms:

```
ea        ;branch form
ea,PCR    ;indexed form
```

EA Calculation: m <- PC + sizeof(basic instruction) + (signed)ea + sizeof(post-bytes)

Description:

The relative addressing mode takes a 1-byte value from the post-byte(s), advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in the PC register, to calculate the address of the operand. An offset value of $00 would indicate the memory location immediately following the complete instruction.

Comments:

The address ea in the source form may be any expression that produces an 8-bit result or any expression preceded by the symbol < (forcing an 8-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Relative addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
BRA     EXIT      ;Branch to address 'EXIT'
LDA     TEMP,PCR  ;Load A from TEMP
                  ; (using TEMP's offset relative to the
                  ; program counter)
```

Post-Bytes:

```
$JJ       (branch form only)
$8CJJ     (indexed form only)
```

Where JJ is the signed 8-bit offset to the operand.

Relative *(16-Bit)*

Source Forms:

```
ea        ;branch form
ea,PCR    ;indexed form
```

EA Calculation: m <- PC + sizeof(basic instruction) + ea + sizeof(post-bytes)

Description:

The relative addressing mode takes a 2-byte value from the post-byte(s), advancing the program counter past the post-bytes after instruction execution.

The value is used as a signed offset from the value stored in the PC register, to calculate the address of the operand. An offset value of $0000 would indicate the memory location immediately following the complete instruction.

Comments:

The address ea in the source form may be any expression that produces a 16-bit result or any expression preceded by the symbol > (forcing a 16-bit value).

The encoding of the instruction includes only the actual address value, and not any of the calculation done to initially obtain it.

Relative addressing accesses 8-, 16-, or 32-bit data, based on the instruction.

Examples:

```
LBRA    EXIT      ;Long ranch to address ‘EXIT’
LDA     >TEMP,PCR ;Load A from TEMP
                  ; (using TEMP’s offset relative to the
                  ; program counter)
```

Post-Bytes:

```
$JJKK       (branch form only)
$8DJJKK     (indexed form only)
```

Where JJ is the most-significant byte, and KK is the least-significant byte, of the signed 16-bit offset to the operand.

SECTION 3
INSTRUCTION REFERENCE

3.1 INTRODUCTION

This section describes each of the instructions of the 6309 processor. Each description includes 10 fields:

Mnemonic	A 2-5 character symbol for the instruction. The Mnemonic for each instruction is printed inside the rounded box at the upper left of its page.
Synopsis	A brief summary of what the instruction does. The Synopsis for each instruction is printed along the line at the top of its page.
Source Forms	The way the instruction is written in 6309 assembly language. This item uses symbols to represent variable portions of the instruction.
Operation	A symbolic, formal description of the instruction's operation.
Condition Codes	Itemizes the effect that the instruction has on each bit of the condition code register (register CC).
Description	Text describing the operation and operands of the instruction.
Addressing Modes	A list of the addressing modes allowed for the instruction.
Comments	Additional information about special considerations when using this instruction.
Examples	A short assembly language program fragment illustrating the use of the instruction.
Encoding	A definition of the machine-language (hexadecimal) encoding of the instruction as it must be stored in memory.

3.2 SOURCE FORMS FIELD NOTATION

The Source Forms field of each description uses the mnemonic and other symbols to memory.

3.2 SOURCE FORMS FIELD NOTATION

The Source Forms field of each description uses the mnemonic and other symbols to describe how a programmer would code the instruction assembly language.

Upper-case letters and punctuation marks represent literal parts of the assembly language instruction. For example, the notation:

```
ABX
```

means that this assembly language instruction consists of the letters A, B, and X in sequence. A space represents one or more “white space” characters, which may be any combination of spaces and tabs.

Lower-case letters represent variable portions of the instruction. The programmer must replace lower-case letters occurring in Source Forms section with appropriate descriptions of the instruction’s operands. The Description and Encoding sections provide information about valid operands for each instruction. In addition, each lower-case symbol identifies the general type of an operand according to the following table:

Symbol	Meaning
`r`	Any 1- character register name valid for this instruction (e.g. A)
`rr, r1, r2`	Any 1- or 2-character register name valid for this instruction (e.g. DP)
`n, k`	A numeric constant (3–8 bits, depending on the instruction). The programmer can use a predefined named constant instead of a number.
`qq`	A direct page (8-bit) constant memory address
`p`	Any valid operand for immediate, extended, direct, or indexed addressing modes (e.g. [3,X])
`pp`	Any valid operand for extended, direct, or indexed addressing modes (e.g. [3,X])
`id`	A signed constant displacement (8-bit or 16-bit depending on the instruction) relative to the address immediately following the last byte of this instruction’s machine language encoding.
`ea`	An “effective address”, which is allowed to be any valid operand for extended, direct, or indexed addressing modes (e.g. [3,X])
`rl`	A list of 1- or 2-character register names, separated only by commas (e.g. SP,D,X,Y)

From the table and the description of SUB, we find that:

From the table and the description of SUB, we find that:

```
SUBD #$3000
```

is an assembly language instruction that fits the Source Forms description:

```
SUBr p
```

In this example, we have replaced ‘r’ with ‘D’ (for the D register), and ‘p’ with ‘#$3000’ (for immediate addressing, constant value $3000) to obtain the actual assembly language instruction.

3.3 OPERATION FIELD NOTATION

The Operation item of each description tells what the instruction does when executed.

Each 6309 instruction performs a different operation when executed, but all of the operations follow a general pattern: whatever the operation, it modifies some combination of internal registers, external memory, and the processor’s internal controls. The Operation item of each description uses symbols to describe these modifications. The Table explains the meaning of each symbol:

Symbol	Meaning
`<-`	Assignment; the entity on the left of this symbol is assigned (”takes on”) the value of the expression on right of this symbol.
`’`	Placed after a register name or memory address, indicates “after execution”. When this symbol is not present, the description refers to “before execution”. (e.g. `A’<-B` means “A after execution takes on the value of B before execution”).
`(unsigned)`	Placed before a register name, this symbol indicates that the 6309 treats the value of the register as an unsigned number when converting it from 8 bits to 16 bits (e.g. `X’<-(unsigned)B` means “X after execution takes on the unsigned value of B before execution”).
`~`	Placed before an expression, indicates the one’s complement of the expression.
`+, -, *, /, %`	Addition, subtraction, multiplication, division, or division remainder of the expressions on the left and right sides of the symbol.
`&, |, ^`	Bit-wise AND, OR or Exclusive-OR of the expressions on the left and right sides of the symbol.
`>>, <<`	Shift the expression on the left by the number of bits indicated by the expression on the right. >> indicates a shift to the right, while <<

`>>, <<`	Shift the expression on the left by the number of bits indicated by the expression on the right. >> indicates a shift to the right, while << indicates a shift to the left.
`=, <>`	Compares the expressions on the right and left for equal (=) or not equal (<>).
`.`	Indicates a single bit of the expression on the left, selected by the expression on the right (e.g. `B.5'` means "bit 5 of B after execution"). The expression on the right may also be the 1-character name of a condition code register bit (e.g. `CC.C` means "the carry bit before execution").
`:`	Indicates a number formed from the expression on the left and the expression on the right, by joining them together (e.g. `$33:$47` is the same as `$3347`)
`[…]`	Used to group together an enclosed expression; or list of expressions; no operation.
`(…)`	The value of the memory location(s) accessed by using the enclosed expression as an address.
`sizeof(…)`	The size, in bytes, of the register named in the parenthesis (e.g. `sizeof(D)` is 2, and `sizeof(A)` is 1).
`r, rr, r1, r2`	The value of a register, identified with the same symbol in the Source Forms item of the instruction description.
`z`	An individual register selected from a list of registers specified in the instruction.
`m, ea`	The address specified by the instruction's addressing mode operand. Usually used with `(...)` (e.g. `(m)`)
`n, k, qq, id`	The value of a numeric constant, identified with the same symbol in the Source Forms item of the instruction description.

`A, B, E, F,`
`D, W, X, Y, U`
`CC, DP, SP,`
`PC, MD, Q`
	The value of a specific register.

`;`	Marks the end of one operation and the beginning of another, performed by the same instruction.
`{or for (m)}`	Used in descriptions of instructions that can manipulate either a register or a memory location. Indicates that the given description is for register manipulation; the description for memory manipulation is identical, with every instance of the symbol `r` replaced by the symbol `(m)`.

3.4 CONDITION CODES FIELD

The Condition Codes field of each description shows the effect of the instruction on the 6309's condition code register (CC). This field describes the effect on each bit individually. For each bit of the condition code register, this field gives either a brief description or "N/C" (not changed).

3.5 ENCODING FIELD

hexadecimal values (6309 machine language). If you're using an assembler program, you'll probably never need to know the encoding of individual instructions. The information in this field is provided for debugging, hand-coding, and special-purpose applications.

For instructions that use the INHERENT addressing mode, the field lists a literal hexadecimal number (e.g. `$3A` for the ABX instruction). When an instruction allows several addressing modes, the Encoding field provides an encoding rule instead of a literal hexadecimal number.

Instructions that allow several addressing modes are encoded differently, depending on the addressing mode. Only the portions of the encoding that specify the instruction and its addressing mode are included in this field; the 6309 uses standard encoding for the instruction's operands given a specific addressing mode. The notation `(+ post-bytes)` in the Encoding field indicates that the rest of the instruction's encoding follows the standard 6309 format for the specified addressing mode, and that the operands immediately follow the given encoding in memory.

Refer to Section 2, Addressing Modes, for additional information.

(This page intentionally left blank)

(This page intentionally left blank)

ABX

Source Forms: ABX

Operation: X’ <- X + (unsigned)B

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C

E’ – N/C
F’ – N/C
I’ – N/C

Description: Adds the contents of the B register to the contexts of the X register, and stores the result in the X register. The contents of the B register are treated as an unsigned 8-bit value.

Addressing Modes: Inherent

Comments: Because this instruction treats B as an unsigned number, it can be used to change the value of the X register by +0 to +255. The similar LEAX B,X instruction changes the value of X by -128 to +127, and also updates the Z bit of the condition code register.

Examples:

```
LDX #TABLE  ;Set up X to point at table
ABX         ;Advance X by the value in B
LDA 0,X     ;Get a value from table
```

Encoding: $3A

Source Forms: ADCr p

Operation: r' <- r + (m) + CC.C

Condition Codes:

H' – 1 if half-carry generated
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Adds the contents of the memory byte, plus the carry bit of the condition code register, to the contents of the A or B register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for multi-precision addition, since it allows the carry from a previous byte or word addition to be added into a subsequent byte addition.

Examples:

```
ADDB #3     ;Add to B register - could create a carry
ADCA #0     ;Now the D register has been incremented by 3
```

Encoding:

```
$X9 (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

Source Forms: `ADCr p`

Operation: r’ <- r + (m) + CC.C

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if overflow, else 0
C’ – 1 if carry, else 0
E’ – N/C
F’ – N/C
I’ – N/C

Description: Adds the contents of the memory word, plus the carry bit of the condition code register, to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for multi-precision addition, since it allows the carry from a previous byte or word addition to be added into a subsequent word addition.

Examples:

```
ADDE 3,S    ;Add value on stack to E register
ADCD #0     ;Accumulate 24 bit result in A.B.E registers
```

Encoding:

```
$10X9 (+ post-bytes)  Register D

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

ADCR

Add Register Contents Plus Carry to Register

Source Forms: `ADCR r1,r2`

Operation: r2' <- r2 + r1 + CC.C

Condition Codes:

H' – N/C (unless r2 = CC)
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C (unless r2 = CC)
F' – N/C (unless r2 = CC)
I' – N/C (unless r2 = CC)

Description: Adds the contents of two registers, plus the carry bit, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be added. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
COMB         ;Force carry bit set
LDY #$4567   ;Now Y has $4567
LDX #$1234   ;Now X has $1234
ADCR X,Y     ;Now X still has $1234 and Y has $579C
```

Encoding: `$1031XY`

X = first register, Y = second register (gets result) from table above

ADD(8-

Add Memory Contents to Register

Source Forms: `ADDr p`

Operation: r' <- r + (m)

Condition Codes:

H' – 1 if half-carry generated
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0

E' – N/C
F' – N/C
I' – N/C

Description: Adds the contents of the memory byte to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for single-precision addition, or for the first byte of multi-precision addition, since it allows adding two quantities ignoring any carry left over from previous operations.

Examples:

```
LDA  ,X      ;Get a value from memory
ADDA <OFFSET ;Add predetermined offset from OFFSET
```

Encoding:

```
$XB (+ post-bytes)    Register A or B
$11XB (+ post bytes)  Register E* or F*

X = register and mode. For register A or E, 8=immediate,
9=direct, A=indexed, B=extended. For register B or F,
```

Source Forms:

```
ADDD p
ADDW p
```

Operation: r’ <- r + (m)

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if overflow, else 0
C’ – 1 if carry, else 0
E’ – N/C
F’ – N/C
I’ – N/C

Description: Adds the contents of the memory word to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for single-precision addition, or for the first word of multi-precision addition, since it allows adding two quantities ignoring any carry left over from previous operations.

Examples:

```
LDW  ,X      ;Get a value from memory
ADDW <OFFSET ;Add predetermined offset from OFFSET
```

Encoding:

```
$X3 (+ post-bytes)     Register D
$11XB (+ post-bytes)   Register W

X = addressing mode. For register D, C = immediate, D =
direct, E = indexed, F = extended. For register W*, 8 =
```

ADDR

Source Forms: `ADDR r1,r2`

Operation: r2' <- r2 + r1

Condition Codes:

H' – N/C (unless r2 = CC)
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C (unless r2 = CC)
F' – N/C (unless r2 = CC)
I' – N/C (unless r2 = CC)

Description: Adds the contents of two registers, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be added. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
ADDR X,Y    ;Now X still has $1234 and Y has $579B
```

Encoding: `$1030XY`

X = first register, Y = second register (gets result) from table above

AIM

AND Immediate Data to Memory

Source Forms: `AIM #n,pp`

Operation: (m)’ <- (m) & n

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always cleared
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Bit-wise ANDs the 8-bit contents of the addressed memory location with the 8-bit immediate data, and stores the result at the memory location.

Addressing Modes:
Direct
Indexed
Extended

Comments: This instruction executes an indivisible read-modify-write cycle.

Examples:

```
AIM #$7F,<FLAGS ;Clear the most-significant bit of FLAGS
```

Encoding:

```
$X2YY (+ post-bytes for addressing mode)
X = addressing mode (0 = direct, 6 = indexed, 7 = extended);
YY = immediate data
```

Source Forms: ANDr p

Operation: r' <- r & (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs the contents of the memory byte to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively clears bits of the specified register. For every clear bit in the operand, the instruction clears the corresponding register bit. The instruction does not change register bits corresponding to set bits in the operand.

Examples:

```
LDA  ,X      ;Get a value from memory
ANDA #$7F    ;Strip the most-significant bit from the value
```

Encoding:

```
$X4 (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

AND(16-

Bit-Wise AND Memory Contents to Register

Source Forms: ANDr p

Operation: r' <- r & (m)

Condition Codes:
H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs the contents of the memory word to the contents of the specified 16-bit register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively clears bits of the specified register. For every clear bit in the operand, the instruction clears the corresponding register bit. The instruction does not change register bits corresponding to set bits in the operand.

Examples:

```
LDD  ,X      ;Get a 16-bit value from memory
ANDA #$7FFF  ;Strip the most-significant bit from the value
```

Encoding:

```
$10X4 (+ post-bytes)  Register D*

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

ANDCC

Source Forms: `ANDCC #n`

Operation: CC' <- CC & n

Condition Codes:

H' ← H & n.5
N' ← N & n.3
Z' ← Z & n.2
V' ← V & n.1
C' ← C & n.0
E' ← E & n.7
F' ← F & n.6
I' ← I & n.4

Description: Bit-wise ANDs the 8-bit contents of the condition code register with the 8-bit immediate data, and stores the result in the condition code register.

Addressing Modes: Immediate (8-Bit)

Comments: The ANDCC instruction may be used to clear any desired bits into the condition code register. ANDCC is most often used to clear the F and I bits, to reenable hardware interrupts after executing a critical section of a program. It is also used to clear the carry bit CC.C.

Examples:

```
ANDCC #$AF  ;Enable IRQ and FIRQ interrupts
```

Encoding:

```
$1CNN

NN = immediate data to AND into condition code register.
```

ANDR

Source Forms: `ANDR r1,r2`

Operation: r2’ <- r2 & r1

Condition Codes:

H’ – N/C (unless r2 = CC)
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always 0
C’ – N/C
E’ – N/C (unless r2 = CC)
F’ – N/C (unless r2 = CC)
I’ – N/C (unless r2 = CC)

Description: Bit-wise ANDs the contents of two registers, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be ANDed. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB’s of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
ANDR X,Y    ;Now X still has $1234 and Y has $0024
```

Encoding: `$1034XY`

`X = first register, Y = second register (gets result) from table above`

Multiply the Signed Operand by Two

Source Forms:

```
ASLr
ASL pp
```

Operation: r’ <- r + r; r.0’ <- 0; CC.C’ <- r.7; CC.V’ <- r.7 ^ r.6; {or for (m)}

Condition Codes:

H’ – Undefined
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if sign changed, else 0
C’ – Set to MSB of operand
E’ – N/C
F’ – N/C
I’ – N/C

Description:

Doubles the value of the specified operand by shifting it left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The least-significant bit of the result is forced to 0.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes:

Inherent
Direct
Indexed
Extended

Comments:

When inherent addressing is used, this instruction shifts the contents of a register.

This instruction is identical to the LSL instruction, described elsewhere.

Examples:

```
LDB #$0F    ;Now B has $0F
ASLB        ;Now B has $1E, C, V clear
```

Encoding:

```
$X8                 Register A or B
X = register. For register A, X=4. For Register B, X=5.

$X8 (+ post-bytes) Memory
X = mode; 0=direct, 6=indexed, 7=extended.
```

ASL(16- Multiply the Signed Operand by Two

Source Forms: ASLr

Operation: r' <- r + r; r.0' <- 0; CC.C' <- r.15; CC.V' <- r.15 ^ r.14

Condition Codes:

H' – Undefined	E' – N/C
N' – 1 if result negative, else 0	F' – N/C
Z' – 1 if result zero, else 0	I' – N/C
V' – 1 if sign changed, else 0	
C' – Set to MSB of operand	

Description: Doubles the value of the specified operand by shifting it left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The least-significant bit of the result is forced to 0.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes: Inherent

Comments: This instruction is identical to the LSL instruction, described elsewhere.

Examples:

```
LDD #$00FF  ;Now D has $00FF
ASLD        ;Now D has $01FE, C, V clear (ASLD for 6309 0nly)
```

Encoding:

```
$1XX8       Register D

XX = register. For register D, XX=04. For register W, XX=05
```

Divide the Signed Operand by Two

Source Forms:

```
ASRr
ASR pp
```

Operation: r' <- r >> 1; r.7' <- r.7; CC.C' <- r.0; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Divides the specified operand by two, treating the operand as a 2's complement signed number. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The most-significant bit of the original operand is copied to the most-significant bit of the result to preserve the sign of the result.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: This instruction differs slightly from LSR. While LSR places a zero in the most-significant bit of the result, ASR preserves the sign of the original operand.

Examples:

```
LDA #$8F    ;Now A has $8F
ASRA        ;Now A has $C7, C is set
```

Encoding:

```
$X7                 Register A or B
X = register. For register A, X=4. For Register B, X=5.

$X7 (+ post-bytes) Memory
X = mode; 0=direct, 6=indexed, 7=extended.
```

Source Forms: ASRr

Operation: r' <- r >> 1; r.15' <- r.15; CC.C' <- r.0

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Divides the specified register by two, treating the operand as a 2's complement signed number. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The most-significant bit of the original operand is copied to the most-significant bit of the result to preserve the sign of the result.

Addressing Modes: Inherent

Comments: This instruction differs slightly from LSR. While LSR places a zero in the most-significant bit of the result, ASR preserves the sign of the original operand.

Examples:

```
LDD #$80FF   ;Now D has $80FF
ASRD         ;Now D has $C07F, C is set (ASRW for 6309 0nly)
```

Encoding:

```
$1XX7              Register D only

XX = register. For register D, XX=04. For register W, XX=05
```

BAND

AND Bit of Memory to Bit of Register

Source Forms: `BAND rr.n,qq.k`

Operation: rr.n’ <- (DP:qq).k & rr.n

Condition Codes:

H’ – N/C (unless rr = CC, n=5)
N’ – N/C (unless rr = CC, n=3)
Z’ – N/C (unless rr = CC, n=2)
V’ – N/C (unless rr = CC, n=1)
C’ – N/C (unless rr = CC, n=0)
E’ – N/C (unless rr = CC, n=7)
F’ – N/C (unless rr = CC, n=6)
I’ – N/C (unless rr = CC, n=4)

Description: ANDs bit (k) of 8-bit direct page memory location (qq) with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BAND A.7,FLAGS.2   ;Set bit 7 of register A to the AND of
                   ;itself and bit 2 of direct page FLAGS
BAND CC.I,FLAGS.3  ;Clear CC.I if bit 3 of FLAGS not set
```

Encoding:

```
$1130XXYY
XX = post-byte (RRkkknnn); YY = direct page address.  RR =
register number from above table; kkk = memory  bit #; nnn =
register bit #
```

Source Forms: `BCC id`

Operation: if (CC.C=0) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) bit is clear.

Behaves like BRN (branch never) if the condition code carry (CC.C) bit is set.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BCC stands for “branch if carry clear”. This instruction is identical to BHS. The opposite instruction is BCS. See also LBCC.

Examples:

```
LSRA        ;Get LSB of A to carry
BCC  EXIT   ;go to ‘EXIT’ if LSB of A was clear
.           ; we’d be here if LSB of A was set
```

Encoding: $24XX

XX = relative offset.

BCS

Set PC If CC Carry Set (Program Counter Relative)

Source Forms: `BCS id`

Operation: if (CC.C=1) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) bit is set.

Behaves like BRN (branch never) if the condition code carry (CC.C) bit is clear.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BCS stands for “branch if carry set”. This instruction is identical to BLO. The opposite instruction is BCC. See also LBCS.

Examples:

```
LSRA        ;Get LSB of A to carry
BCS  EXIT   ;go to ‘EXIT’ if LSB of A was set
.           ; we’d be here if LSB of A was clear
```

Encoding:

$25XX

XX = relative offset.

BEOR

Exclusive-OR Bit of Memory to Bit of Register

Source Forms: `BEOR rr.n,qq.k`

Operation: rr.n’ <- (DP:qq).k ^ rr.n

Condition Codes:

H’ – N/C (unless rr = CC, n=5)
N’ – N/C (unless rr = CC, n=3)
Z’ – N/C (unless rr = CC, n=2)
V’ – N/C (unless rr = CC, n=1)
C’ – N/C (unless rr = CC, n=0)
E’ – N/C (unless rr = CC, n=7)
F’ – N/C (unless rr = CC, n=6)
I’ – N/C (unless rr = CC, n=4)

Description: Exclusive-ORs bit (k) of 8-bit direct page memory location (qq) with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A        10 B        11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BEOR A.7,FLAGS.2   ;Set bit 7 of register A to the XOR of
                   ;itself and bit 2 of FLAGS
ORCC #1            ;Force carry set
BEOR CC.C,FLAGS.3  ;Flip carry bit if bit 3 of FLAGS set
```

Encoding:

```
$1134XXYY
XX = post-byte (RRkkknnn); Y = direct page address.  RR =
register number from above table; kkk = memory  bit #; nnn =
register bit #
```

BEQ

Set PC If CC Equal (Program Counter Relative)

Source Forms: `BEQ id`

Operation: if (CC.Z=1) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the most recent arithmetic operation left the condition code zero (CC.Z) bit set. This generally indicates that the result of the last signed or unsigned arithmetic operation was zero.

Behaves like BRN (branch never) if the most recent arithmetic operation left the condition code zero (CC.C) bit clear. This generally indicates that the result of the last signed or unsigned arithmetic operation was non-zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BEQ stands for "branch if equal". The opposite instruction is BNE. See also LBEQ.

Examples:

```
LDA  #$50
SUBA #$50
BEQ  EXIT  ;Now A=$00; $50 is "equal" to $50; go to 'EXIT'
.          ; we'd be here if 2nd instruction was SUBA #$43
```

Encoding:

```
$27XX

XX = relative offset.
```

BGE — Set PC If CC Greater or Equal (Program Counter Relative)

Source Forms: `BGE id`

Operation: if (CC.N = CC.V) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) and overflow (CC.V) bits match. This generally indicates that the result of the last signed arithmetic operation was "greater than or equal to" zero.

Behaves like BRN (branch never) if the condition code negative (CC.N) and overflow (CC.V) bits differ. This generally indicates that the result of the last signed arithmetic operation was "less than" zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BGE stands for "branch if greater or equal". The opposite instruction is BLT. See also LBGE.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
BGE  MORE   ;go to 'MORE' - result is greater.
.           ; continue here.
```

Encoding: $2CXX

XX = relative offset.

BGT

Set PC If CC Greater (Program Counter Relative)

Source Forms: `BGT id`

Operation: if (CC.N=CC.V and CC.Z=0) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) and overflow (CC.V) bits match, and the zero bit (CC.Z) is clear. This generally indicates that the result of the last signed arithmetic operation was “greater than” zero.

Behaves like BRN (branch never) if the condition code negative (CC.N) and overflow (CC.V) bits differ, or if the zero (CC.Z) bit is set. This generally indicates that the result of the last signed arithmetic operation was “less than or equal to” zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BGT stands for “branch if greater than”. The opposite instruction is BLE. See also LBGT.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
BGT  MORE   ;go to ‘MORE’ - result is greater.
.           ; continue here.
```

Encoding: $2EXX

XX = relative offset.

BHI

Set PC If CC Higher (Program Counter Relative)

Source Forms: `BHI id`

Operation: if (CC.C=0 and CC.Z=0) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) and zero (CC.Z) bits are clear. This generally indicates that the result of the last unsigned arithmetic operation was "higher than" zero.

Behaves like BRN (branch never) if the condition code carry (CC.C) or zero (CC.Z) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was "lower than or same as" zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BHI stands for "branch if higher". The opposite instruction is BLS. See also LBHI.

Examples:

```
LDA  #$80
SUBA #$50
BHI  EXIT  ;Now A=$30; $80 is "higher" than $50; go to 'EXIT'
.          ; we'd be here if 2nd instruction was SUBA #$90
```

Encoding:

$22XX

XX = relative offset.

BHS — Set PC If CC Higher or Same (Program Counter Relative)

Source Forms: `BHS id`

Operation: if (CC.C=0) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) bit is clear. This generally indicates that the result of the last unsigned arithmetic operation was "higher than or same as" zero.

Behaves like BRN (branch never) if the condition code carry (CC.C) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was "lower than" zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BHS stands for "branch if higher or same". This instruction is identical to BCC. The opposite instruction is BLO. See also LBHS.

Examples:

```
LDA  #$50
SUBA #$50
BHS  EXIT  ;Now A=$00; $50 is "same" as $50; go to 'EXIT'
.          ; we'd be here if 2nd instruction was SUBA #$90
```

Encoding: $24XX

XX = relative offset.

BIAND

AND Inverse Bit of Memory to Bit of Register

Source Forms: `BIAND rr.n,qq.k`

Operation: rr.n' <- ~[(DP:qq).k] & rr.n

Condition Codes:

H' – N/C (unless rr = CC, n=5)
N' – N/C (unless rr = CC, n=3)
Z' – N/C (unless rr = CC, n=2)
V' – N/C (unless rr = CC, n=1)
C' – N/C (unless rr = CC, n=0)
E' – N/C (unless rr = CC, n=7)
F' – N/C (unless rr = CC, n=6)
I' – N/C (unless rr = CC, n=4)

Description: ANDs the inverse of bit (k) of 8-bit direct page memory location (qq), with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BIAND A.7,FLAGS.2   ;Set bit 7 of register A to the AND of
                    ;itself and the inverse of bit 2 of FLAGS
ORCC #1             ;Force carry set
BIAND CC.C,FLAGS.3  ;Clear carry bit if bit 3 of FLAGS set
```

Encoding:

$1131XXYY
XX = post-byte (RRkkknnn); YY = direct page address. RR = register number from above table; kkk = memory bit #; nnn = register bit #

Source Forms: `BIEOR rr.n,qq.k`

Operation: rr.n’ <- ~[(DP:qq).k] ^ rr.n; if rr=CC, n=C bit

Condition Codes:

H’ – N/C (unless rr = CC, n=5)
N’ – N/C (unless rr = CC, n=3)
Z’ – N/C (unless rr = CC, n=2)
V’ – N/C (unless rr = CC, n=1)
C’ – N/C (unless rr = CC, n=0)
E’ – N/C (unless rr = CC, n=7)
F’ – N/C (unless rr = CC, n=6)
I’ – N/C (unless rr = CC, n=4)

Description: Exclusive-ORs the inverse of bit (k) of 8-bit direct page memory location (qq), with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

00 CC (CCR) 01 A 10 B 11 ---

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BIEOR A.7,FLAGS.2   ;Set bit 7 of register A to the XOR of
                    ;itself and the inverse of bit 2 of FLAGS
ANDCC #$AF          ;Force carry clear
BIEOR CC.C,FLAGS.3  ;Set carry bit if bit 3 of FLAGS set
```

Encoding:

$1135XXYY
XX = post-byte (RRkkknnn); YY = direct page address. RR = register number from above table; kkk = memory bit #; nnn = register bit #

BIOR

OR Inverse Bit of Memory to Bit of Register

Source Forms: `BIOR rr.n,qq.k`

Operation: rr.n’ <- ~[(DP:qq).k] | rr.n

Condition Codes:

H’ – N/C (unless rr = CC, n=5)
N’ – N/C (unless rr = CC, n=3)
Z’ – N/C (unless rr = CC, n=2)
V’ – N/C (unless rr = CC, n=1)
C’ – N/C (unless rr = CC, n=0)
E’ – N/C (unless rr = CC, n=7)
F’ – N/C (unless rr = CC, n=6)
I’ – N/C (unless rr = CC, n=4)

Description: ORs the inverse of bit (k) of 8-bit direct page memory location (qq), with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BIOR A.7,FLAGS.2   ;Set bit 7 of register A to the OR of
                   ;itself and the inverse of bit 2 of FLAGS
ANDCC #$AF         ;Force carry clear
BIOR CC.C,FLAGS.3  ;Set carry bit if bit 3 of FLAGS not set
```

Encoding:

$1133XXYY
XX = post-byte (RRkkknnn); YY = direct page address. RR = register number from above table; kkk = memory bit #; nnn = register bit #

Bit-Wise AND Memory Contents with Register

Source Forms: `BITr p`

Operation: r & (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs the contents of the memory byte with the contents of the specified register. The condition code register is updated according to the result, and the result is discarded.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction is generally used to determine whether certain bits of a value are 1 or 0.

Examples:

```
LDA  ,X      ;Get a value from memory
BITA #$40    ;See if 2nd MSB of the value is set
BNE  IS_SET  ;Branch if the bit is set
```

Encoding:

```
$X5 (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

Source Forms: BITr p

Operation: r & (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs the contents of the memory word with the 16-bit contents of the specified register. The condition code register is updated according to the result, and the result is discarded.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction is generally used to determine whether certain bits of a value are 1 or 0.

This is a new instruction for the 6309 and 6309E only.

Examples:

```
LDD  ,X       ;Get a 16-bit value from memory
BITD #$0480   ;See if either of two specified bits is set
BNE  IS_SET   ;Branch if either bit is set
```

Encoding:

```
$10X5 (+ post-bytes)    Register D

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

BITMD

Bit Test MD Register With Immediate Data

Source Forms: `BITMD #n`

Operation: MD & n

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs MD register contents with the 8-bit immediate data, and sets the condition code register based on the result. The MD register is not modified, but bits 6 and 7 are cleared. The Table lists the immediate values used to test specific bits of the MD register.

```
01000000  Illegal instruction TRAP flag; 1 = TRAP occurred
10000000  Division by 0 TRAP flag; 1 = TRAP occurred
```

Addressing Modes: Immediate

Comments: The HD6309 initializes the MD register to $00 at reset, for full compatibility with the MC6809. The processor operates in Native Mode when bit 0 of the MD register is set by the LDMD instruction. This bit, and the FIRQ mode bit of the MD register, are write-only and cannot be tested with the BITMD instruction.

Examples:

```
DOTRAP BITMD #$40  ;Trap entry - see if Division by 0 TRAP
       BEQ NotBy0  ;No - must be illegal instruction TRAP
       LDMD #$00   ;Clear /0 bit, stay in emulation mode
       RTI         ;Return from TRAP
```

Encoding:

```
$113CXX

XX is the immediate data to be ANDed with the MD register
```

BLE

Set PC If CC Less or Equal (Program Counter Relative)

Source Forms: `BLE id`

Operation: if (CC.N<>CC.V or CC.Z=0) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) and overflow (CC.V) bits differ, or if the zero (CC.Z) bit is set. This generally indicates that the result of the last signed arithmetic operation was “less than or equal to” zero.

Behaves like BRN (branch never) if the condition code negative (CC.N) and overflow (CC.V) bits match, and the zero bit (CC.Z) is clear. This generally indicates that the result of the last signed arithmetic operation was “greater than” zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BLE stands for “branch if less than or equal”. The opposite instruction is BGT. See also LBLE.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
BLE  LESS   ;don’t go to ‘LESS’ - result is greater.
.           ; continue here.
```

Encoding: $2FXX

XX = relative offset.

BLO

Set PC If CC Lower (Program Counter Relative)

Source Forms: `BLO id`

Operation: if (CC.C=1) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' –N/C
N' –N/C
Z' –N/C
V' –N/C
C' –N/C
E' –N/C
F' –N/C
I' –N/C

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than” zero.

Behaves like BRN (branch never) if the most recent arithmetic operation left the condition code carry (CC.C) bit clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than or same as” zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BLO stands for “branch if lower”. This instruction is identical to BCS. The opposite instruction is BHS. See also LBLO.

Examples:

```
LDA  #$50
SUBA #$50
BLO  EXIT  ;Now A=$00; $50 is “same” as $50; continue
.          ; we’d ‘EXIT’ if 2nd instruction was SUBA #$90
```

Encoding:

```
$24XX

XX = relative offset.
```

BLS

Set PC If CC Lower or Same (Program Counter Relative)

Source Forms: `BLS id`

Operation: if (CC.C=1 or CC.Z=1) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like BRA (branch always) if the condition code carry (CC.C) or zero (CC.Z) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than or same as” zero.

Behaves like BRN (branch never) if the condition code carry (CC.C) and zero (CC.Z) bits are clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than” zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BLS stands for “branch if lower or same”. The opposite instruction is BHI. See also LBLS.

Examples:

```
LDA  #$80
SUBA #$50
BLS  EXIT  ;Now A=$30; $80 is “higher” than $50; continue.
.          ; we’d ‘EXIT’ if 2nd instruction was SUBA #$90
```

Encoding:

`$23XX`

`XX = relative offset.`

BLT

Set PC If CC Less (Program Counter Relative)

Source Forms: `BLT id`

Operation: if (CC.N <> CC.V) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C	E' – N/C
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) and overflow (CC.V) bits differ. This generally indicates that the result of the last signed arithmetic operation was "less than" zero.

Behaves like BRN (branch never) if the condition code negative (CC.N) and overflow (CC.V) bits match. This generally indicates that the result of the last signed arithmetic operation was "greater than or equal to" zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BLT stands for "branch if less than". The opposite instruction is BGE. See also LBLT.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
BLT  LESS   ;don't go to 'LESS' - result is greater.
.           ; continue here.
```

Encoding: `$2DXX`

`XX = relative offset.`

BMI

Set PC If CC Negative Set (Program Counter Relative)

Source Forms: `BMI id`

Operation: if (CC.N=1) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) bit is set. This generally indicates that the result of the last signed arithmetic operation was negative.

Behaves like BRN (branch never) if the condition code negative (CC.N) bit is clear. This generally indicates that the result of the last signed arithmetic operation was positive (or zero).

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BMI stands for “branch if minus”. The opposite instruction is BPL. See also LBMI.

Examples:

```
LDA  #-128  ;Value of -128
ADDA #200   ;Result is +72
BMI  NEGTIV ;don’t go to ‘NEGTIV’ - result is positive.
.           ; continue here.
```

Encoding: $2BXX

XX = relative offset.

BNE

Set PC If CC Not Equal (Program Counter Relative)

Source Forms: `BNE id`

Operation: if (CC.Z=0) then PC’ <- PC+id+2; else PC’ <- PC+2

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like BRA (branch always) if the most recent arithmetic operation left the condition code zero (CC.Z) bit clear. This generally indicates that the result of the last signed or unsigned arithmetic operation was non-zero.

Behaves like BRN (branch never) if the most recent arithmetic operation left the condition code zero (CC.C) bit set. This generally indicates that the result of the last signed or unsigned arithmetic operation was zero.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BNE stands for “branch if not equal”. The opposite instruction is BEQ. See also LBNE.

Examples:

```
LDA  #$50
SUBA #$50
BNE  EXIT  ;Now A=$00; $50 is “equal” to $50; continue
.          ; we’d ‘EXIT’ if 2nd instruction was SUBA #$43
```

Encoding: $26XX

XX = relative offset.

BOR

OR Bit of Memory to Bit of Register

Source Forms: `BOR rr.n,qq.k`

Operation: rr.n' <- (DP:qq).k | rr.n

Condition Codes:

H' – N/C (unless rr = CC, n=5)
N' – N/C (unless rr = CC, n=3)
Z' – N/C (unless rr = CC, n=2)
V' – N/C (unless rr = CC, n=1)
C' – N/C (unless rr = CC, n=0)
E' – N/C (unless rr = CC, n=7)
F' – N/C (unless rr = CC, n=6)
I' – N/C (unless rr = CC, n=4)

Description: ORs bit (k) of 8-bit direct page memory location (qq) with bit (n) of register (rr), storing the result in bit (n) of the register. The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function
(immed. register number,
immed. bit numbers, direct
operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
BOR A.7,FLAGS.2   ;Set bit 7 of register A to the OR of
                  ;itself and bit 2 of FLAGS
ANDCC #$AF        ;Force carry clear
BOR CC.C,FLAGS.3  ;Set carry bit if bit 3 of FLAGS set
```

Encoding:

```
$1132XXYY
XX = post-byte (RRkkknnn); YY = direct page address.  RR =
register number from above table; kkk = memory  bit #; nnn =
register bit #
```

BPL

Set PC If CC Negative Clear (Program Counter Relative)

Source Forms: `BPL id`

Operation: if (CC.N=0) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code negative (CC.N) bit is clear. This generally indicates that the result of the last signed arithmetic operation was positive (or zero).

Behaves like BRN (branch never) if the condition code negative (CC.N) bit is set. This generally indicates that the result of the last signed arithmetic operation was negative.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BPL stands for "branch if plus". The opposite instruction is BMI. See also LBPL.

Examples:

```
LDA  #-128  ;Value of -128
ADDA #200   ;Result is +72
BPL  POSTIV ;go to 'POSTIV' - result is positive.
.          ; continue here if 2nd instruction was ADDA #88
```

Encoding: $2AXX

XX = relative offset.

BRA

Set Next Execution Address (Program Counter Relative)

Source Forms: `BRA id`

Operation: PC' <- PC+id+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Calculates a new value for the program counter, relative to its current value. This causes execution to continue starting from the instruction at the new program counter value.

The 8-bit signed imediate data (id) plus 2 is added to the original value of the program counter. This allows the BRA instruction to transfer control to an address anywhere within +127 to -128 bytes of the instruction immediately following the BRA.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. Use the JMP instruction for equivalent position-dependent code.

The ‘BRA id’ instruction is equivalent to ‘JMP id,PC’, but it occupies less memory. BRA stands for “branch always”.

Examples:

```
BRA  EXIT  ;Continue execution starting at label 'EXIT'
```

Encoding:

```
$20XX

XX = relative offset.
```

BRN

Skip One Byte

Source Forms: `BRN id`

Operation: PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction like NOP, has no effect on the processor, memory, or peripherals.

The BRN instruction occupies two memory locations, and always takes the same amount of time to execute. The second byte of the BRN instruction may be any data, including a one-byte instruction.

Addressing Modes: Inherent

Comments: The BRN instruction is most often used to “branch around” an immediately following single-byte instruction. It may also be used during program debugging, to temporarily disable a conditional branch by replacing it with BRN. BRN stands for “branch never”. Technically, BRN is the opposite of BRA (”branch always”).

Examples:

```
L0: CLRA       ;Clear Reg-A
    FCB $21    ;Op-code for BRN (skip to the BNE)
L1: ASLA       ;Shift Reg-A
    ADDA #$46  ;Add value $46 to either zero (L0) or 2*A (L1)
```

Encoding:

```
$21XX

XX = data to be skipped.
```

BVC

Set PC If CC Overflow Clear (Program Counter Relative)

Source Forms: `BVC id`

Operation: if (CC.V=0) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code overflow (CC.V) bit is clear. This generally indicates that overflow did not occur during the last signed arithmetic operation.

Behaves like BRN (branch never) if the condition code carry (CC.C) bit is set. This generally indicates that overflow occured during the last signed arithmetic operation.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BVC stands for "branch if overflow clear". The opposite instruction is BVS. See also LBVC.

Examples:

```
LDA  #$80  ;Value of -128
SUBA #4    ;Subtract 4 - creates an overflow!
BVC  OK    ;go to 'OK' if no overflow.
.          ; handle the overflow error here
```

Encoding: $28XX

XX = relative offset.

BVS

Set PC If CC Overflow Set (Program Counter Relative)

Source Forms: `BVS id`

Operation: if (CC.V=1) then PC' <- PC+id+2; else PC' <- PC+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like BRA (branch always) if the condition code overflow (CC.V) bit is set. This generally indicates that overflow occured during the last signed arithmetic operation.

Behaves like BRN (branch never) if the condition code carry (CC.C) bit is clear. This generally indicates that overflow did not occur during the last signed arithmetic operation.

Addressing Modes: Relative (8 bit)

Comments: This instruction produces position-independent code. BVS stands for "branch if overflow set". The opposite instruction is BVC. See also LBVS.

Examples:

```
LDA  #$80  ;Value of -128
SUBA #4    ;Subtract 4 - creates an overflow!
BVS  ERROR ;go to 'ERROR' if overflow.
.          ; continue here if no overflow
```

Encoding: $29XX

XX = relative offset.

Clear the Operand to Value 0

Source Forms:

```
CLRr
CLR pp
```

Operation: r' <- 0; {or for (m)}

Condition Codes:

H' – N/C
N' – Always cleared
Z' – Always set
V' – Always cleared
C' – Always cleared
E' – N/C
F' – N/C
I' – N/C

Description: Clears the specified operand to the constant zero, and updates the condition codes.

Addressing Modes:

Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction clears the contents of a register. Any one of the following registers may be specified by r: A, B, E*, F*.

Examples:

```
LDA #$0F     ;Now A = $0F
...          ;Assume processing here that makes A unknown
CLRA         ;Clear the A register; now A=$00
```

Encoding:

```
$XF                  (X = register; 4=A, 5=B)
$1XXF                (XX = register; 14=E, 15=F)
$XF (+ post-bytes) (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: CLRr

Operation: r' <- 0

Condition Codes:
H' – N/C
N' – Always cleared
Z' – Always set
V' – Always cleared
C' – Always cleared
E' – N/C
F' – N/C
I' – N/C

Description: Clears the specified operand to the constant zero, and updates the condition codes.

Addressing Modes: Inherent

Comments: This instruction clears the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Examples:

```
LDW #$00FF  ;Now W = $00FF
...         ;Assume processing here that makes W unknown
CLRW        ;Clear the W register; now W=$0000
```

Encoding: $1XXF (XX = register; 04=D, 05=W)

Source Forms: `CMPr p`

Operation: r - (m)

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if borrow, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts the contents of the memory byte from the contents of the specified register. The condition code register is updated according to the result, and the result is discarded.

Addressing Modes:

Immediate
Extended
Direct
Indexed

Comments: The comparison may be either signed or unsigned, depending on how the program interprets the condition code register after executing the CMP instruction.

Refer to the description of the BIT instruction for related information.

Examples:

```
LDA  ,X      ;Get a value from memory
CMPA #$40    ;Compare value to $40 (64 decimal)
BHI  BIGGER  ;Branch if the unsigned value exceeds $40
```

Encoding:

```
$X1 (+ post-bytes)    Register A or B
$11X1 (+ post-bytes)  Register E* or F*

X = register and mode. For register A or E, 8=immediate,
9=direct, A=indexed, B=extended. For register B or F,
```

Source Forms: `CMPr p`

Operation: r - (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if borrow, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts the contents of the memory word from the 16-bit contents of the specified register. The condition code register is updated according to the result, and the result is discarded.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: The comparison may be either signed or unsigned, depending on how the program interprets the condition code register after executing the CMP instruction.

Refer to the description of the BIT instruction for related information.

Examples:

```
LDU  ,X      ;Get a value from memory
CMPU #$1240  ;Compare value to constant $1240
BHI  BIGGER  ;Branch if the unsigned value exceeds $1240
```

Encoding:

```
$XC                     Register X
$10XY (+ post-bytes)  Register D, W*, or Y
$11XY (+ post-bytes)  Register U or S

X = mode; 8=immed., 9=direct, A=indxd, B=extd. Y = reg.; 1=W,
```

Source Forms: `CMPR r1,r2`

Operation: r2 - r1

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C (unless r2 = CC)
F' – N/C (unless r2 = CC)
I' – N/C (unless r2 = CC)

Description: Compares the contents of r1 to r2, and sets the condition code register according to the result. The post-byte of this instruction identifies the two registers to be subtracted. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

0000 D (A:B)	0100 SP	1000 A	1100 ---
0001 X	0101 PC	1001 B	1101 ---
0010 Y	0110 W (E:F)*	1010 CC (CCR)	1110 E *
0011 U (US)	0111 V *	1011 DP (DPR)	1111 F *

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
CMPR X,Y    ;Now X still has $1234 and Y still has $4567
BHI FOO     ;We'll go to FOO now, because Y > X
```

Encoding: `$1037XY`

`X = first register, Y = second register (gets result) from table above`

Source Forms:

```
COMr
COM pp
```

Operation: r' <- ~r; or (m)' <- ~(m)

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – Always set
E' – N/C
F' – N/C
I' – N/C

Description: Replaces the specified operand with its 1's complement. This is the same as inverting each bit of the operand.

Addressing Modes:

Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction complements the contents of a register. Any one of the following registers may be specified by r: A, B, E*, F*

Examples:

```
LDE #$0F     ;Now E has $0F
COME         ;Now E has $F0, C is set, V is cleared
             ;(6309 0nly)
```

Encoding:

```
$X1                 (X = register; 4=A, 5=B)
$1XX1               (XX = register; 14=E, 15=F)
$X1 (+ post-bytes)  (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: COMr

Operation: r' <- ~r

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – Always set
E' – N/C
F' – N/C
I' – N/C

Description: Replaces the specified operand with its 1's complement. This is the same as inverting each bit of the operand.

Addressing Modes: Inherent

Comments: This instruction complements the contents of a register. Any one of the following registers may be specified by r: D*, W*

Examples:

```
LDW #$00FF  ;Now W has $00FF
COMW        ;Now W has $FF00, C is set, V is cleared
            ;(6309 0nly)
```

Encoding:

```
$1XX1           (XX = register; 04=D, 05=W)
```

CWAI

Clear Selected CC Bits and Wait For Interrupt

Source Forms: `CWAI #n`

Operation: CC' <- CC & n; CC.E' <-1; Stack all registers; Wait for interrupt

Condition Codes:

H' – H & n.5
N' – N & n.3
Z' – Z & n.2
V' – V & n.1
C' – C & n.0
E' – Always 1
F' – F & n.6
I' – I & n.4

Description: When the processor executes a CWAI instruction, no other instructions will be executed until after the next enabled hardware interrupt. This synchronizes the processor to the hardware interrupt. This instruction may be executed regardless of the state of the I and F interrupt mask bits.

While waiting for the interrupt, the processor does not place its address and data busses in a high-impedance state. On the 6309 and 6309E, the processor enters a low-power "sleep mode".

Addressing Modes: Immediate

Comments: CWAI allows faster interrupt processing, since it pre-stacks the registers. Most applications of CWAI use it to modify interrupt mask bits CC.F and CC.I. Note that if CWAI clears CC.F, and FIRQ occurs, the processor will have saved ALL registers (including W in Native Mode) before entering the interrupt service routine.

Examples:

```
CWAI #$40  ;Enable FIRQ (assume IRQ previously disabled)
           ; and wait for interrupt
```

Encoding:

```
$3CNN

NN = 8-bit value to AND with condition code register
```

Source Forms: DAA

Operation: A' <- A+cf(A,CC.C)

Condition Codes:

H' – N/C
N' – Set if result < 0; else clear
Z' – Set if result 0; else clear
V' – Undefined
C' – See Comments, below
E' – N/C
F' – N/C
I' – N/C

Description: This instruction converts the result of an 8-bit binary addition into the result of a 2-digit BCD (binary coded decimal) addition. The conversion value cf(A) is calculated as two 4-bit nybbles from the current values of A and CC.C. The low nybble of cf(A) is:

```
6 if (CC.C=1) or (low nybble of A > $9)
0 otherwise
```

The high nybble of cf(A) is:

6 if (CC.C=1) or (high nybble of A > $9) or
(high nybble of A > $8 AND low nybble of A > $9)
0 otherwise

Addressing Modes: Inherent

Comments: The DAA instruction doesn't modify carry bit CC.C if the bit was set prior to executing DAA. If the bit was initially clear, DAA sets or clears carry based on the result of the addition that it performs (similarly to ADD).

Examples:

```
LDA  #$14  ;BCD for fourteen
ADDA #$37  ;BCD for thirty-seven; now A=$4B and CC.C=0
DAA        ;Fix up BCD number; cf(A) was $06; now A=$51
```

Encoding: $19

Source Forms:
```
DECr
DEC pp
```

Operation: r’ <- r - 1; {or for (m)}

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if overflow, else 0
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Decrements the specified signed operand and updates the condition codes.

Overflow occurs if the operand is $80, since decrementing this operand produces a negative number that cannot be expressed in 8 bits.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction decrements the contents of a register. Any one of the following registers may be specified by r: A, B, E*, F*.

Note that DEC does not effect the carry bit.

Examples:
```
LDE #$0  F  ;Now E has $0F
DECE        ;Now E has $0E (DECE for 6309 0nly)
```

Encoding:
```
$XA                  (X = register; 4=A, 5=B)
$1XXA                (XX = register; 14=E, 15=F)
$XA (+ post-bytes)   (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: `DECr`

Operation: r’ <- r - 1

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if overflow, else 0
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Decrements the specified signed operand and updates the condition codes.

Overflow occurs if the operand is $8000, since decrementing this operand produces a negative number that cannot be expressed in 16 bits.

Addressing Modes: Inherent

Comments: This instruction decrements the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Note that DEC does not effect the carry bit.

Examples:

```
LDW #$00FF  ;Now W has $00FF
DECW        ;Now W has $00FE (DECW for 6309 0nly)
```

Encoding:

```
$1XXA            (XX = register; 04=D, 05=W)
```

DIVD

Signed Integer Divide

Source Forms: `DIVD p`

Operation: B' <- D / (m); A' <- D % (m)

Condition Codes:

H' – N/C
N' – 1 if negative, else 0
Z' – 1 if quotient zero, else 0
V' – 1 if overflow, else 0
C' – 1 if quotient odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Divides the 16-bit signed contents of the D register (A:B) by the 8-bit signed contents of the memory location referenced by the addressing mode. Stores the 8-bit signed quotient in B, and the 8-bit unsigned remainder in A.

Addressing Modes:
Immediate
Direct
Indexed
Extended

Comments: In case of division by zero, this instruction generates a TRAP to the address at vector $FFF0. The BITMD instruction must be used to distinguish between a TRAP caused by zero division and a TRAP caused by an illegal instruction. In the case of overflow, the V bit is set and the contents of D are unchanged.

Examples:

```
LDD #$3006  ;Now D has $3006
DIVD #$60   ;Divide by $60
...         ;Now B has $80 (quotient), A has $06 (remainder)
```

Encoding:

```
$11XD (+ post-bytes for addressing mode)

X = addressing mode (8=immed., 9=direct, A=indexed,
B=extended)
```

DIVQ

Source Forms: `DIVQ p`

Operation: W' <- Q / (m); D' <- Q % (m)

Condition Codes:

H' – N/C
N' – 1 if negative, else 0
Z' – 1 if quotient zero, else 0
V' – 1 if overflow, else 0
C' – 1 if quotient odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Divides the 32-bit signed contents of the Q register (A:B:E:F) by the 16-bit signed contents of the memory location referenced by the addressing mode p. Stores the 16-bit signed quotient in W*, and the 16-bit unsigned remainder in D.

Addressing Modes:
Immediate
Direct
Indexed
Extended

Comments: In case of division by zero, this instruction generates a TRAP to the address at vector $FFF0. The BITMD instruction must be used to distinguish between a TRAP caused by zero division and a TRAP caused by an illegal instruction. In the case of overflow, the V bit is set and the contents of Q are unchanged.

Examples:

```
LDQ #$0456B56A  ;Now Q has $0456B56A
DIVQ #$1001     ;Divide by $1001
...             ;Now W has $4567, D has $0003
```

Encoding:

```
$11XE (+ post-bytes for addressing mode)

X = addressing mode (8=immed., 9=direct, A=indexed,
B=extended)
```

EIM

Source Forms: `EIM #n,pp`

Operation: (m)’ <- (m) ^ n

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always cleared
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Bit-wise exclusive-ORs the 8-bit contents of the addressed memory location with the 8-bit immediate data, and stores the result at the memory location.

Addressing Modes:
Direct
Indexed
Extended

Comments: This instruction executes an indivisible read-modify-write cycle.

Examples:

```
EIM #$80,<FLAGS ;Flip the most-significant bit of FLAGS
```

Encoding:

```
$X5YY (+ post-bytes for addressing mode)
X = addressing mode (0 = direct, 6 = indexed, 7 = extended);
YY = immediate data
```

Source Forms: `EORr p`

Operation: r' <- r ^ (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise Exclusive-ORs the contents of the memory byte to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively inverts bits of the specified register. For every set bit in the operand, the instruction inverts the corresponding register bit. The instruction does not change register bits corresponding to clear bits in the operand.

Examples:

```
LDA  ,X       ;Get a value from memory
EORA #$55     ;Invert every other bit of the value
```

Encoding:

```
$X8 (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

Source Forms: `EORr p`

Operation: r' <- r ^ (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise Exclusive-ORs the contents of the memory word to the contents of the specified 16-bit register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively inverts bits of the specified register. For every set bit in the operand, the instruction inverts the corresponding register bit. The instruction does not change register bits corresponding to clear bits in the operand.

Examples:

```
LDD  ,X      ;Get a 16-bit value from memory
EORD #$5555  ;Invert every other bit of the value
```

Encoding:

```
$10X8 (+ post-bytes)    Register D*

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

Source Forms: `EORR r1,r2`

Operation: r2' <- r2 ^ r1

Condition Codes:

H' – N/C (unless r2 = CC)
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C (unless r2 = CC)
F' – N/C (unless r2 = CC)
I' – N/C (unless r2 = CC)

Description: Bit-wise exclusive-ORs the contents of two registers, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be XORed. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

0000 D (A:B)	0100 SP	1000 A	1100 ---
0001 X	0101 PC	1001 B	1101 ---
0010 Y	0110 W (E:F)*	1010 CC (CCR)	1110 E *
0011 U (US)	0111 V *	1011 DP (DPR)	1111 F *

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
EORR X,Y    ;Now X still has $1234 and Y has $5753
```

Encoding: `$1036XY`

X = first register, Y = second register (gets result) from table above

EXG

Source Forms: EXG r1,r2

Operation: r2' <- r1; r1' <- r2

Condition Codes:

H' – N/C (unless r1 or r2 = CC)
N' – N/C (unless r1 or r2 = CC)
Z' – N/C (unless r1 or r2 = CC)
V' – N/C (unless r1 or r2 = CC)
C' – N/C (unless r1 or r2 = CC)
E' – N/C (unless r1 or r2 = CC)
F' – N/C (unless r1 or r2 = CC)
I' – N/C (unless r1 or r2 = CC)

Description: Exchanges the contents of two registers of the same size. The post-byte of this instruction identifies the two registers to be swapped. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r1 or r2 is the condition code register (CC), all condition code flags are set to the values of corresponding bits in the other register.

The order of the registers in the post-byte does not effect operation. The same register can be used in both positions if desired.

Examples:

```
LDD #$0000  ;Now D has $0000
LDX #$1234  ;Now X has $1234
EXG D,X     ;Now X has $0000 and D has $1234
```

Encoding: $1EXY

X = first register, Y = second register from table above

Source Forms:

```
INCr
INC pp
```

Operation: r' <- r + 1; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Increments the specified signed operand and updates the condition codes.

Overflow occurs if the operand is $7F, since incrementing this operand produces a signed positive number that cannot be expressed in 8 bits.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction increments the contents of a register. Any one of the following registers may be specified by r: A, B, E*, F*.

Note that INC does not effect the carry bit.

Examples:

```
LDE #$0F     ;Now E has $0F
INCE         ;Now E has $10 (INCE for 6309 0nly)
```

Encoding:

```
$XC                 (X = register; 4=A, 5=B)
$1XXC               (XX = register; 14=E, 15=F)
$XC (+ post-bytes) (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: `INCr`

Operation: r’ <- r + 1

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if overflow, else 0
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Increments the specified signed operand and updates the condition codes.

Overflow occurs if the operand is $7FFF, since incrementing this operand produces a signed positive number that cannot be expressed in 16 bits.

Addressing Modes: Inherent

Comments: This instruction increments the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Note that INC does not effect the carry bit.

Examples:

```
LDW #$00FF  ;Now W has $00FF
INCW        ;Now W has $0100 (INCW for 6309 0nly)
```

Encoding:

```
$1XXC              (XX = register; 04=D, 05=W)
```

JMP

Set Next Execution Address

Source Forms: `JMP ea`

Operation: PC' <- ea

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Calculates an effective address (ea), and stores it in the program counter. This causes execution to continue starting from the instruction at the effective address.

Addressing Modes:
Extended
Direct
Indexed

Comments: When used with extended addressing, this instruction produces position-dependent code. Use the LBRA instruction for equivalent position-independent code.

Examples:

```
JMP  >RESET  ;Continue execution starting at label 'RESET'
```

Encoding:

```
$XE (+ post-bytes)

X = addressing mode. 0=direct, 6=indexed, 7=extended.
```

JSR

Call A Subroutine

Source Forms: `JSR ea`

Operation: SP’ <- SP-2; (SP-2)’ <- PC+3; PC’ <- ea

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Pushes the absolute address of the next instruction onto the stack, then calculates an effective address (ea), and stores it in the program counter.

This causes execution to continue starting from the instruction at the effective address. Subsequent execution of an RTS instruction causes execution to resume at the instruction immediately following the JSR.

Addressing Modes:
Extended
Direct
Indexed

Comments: When used with extended addressing, this instruction produces position-dependent code. Use the LBSR instruction for equivalent position-independent code.

Examples:

```
JSR  >SETUP  ;Call subroutine starting at label ‘SETUP’
.            ; continue here after subroutine
.
```

Encoding:

```
$XD (+ post-bytes)

X = addressing mode. 9=direct, A=indexed, B=extended.
```

LBCC

Set PC If CC Carry Clear (Program Counter Relative)

Source Forms: `LBCC id`

Operation: if (CC.C=0) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the condition code carry (CC.C) bit is clear.

Behaves like LBRN if the condition code carry (CC.C) bit is set.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBCC stands for "long branch if carry clear". This instruction is identical to LBHS. The opposite instruction is LBCS. See also BCC.

Examples:

```
LSRA       ;Get LSB of A to carry
LBCC EXIT  ;go to 'EXIT' if LSB of A was clear
.          ; we'd be here if LSB of A was set
```

Encoding:

```
$1024XXXX

XXXX = relative offset.
```

Source Forms: `LBCS id`

Operation: if (CC.C=1) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like LBRA (branch always) if the condition code carry (CC.C) bit is set.

Behaves like LBRN (branch never) if the condition code carry (CC.C) bit is clear.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBCS stands for “long branch if carry set”. This instruction is identical to LBLO. The opposite instruction is LBCC. See also BCS.

Examples:

```
LSRA        ;Get LSB of A to carry
LBCS EXIT   ;go to ‘EXIT’ if LSB of A was set
.           ; we’d be here if LSB of A was clear
```

Encoding:

```
$1025XXXX

XXXX = relative offset.
```

LBEQ

Set PC If CC Equal (Program Counter Relative)

Source Forms: `LBEQ id`

Operation: if (CC.Z=1) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like LBRA if the most recent arithmetic operation left the condition code zero (CC.Z) bit set. This generally indicates that the result of the last signed or unsigned arithmetic operation was zero.

Behaves like LBRN if the most recent arithmetic operation left the condition code zero (CC.C) bit clear. This generally indicates that the result of the last signed or unsigned arithmetic operation was non-zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBEQ stands for “long branch if equal”. The opposite instruction is LBNE. See also BEQ.

Examples:

```
LDA  #$50
SUBA #$50
LBEQ EXIT  ;Now A=$00; $50 is “equal” to $50; go to ‘EXIT’
.          ; we’d be here if 2nd instruction was SUBA #$43
```

Encoding: $1027XXXX

XXXX = relative offset.

LBGE — Set PC If CC Greater or Equal (Program Counter Relative)

Source Forms: `LBGE id`

Operation: if (CC.N = CC.V) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like LBRA if the condition code negative (CC.N) and overflow (CC.V) bits match. This generally indicates that the result of the last signed arithmetic operation was “greater than or equal to” zero.

Behaves like LBRN if the condition code negative (CC.N) and overflow (CC.V) bits differ. This generally indicates that the result of the last signed arithmetic operation was “less than” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBGE stands for “long branch if greater or equal”. The opposite instruction is LBLT. See also BGE.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
LBGE MORE   ;go to ‘MORE’ - result is greater.
.           ; continue here.
```

Encoding: `$102CXXXX`

XXXX = relative offset.

LBGT

Source Forms: `LBGT id`

Operation: if (CC.N=CC.V and CC.Z=0) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like LBRA if the condition code negative (CC.N) and overflow (CC.V) bits match, and the zero bit (CC.Z) is clear. This generally indicates that the result of the last signed arithmetic operation was “greater than” zero.

Behaves like LBRN if the condition code negative (CC.N) and overflow (CC.V) bits differ, or if the zero (CC.Z) bit is set. This generally indicates that the result of the last signed arithmetic operation was “less than or equal to” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBGT stands for “long branch if greater than”. The opposite instruction is LBLE. See also BGT.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
LBGT MORE   ;go to ‘MORE’ - result is greater.
.           ; continue here.
```

Encoding: $102EXXXX

XXXX = relative offset.

LBHI

Set PC If CC Higher (Program Counter Relative)

Source Forms: `LBHI id`

Operation: if (CC.C=0 and CC.Z=0) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like LBRA if the condition code carry (CC.C) and zero (CC.Z) bits are clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than” zero.

Behaves like LBRN if the condition code carry (CC.C) or zero (CC.Z) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than or same as” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBHI stands for “long branch if higher”. The opposite instruction is LBLS. See also BHI.

Examples:

```
LDA  #$80
SUBA #$50
LBHI EXIT  ;Now A=$30; $80 is “higher” than $50; go to ‘EXIT’
.          ; we’d be here if 2nd instruction was SUBA #$90
```

Encoding:

$1022XXXX

XXXX = relative offset.

LBHS

Set PC If CC Higher or Same (Program Counter Relative)

Source Forms: `LBHS id`

Operation: if (CC.C=0) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the condition code carry (CC.C) bit is clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than or same as” zero.

Behaves like LBRN if the condition code carry (CC.C) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBHS stands for “long branch if higher or same”. This instruction is identical to LBCC. The opposite instruction is LBLO. See also BHS.

Examples:

```
LDA  #$50
SUBA #$50
LBHS EXIT  ;Now A=$00; $50 is “same” as $50; go to ‘EXIT’
.          ; we’d be here if 2nd instruction was SUBA #$90
```

Encoding: $1024XXXX

XXXX = relative offset.

LBLE

Set PC If CC Less or Equal (Program Counter Relative)

Source Forms: `LBLE id`

Operation: if (CC.N<>CC.V or CC.Z=0) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the condition code negative (CC.N) and overflow (CC.V) bits differ, or if the zero (CC.Z) bit is set. This generally indicates that the result of the last signed arithmetic operation was "less than or equal to" zero.

Behaves like LBRN if the condition code negative (CC.N) and overflow (CC.V) bits match, and the zero bit (CC.Z) is clear. This generally indicates that the result of the last signed arithmetic operation was "greater than" zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBLE stands for "long branch if less than or equal". The opposite instruction is LBGT. See also BLE.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
LBLE LESS   ;don't go to 'LESS' - result is greater.
.           ; continue here.
```

Encoding:

```
$102FXXXX

XXXX = relative offset.
```

LBLO

Set PC If CC Lower (Program Counter Relative)

Source Forms: `LBLO id`

Operation: if (CC.C=1) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C	E' – N/C
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: Behaves like LBRA if the condition code carry (CC.C) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than” zero.

Behaves like LBRN if the most recent arithmetic operation left the condition code carry (CC.C) bit clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than or same as” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBLO stands for “long branch if lower”. This instruction is identical to LBCS. The opposite instruction is LBHS. See also BLO.

Examples:

```
LDA  #$50
SUBA #$50
LBLO EXIT  ;Now A=$00; $50 is “same” as $50; continue
.          ; we’d ‘EXIT’ if 2nd instruction was SUBA #$90
```

Encoding: `$1024XXXX`

`XXXX = relative offset.`

LBLS

Set PC If CC Lower or Same (Program Counter Relative)

Source Forms: `LBLS id`

Operation: if (CC.C=1 or CC.Z=1) then PC’ <- PC+id+4; else PC’ <- PC+2

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like LBRA if the condition code carry (CC.C) or zero (CC.Z) bit is set. This generally indicates that the result of the last unsigned arithmetic operation was “lower than or same as” zero.

Behaves like LBRN if the condition code carry (CC.C) and zero (CC.Z) bits are clear. This generally indicates that the result of the last unsigned arithmetic operation was “higher than” zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBLS stands for “long branch if lower or same”. The opposite instruction is LBHI. See also BLS.

Examples:

```
LDA  #$80
SUBA #$50
LBLS EXIT  ;Now A=$30; $80 is “higher” than $50; continue.
.          ; we’d ‘EXIT’ if 2nd instruction was SUBA #$90
```

Encoding: `$1023XX`

`XX = relative offset.`

LBLT

Set PC If CC Less (Program Counter Relative)

Source Forms: `LBLT id`

Operation: if (CC.N <> CC.V) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the condition code negative (CC.N) and overflow (CC.V) bits differ. This generally indicates that the result of the last signed arithmetic operation was "less than" zero.

Behaves like LBRN if the condition code negative (CC.N) and overflow (CC.V) bits match. This generally indicates that the result of the last signed arithmetic operation was "greater than or equal to" zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBLT stands for "long branch if less than". The opposite instruction is LBGE. See also BLT.

Examples:

```
LDA  #-10   ;Value of -10
CMPA #-20   ;Result: -10 is greater than -20
LBLT LESS   ;don't go to 'LESS' - result is greater.
.           ; continue here.
```

Encoding: $102DXXXX

XXXX = relative offset.

LBMI

Set PC If CC Negative Set (Program Counter Relative)

Source Forms: `LBMI id`

Operation: if (CC.N=1) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C	E' – N/C
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: Behaves like LBRA if the condition code negative (CC.N) bit is set. This generally indicates that the result of the last signed arithmetic operation was negative.

Behaves like LBRN if the condition code negative (CC.N) bit is clear. This generally indicates that the result of the last signed arithmetic operation was positive (or zero).

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBMI stands for "long branch if minus". The opposite instruction is LBPL. See also BMI.

Examples:

```
LDA  #-128  ;Value of -128
ADDA #200   ;Result is +72
LBMI NEGTIV ;don't go to 'NEGTIV' - result is positive.
.           ; continue here.
```

Encoding: $102BXXXX

XXXX = relative offset.

LBNE

Set PC If CC Not Equal (Program Counter Relative)

Source Forms: `BNE id`

Operation: if (CC.Z=0) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the most recent arithmetic operation left the condition code zero (CC.Z) bit clear. This generally indicates that the result of the last signed or unsigned arithmetic operation was non-zero.

Behaves like LBRN if the most recent arithmetic operation left the condition code zero (CC.C) bit set. This generally indicates that the result of the last signed or unsigned arithmetic operation was zero.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBNE stands for "long branch if not equal". The opposite instruction is LBEQ. See also BNE.

Examples:

```
LDA  #$50
SUBA #$50
LBNE EXIT  ;Now A=$00; $50 is "equal" to $50; continue
.          ; we'd 'EXIT' if 2nd instruction was SUBA #$43
```

Encoding:

$1026XXXX

XXXX = relative offset.

LBPL

Source Forms: `LBPL id`

Operation: if (CC.N=0) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C	E’ – N/C
N’ – N/C	F’ – N/C
Z’ – N/C	I’ – N/C
V’ – N/C	
C’ – N/C	

Description: Behaves like LBRA if the condition code negative (CC.N) bit is clear. This generally indicates that the result of the last signed arithmetic operation was positive (or zero).

Behaves like LBRN if the condition code negative (CC.N) bit is set. This generally indicates that the result of the last signed arithmetic operation was negative.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBPL stands for “long branch if plus”. The opposite instruction is LBMI. See also BPL.

Examples:

```
LDA  #-128  ;Value of -128
ADDA #200   ;Result is +72
LBPL POSTIV ;go to ‘POSTIV’ - result is positive.
.          ; continue here if 2nd instruction was ADDA #88
```

Encoding: $102AXXXX

XXXX = relative offset.

Source Forms: `LBRA id`

Operation: PC' <- PC+id+3

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Calculates a new value for the program counter, relative to its current value. This causes execution to continue starting from the instruction at the new program counter value.

The 16-bit signed imediate data (id) plus 3 is added to the original value of the program counter. This allows the LBRA instruction to transfer control to an address anywhere within +32767 to -32768 bytes (modulo 65536) of the instruction immediately following the LBRA.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. Use the JMP instruction for equivalent position-dependent code.

The ‘LBRA id’ instruction is equivalent to ‘JMP id,PC’, but it occupies less memory. LBRA stands for “long branch always”

Examples:

```
LBRA  RESET  ;Continue execution starting at label ‘RESET’
```

Encoding:

$16XXXX

XXXX = relative offset.

LBRN

Skip Two Bytes

Source Forms: `LBRN id`

Operation: PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction like NOP, has no effect on the processor, memory, or peripherals.

The LBRN instruction occupies four memory locations, and always takes the same amount of time to execute. The 3rd and 4th bytes of the LBRN instruction may be any data, including a two-byte instruction.

Addressing Modes: Inherent

Comments: The LBRN instruction is most often used during program debugging, to temporarily disable a conditional branch by replacing it with LBRN. LBRN stands for "long branch never". Technically, LBRN is the opposite of LBRA ("long branch always").

Examples:

```
LBRN NEVER    ;Don't transfer control from here to 'NEVER'
```

Encoding:

```
$1021XXXX

XXXX = data to be skipped.
```

LBSR

Call A Subroutine (Program Counter Relative)

Source Forms: `LBSR id`

Operation: SP' <- SP-2; (SP-2)' <- PC+3; PC' <- PC+id+3

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Pushes the absolute address of the next instruction onto the stack, then calculates a new value for the program counter, relative to its current value. The 16-bit signed imediate data (id) plus 3 is added to the original value of the program counter.

This causes execution to continue starting from the instruction at the effective address. Subsequent execution of an RTS instruction causes execution to resume at the instruction immediately following the LBSR.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. Use the JSR instruction for equivalent position-dependent code.

The 'LBSR id' instruction is equivalent to 'JSR id,PC', but it occupies less memory. LBSR stands for "long branch to subroutine"

Examples:

```
LBSR  SETUP   ;Call subroputine starting at label 'SETUP'
.             ; continue execution here after subroutine
.
```

Encoding: $17XXXX

XXXX = relative offset.

Source Forms: `LBVC id`

Operation: if (CC.V=0) then PC' <- PC+id+4; else PC' <- PC+4

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Behaves like LBRA if the condition code overflow (CC.V) bit is clear. This generally indicates that overflow did not occur during the last signed arithmetic operation.

Behaves like LBRN if the condition code carry (CC.C) bit is set. This generally indicates that overflow occured during the last signed arithmetic operation.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBVC stands for "long branch if overflow clear". The opposite instruction is LBVS. See also BVC.

Examples:

```
LDA  #$80  ;Value of -128
SUBA #4    ;Subtract 4 - creates an overflow!
LBVC OK    ;go to 'OK' if no overflow.
.          ; handle the overflow error here
```

Encoding: $1028XXXX

XXXX = relative offset.

LBVS

Set PC If CC Overflow Set (Program Counter Relative)

Source Forms: `LBVS id`

Operation: if (CC.V=1) then PC’ <- PC+id+4; else PC’ <- PC+4

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Behaves like LBRA if the condition code overflow (CC.V) bit is set. This generally indicates that overflow occured during the last signed arithmetic operation.

Behaves like LBRN if the condition code carry (CC.C) bit is clear. This generally indicates that overflow did not occur during the last signed arithmetic operation.

Addressing Modes: Relative (16 bit)

Comments: This instruction produces position-independent code. LBVS stands for “long branch if overflow set”. The opposite instruction is LBVC. See also BVS.

Examples:

```
LDA  #$80  ;Value of -128
SUBA #4    ;Subtract 4 - creates an overflow!
LBVS ERROR ;go to ‘ERROR’ if overflow.
.          ; continue here if no overflow
```

Encoding: $1029XXXX

XXXX = relative offset.

Source Forms: `LDr p`

Operation: r' <- (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies the contents of the memory byte to the specified register. The condition code register is updated according to the result.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: The value loaded may be either signed or unsigned, depending on how the program interprets the condition code register after executing the LD instruction.

Examples:

```
LDA  ,X      ;Load register A with 8-bit value from memory
LDB  #$40    ;Load register B with the constant $40
```

Encoding:

```
$X6 (+ post-bytes)    Register A or B
$11X6 (+ post-bytes)  Register E* or F*

X = register and mode. For register A or E, 8=immediate,
9=direct, A=indexed, B=extended. For register B or F,
```

Source Forms: LDr p

Operation: r' <- (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies the contents of the memory word to the specified 16-bit register. The condition code register is updated according to the result.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: The value loaded may be either signed or unsigned, depending on how the program interprets the condition code register after executing the LD instruction.

Examples:

```
LDD  ,X      ;Load register D with 16-bit value from memory
LDY  #$0400  ;Load register Y with the constant $0400
```

Encoding:

```
$XY                     Register D, U or X
$10XY (+ post-bytes)  Register S, W*, or Y

X = mode; for W*, X, or Y, 8=immed., 9=direct, A=indxd,
B=extd.; for D, S or U, C=immed., D=direct, E=indxd, F=extd.
```

LD(32-

Source Forms: `LDr p`

Operation: r' <- (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies the contents of the long memory word to the specified 16-bit register. The condition code register is updated according to the result.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: The value loaded may be either signed or unsigned, depending on how the program interprets the condition code register after executing the LD instruction.

Examples:

```
LDQ  ,X      ;Load register Q with 32-bit value from memory
```

Encoding:

```
$CDNNNNNNNN           Register Q, immediate (N = immed. data)
$10XC (+ post-bytes)  Register Q, other modes

X = mode; for register Q, D=direct, E=indexed, F=extended.
```

LDBT

Load Bit of Memory to Bit of Register

Source Forms: `LDBT rr.n,qq.k`

Operation: rr.n' <- (DP:qq).k

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C (unless rr = CC)
E' – N/C
F' – N/C
I' – N/C

Description: Copies bit (k) of 8-bit direct page memory location (qq) to bit (n) of register (rr). The memory location and other bits of the register are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
LDBT A,7,FLAGS,2 ;Set bit 7 of register A to bit 2 of FLAGS
LDBT CC,FLAGS,3  ;Set carry to value of bit 3 of FLAGS
```

Encoding:

```
$1136XXYY
XX = post-byte (RRkkknnn); YY = direct page address.  RR =
register number from above table; kkk = memory  bit #; nnn =
register bit #
```

LDMD

Source Forms: `LDMD #n`

Operation: MD' <- n

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction controls the operating mode of the microprocessor by writing data into the MD register.

Bit 0 (the least significant bit) of the MD register controls execution mode: a value of 0 selects emulation of the MC6809, while a value of 1 selects "native" HD6309 mode. Bit 1 of the MD register controls the operation of the FIRQ* interrupt. A value of 0 emulates MC6809 operation, while a value of 1 causes the processor to respond to FIRQ* as it would to IRQ*.

Addressing Modes: Immediate

Comments: The HD6309 initializes the MD register to $00 at reset, for full compatibility with the MC6809. All special 6309 instructions may be used in this emulation mode. The "native" mode of the HD6309 shortens many instructions by 1 cycle and stacks the W register between DP and B during interrupts.

Examples:

```
LDMD #$01  ;Begin processing in Native Mode
```

Encoding:

```
$113DXX

XX is the immediate data to be stored in the MD register
```

LEA

Source Forms: `LEAr ea`

Operation: r' <- ea

Condition Codes:

H' – N/C
N' – N/C
Z' – See Comments below.
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Calculates an effective address (ea), and stores it in the specified 16-bit register.

Addressing Modes: Indexed

Comments: The LEAX and LEAY variants of this instruction effects the zero bit of the condition code register (CC.Z).

The LEAU and LEAS variants do not effect any condition code register bits.

Examples:

```
LEAX  TABLE,PCR  ;Load X with PC-relative address of TABLE
LEAY  -1,Y       ;Decrement the Y register
LEAU  ,U         ;Test the U register for zero
LEAX  0,Y        ;Faster than TFR X,Y!
```

Encoding:

```
$3X (+ post-bytes)   ;Register X, Y, S, or U

X = register. 0=X, 1=Y, 2=S, 3=U.
```

Source Forms:

```
LSLr
LSL pp
```

Operation: r’ <- r << 1; r.0’ <- 0; CC.C’ <- r.7; CC.V’ <- r.7 ^ r.6; {or for (m)}

Condition Codes:

H’ – Undefined
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if sign changed, else 0
C’ – Set to MSB of operand
E’ – N/C
F’ – N/C
I’ – N/C

Description: Shifs the specified operand left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The least-significant bit of the result is forced to 0.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes: Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction shifts the contents of a register.

This instruction is identical to the ASL instruction, described elsewhere.

Examples:

```
LDB #$0F    ;Now B has $0F
LSLB        ;Now B has $1E, C, V clear
```

Encoding:

```
$X8                 Register A or B
X = register. For register A, X=4. For Register B, X=5.

$X8 (+ post-bytes) Memory
X = mode; 0=direct, 6=indexed, 7=extended.
```

Logically Shift the Operand Left One Position

Source Forms: LSLr

Operation: r' <- r << 1; r.0' <- 0; CC.C' <- r.15; CC.V' <- r.15 ^ r.14

Condition Codes:

H' – Undefined	E' – N/C
N' – 1 if result negative, else 0	F' – N/C
Z' – 1 if result zero, else 0	I' – N/C
V' – 1 if sign changed, else 0	
C' – Set to MSB of operand	

Description: Shifts the operand left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The least-significant bit of the result is forced to 0.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes: Inherent

Comments: This instruction is identical to the ASL instruction, described elsewhere.

Examples:

```
LDD #$00FF  ;Now D has $00FF
LSLD        ;Now D has $01FE, C, V clear (LSLD for 6309 0nly)
```

Encoding:

```
$1XX8       Register D

XX = register. For register D, XX=04.
```

Source Forms:

```
LSRr
LSR pp
```

Operation: r' <- r >> 1; r.7' <- 0; CC.C' <- r.0; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Shifts the specified operand right one position. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The most-significant bit of the result is forced to 0.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: This instruction differs slightly from the ASR instruction. While the ASR instruction preserves the sign of the original operand, the LSR instruction always clears the most-signification bit of the result.

Examples:

```
LDB #$0F    ;Now B has $0F
LSRB        ;Now B has $07, C is set
```

Encoding:

```
$X4                 Register A or B
X = register. For register A, X=4. For Register B, X=5.

$X4 (+ post-bytes) Memory
X = mode; 0=direct, 6=indexed, 7=extended.
```

Source Forms: LSRr

Operation: r' <- r >> 1; r.15' <- 0; CC.C' <- r.0

Condition Codes:

H' – N/C
N' – Always cleared
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0

E' – N/C
F' – N/C
I' – N/C

Description: Shifts the specified operand right one position. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The most-significant bit of the result is forced to 0.

Addressing Modes: Inherent

Comments: This instruction differs slightly from the ASR instruction. While the ASR instruction preserves the sign of the original operand, the LSR instruction always clears the most-signification bit of the result.

Examples:

```
LDW #$00FF   ;Now W has $00FF
LSRW         ;Now W has $007F, C is set (LSRW for 6309 0nly)
```

Encoding: $1XX4

XX = register. For register D, XX=04. For register W*, XX=05.

Source Forms: MUL

Operation: D’ <- A * B

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – 1 if result zero, else 0
V’ – N/C
C’ – 1 if B’.7 set, else 0
E’ – N/C
F’ – N/C
I’ – N/C

Description: Multiplies the 8-bit unsigned contents of the A and B registers, and stores the 16-bit unsigned result in the D register (A:B).

Addressing Modes: Inherent

Comments:

Examples:

```
LDA  #16    ;Now A has 16
LDB  #45    ;Now B has 45
MUL         ;Multiply; now D has 720 decimal
```

Encoding: $3D

MULD

Signed Double-Precision Integer Multiply

Source Forms: `MULD p`

Operation: Q' <- D * (m)

Condition Codes:

H' – N/C
N' – 1 if result < 0, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – Always cleared

E' – N/C
F' – N/C
I' – N/C

Description: Multiplies the 16-bit signed contents of the D register (A:B) by the 16-bit signed contents of the memory location referenced by the addressing mode, and stores the 32-bit signed result in the Q register (A:B:E:F).

Addressing Modes:
Immediate
Direct
Indexed
Extended

Comments:

Examples:

```
LDD #$4567  ;Now D has $4567
MULD #$1001 ;Multiply by $1001
...         ;Now Q has $0456B567 (D has $0456, W has $B567)
```

Encoding:

```
$11XF (+ post-bytes for addressing mode)

X = addressing mode (8=immed., 9=direct, A=indexed,
B=extended)
```

Source Forms:

```
NEGr
NEG pp
```

Operation: r' <- 0 - r; or (m)' <- 0 - (m)

Condition Codes:

H' – Undefined	E' – N/C
N' – 1 if result negative, else 0	F' – N/C
Z' – 1 if result zero, else 0	I' – N/C
V' – 1 if value was 10000000	
C' – 1 if borrow, else 0	

Description: Replaces the specified operand with its 2's complement.

The V bit will be set only if the operand is $80 (representing -128); the negative of this value is 128, which cannot be expressed as an 8-bit signed integer; the result in this case is $80.

The C bit will be cleared only if the operand is $00; the negative of this value is $00, so no borrow is required to calculate it. In all other cases, V will be cleared and C will be set.

Addressing Modes: Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction negates the contents of a register. Any one of the following registers may be specified by r: A, B

Examples:

```
LDA #$0F    ;Now A has $0F
NEGA        ;Now A has $F1, C is set, V is cleared
```

Encoding:

```
$X0                  (X = register; 4=A, 5=B)
$X0 (+ post-bytes) (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: NEGr

Operation: r' <- 0 - r

Condition Codes:

- H' – Undefined
- N' – 1 if result negative, else 0
- Z' – 1 if result zero, else 0
- V' – 1 if value was $8000
- C' – 1 if borrow, else 0
- E' – N/C
- F' – N/C
- I' – N/C

Description: Replaces the specified operand with its 2's complement.

The V bit will be set only if the operand is $8000 (-32768); the negative of this value is 32768, which cannot be expressed as a 16-bit signed integer; the result in this case is $8000.

The C bit will be cleared only if the operand is $0000; the negative of this value is $0000, so no borrow is required to calculate it. In all other cases, V will be cleared and C will be set.

Addressing Modes: Inherent

Comments: This instruction negates the contents of a register. Any one of the following registers may be specified by r: D*

Examples:

```
LDD #$00FF  ;Now D has $00FF
NEGD        ;Now D has $FF01, C is set, V is cleared
            ;(6309 Only)
```

Encoding:

```
$1XX0            (XX = register; 04=D, 05=W)
```

NOP

Source Forms: NOP

Operation:

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction has no effect on the processor, memory, or peripherals.

The NOP instruction occupies a single memory location, and always takes the same amount of time to execute.

Addressing Modes: Inherent

Comments: NOP is often used during program debugging. You can effectively eliminate instructions from a program by replacing them with a series of NOP instructions having the same total length.

NOP is also sometimes used in timing loops.

Examples:

```
      LDA #15  ;Initialize loop counter
L1:   NOP      ;Burn some time
      DECA     ;Decrement loop counter
       BNE L1  ;Keep going until 15 times
```

Encoding: $12

OIM

Source Forms: `OIM #n,pp`

Operation: (m)’ <- (m) | n

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always cleared
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Bit-wise ORs the 8-bit contents of the addressed memory location with the 8-bit immediate data, and stores the result at the memory location.

Addressing Modes:
Direct
Indexed
Extended

Comments: This instruction executes an indivisible read-modify-write cycle.

Examples:

```
OIM #$80,<FLAGS ;Set the most-significant bit of FLAGS
```

Encoding:

```
$X1YY (+ post-bytes for addressing mode)
X = addressing mode (0 = direct, 6 = indexed, 7 = extended);
YY = immediate data
```

Bit-Wise Inclusive-OR Memory Contents to Register

Source Forms: ORr p

Operation: r' <- r | (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise inclusive-ORs the contents of the memory byte to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively sets bits of the specified register. For every set bit in the operand, the instruction sets the corresponding register bit. The instruction does not change register bits corresponding to clear bits in the operand.

Examples:

```
LDA  ,X      ;Get a value from memory
ORA  #$55    ;Set every other bit of the value
```

Encoding:

```
$XA (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

Source Forms: `ORr p`

Operation: r' <- r | (m)

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise inclusive-ORs the contents of the memory word to the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction selectively sets bits of the specified register. For every set bit in the operand, the instruction sets the corresponding register bit. The instruction does not change register bits corresponding to clear bits in the operand.

Examples:

```
LDD  ,X      ;Get a 16-bit value from memory
ORD  #$5555  ;Set every other bit of the value
```

Encoding:

```
$10XA (+ post-bytes)  Register D*

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

ORCC

OR Immediate Data to Condition Code Register

Source Forms: `ORCC #n`

Operation: CC’ <- CC | n

Condition Codes:

H’ – H | n.5
N’ – N | n.3
Z’ – Z | n.2
V’ – V | n.1
C’ – C | n.0
E’ – E | n.7
F’ – F | n.6
I’ – I | n.4

Description: Bit-wise ORs the 8-bit contents of the condition code register with the 8-bit immediate data, and stores the result in the condition code register.

Addressing Modes: Immediate (8-Bit)

Comments: The ORCC instruction may be used to set any desired bits into the condition code register. ORCC is most often used to set the F and I bits, to disable hardware interrupts during critical sections of a program. It is also used to set the carry bit CC.C.

Examples:

```
ORCC #$50  ;Disable IRQ and FIRQ interrupts
```

Encoding:

```
$1ANN

NN = immediate data to OR into condition code register.
```

ORR

Source Forms: `ORR r1,r2`

Operation: r2’ <- r2 | r1

Condition Codes:

H’ – N/C (unless r2 = CC)
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always cleared
C’ – N/C
E’ – N/C (unless r2 = CC)
F’ – N/C (unless r2 = CC)
I’ – N/C (unless r2 = CC)

Description: Bit-wise ORs the contents of two registers, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be ORed. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB’s of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
ORR X,Y     ;Now X still has $1234 and Y has $5777
```

Encoding: `$1035XY`

X = first register, Y = second register (gets result) from table above

PSH

Push Selected Registers Onto Stack

Source Forms:

```
PSHr rl     ;Form 1
PSHr #n     ;Form 2
```

Operation: for each specified register ‘z’: r’ <- r - sizeof(z); (r-sizeof(z))’ <- z

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: This instruction pushes zero or more register values onto either the system or the user stack. When using Form 2, set bits in the immediate operand identify the registers to be pushed, as follows:

```
Bit # Set:   7   6   5   4   3   2   1   0
Register:   PC  S/U  Y   X  DP   B   A  CC
```

For either form, registers are pushed by scanning the equivalent Form 2 operand from left to right (e.g. Y before B; CC last)

Addressing Modes: Immediate

Comments: PSH pushes the low-order byte of a 16-bit register before pushing its high-order byte; programs can manipulate stacked register values like any other 16-bit quantity. If operand bit n.6 is set, PSHS pushes register U to the system stack; PSHU pushes register SP to the user stack. Refer to PSHSW for more information.

Examples:

```
PSHS  U,Y,D      ;Push U, Y, B, and A to system stack
```

Encoding:

```
$3XNN     ;Register S or U

X=register; 4=S, 6=U. N=immediate register selection bit
field operand
```

PSHSW

Push the W Register to the System Stack

Source Forms: `PSHSW`

Operation: SP’ <- SP - 2; (SP-2)’ <- W

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – N/C
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: This instruction pre-decrements the system stack pointer (register SP) and then stores the contents of the W register at the 16-bit memory location addressed by SP.

Addressing Modes: Inherent

Comments: The standard PSHS instruction cannot accommodate the W register. You must use PSHSW, or another pre-decrement / store instruction, to store W on the stack.

Examples:

```
PSHSW       ;Push W to system stack
```

Encoding: `$1038`

PSHUW

Source Forms: `PSHUW`

Operation: U' <- U - 2; (U-2)' <- W

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction pre-decrements the user stack pointer (register U) and then stores the contents of the W register at the 16-bit memory location addressed by U.

Addressing Modes: Inherent

Comments: The standard PSHU instruction cannot accommodate the W register. You must use PSHUW, or another pre-decrement / store instruction, to store W on the user stack.

Examples:

```
PSHUW      ;Push W to user stack
```

Encoding: `$103A`

Source Forms:

```
PULr rl     ;Form 1
PULr #n     ;Form 2
```

Operation: for each specified register ‘z’: z’<-(r-sizeof(z)); r’ <- r + sizeof(z);

Condition Codes:

H’ – N/C (unless n.0 set)
N’ – N/C (unless n.0 set)
Z’ – N/C (unless n.0 set)
V’ – N/C (unless n.0 set)
C’ – N/C (unless n.0 set)
E’ – N/C (unless n.0 set)
F’ – N/C (unless n.0 set)
I’ – N/C (unless n.0 set)

Description: This instruction pulls zero or more register values from either the system or the user stack. When using Form 2, set bits in the immediate operand identify the registers to be pulled, as follows:

```
Bit # Set:   7   6   5   4   3   2   1   0
Register:   PC  S/U  Y   X  DP   B   A  CC
```

For either form, registers are pulled by scanning the equivalent Form 2 operand from right to left (e.g. CC first; B before Y).

Addressing Modes: Immediate

Comments: PUL pulls the high-order byte of a 16-bit register before pulling its low-order byte; programs can manipulate stacked register values like any other 16-bit quantity. If operand bit n.6 is set, PULS pulls register U from the system stack; PULU pulls register SP from the user stack. Refer to PULSW for more information.

Examples:

```
PULS  D,Y,U      ;Pull A, B, Y, and U from system stack
```

Encoding:

```
$3XNN     ;Register S or U

X=register; 5=S, 7=U. N=immediate register selection bit
field operand
```

PULSW

Source Forms: `PULSW`

Operation: W' <- (SP); SP' <- SP + 2

Condition Codes:

H' – N/C	E' – N/C
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: This instruction loads the contents of the 16-bit memory location addressed by SP into the W register and then post-increments the system stack pointer (register SP).

Addressing Modes: Inherent

Comments: The standard PULS instruction cannot accommodate the W register. You must use PULSW, or another load / post-increment instruction, to load W from the stack.

Examples:

```
PULSW      ;Pull W from system stack
```

Encoding: `$1039`

PULUW

Source Forms: `PULUW`

Operation: W' <- (U); U' <- U + 2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction loads the contents of the 16-bit memory location addressed by U into the W register and then post-increments the system stack pointer (register U).

Addressing Modes: Inherent

Comments: The standard PULU instruction cannot accommodate the W register. You must use PULUW, or another load / post-increment instruction, to load W from the user stack.

Examples:

```
PULUW      ;Pull W from user stack
```

Encoding: `$103B`

Rotate the Operand and Carry Left One Bit

Source Forms:

```
ROLr
ROL pp
```

Operation: r' <- r << 1; r.0' <- CC.C; CC.C' <- r.7; CC.V' <-r.7 ^ r.6; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if sign changed, else 0
C' – Set to MSB of operand
E' – N/C
F' – N/C
I' – N/C

Description: Rotates the specified operand and carry left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The original carry bit is copied to the least-significant bit of the result.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes:

Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction rotates the contents of a register. Any one of the following registers may be specified by r: A, B.

Examples:

```
LDA #$0F    ;Now A has $0F
ORCC #1     ;Force carry set
ROLA        ;Now A has $1F, C is clear
```

Encoding:

```
$X9                  (X = register; 4=A, 5=B)
$X9 (+ post-bytes) (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: ROLr

Operation: r’ <- r << 1; r.0’ <- CC.C; CC.C’ <- r.15; CC.V’ <-r.15 ^ r.14

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – 1 if sign changed, else 0
C’ – Set to MSB of operand
E’ – N/C
F’ – N/C
I’ – N/C

Description: Rotates the specified operand and carry left one position. The most-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The original carry bit is copied to the least-significant bit of the result.

The overflow bit (V) of the condition code is set if the most-significant bit (the sign bit) of the result differs from the most-significant bit of the operand.

Addressing Modes: Inherent

Comments: This instruction rotates the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Examples:

```
LDW #$00FF   ;Now W has $00FF
ORCC #1      ;Force carry set
ROLW         ;Now W has $01FF, C is clear (ROLW for 6309 0nly)
```

Encoding: $1XX9 (XX = register; 04=D, 05=W)

Source Forms:

```
RORr
ROR pp
```

Operation: r' <- r >> 1; r.7' <- CC.C; CC.C' <- r.0; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Rotates the specified operand and carry right one position. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The original carry bit is copied to the most-significant bit of the result.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction rotates the contents of a register. Any one of the following registers may be specified by r: A, B

Examples:

```
LDA #$0F     ;Now A has $0F
ORCC #1      ;Force carry set
RORA         ;Now A has $87, C is set
```

Encoding:

```
$X6                 (X = register; 4=A, 5=B)
$X6 (+ post-bytes)  (X=mode; 0=direct, 6=indexed, 7=extended)
```

Source Forms: RORr

Operation: r' <- r >> 1; r.15' <- CC.C; CC.C' <- r.0

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – N/C
C' – 1 if operand odd, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Rotates the specified operand and carry right one position. The least-significant bit of the original operand is copied to the carry bit (C) of the condition code register. The original carry bit is copied to the most-significant bit of the result.

Addressing Modes: Inherent

Comments: This instruction rotates the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Examples:

```
LDW #$00FF  ;Now W has $00FF
ORCC #1     ;Force carry set
RORW        ;Now W has $807F, C is set (RORW for 6309 0nly)
```

Encoding:

```
$1XX6              (XX = register; 04=D, 05=W)
```

RTI

Return From Interrupt

Source Forms: RTI

Operation: pull register values (including CC and PC) from stack

Condition Codes:

H’ –(SP).5
N’ –(SP).3
Z’ –(SP).2
V’ –(SP).1
C’ –(SP).0
E’ –(SP).7
F’ –(SP).6
I’ –(SP).4

Description: This instruction pulls CC, PC, and possibly other register values from the system stack. The exact operation of the instruction depends on the curent processor mode (Native or Emulation), and on the state of the stacked CC.E bit, as follows:

Mode	(SP).E	Registers Pulled (in order)
Emulation	0	CC, PC
Emulation	1	CC, A, B, DP, X, Y, U, PC
Native	0	CC, PC
Native	1	CC, A, B, E, F, DP, X, Y, U, PC

Addressing Modes: Inherent

Comments: This instruction is most often used at the end of an interrupt service routine, to restore register values (including PC) to their states just prior to the interrupt. Refer to the descriptions of CWAI, LDMD, PULS, SWI, SWI2, SWI3, IRQ, FIRQ, and NMI for additional information.

Examples:

```
LDA >HWDATA   ;Trivial interrupt service: get device data,
STA >MYTEMP   ; save device data,
CLR >HWSTAT   ; clear device status,
RTI           ;Return from interrupt
```

Encoding: $3B

RTS

Source Forms: RTS

Operation: PC' <- (SP); SP' <- SP+2

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction pulls the value of the program counter (register PC) from the system stack.

Addressing Modes: Inherent

Comments: This instruction is most often used at the end of a subroutine, to return control to the instruction immediately following the calling JSR, LBSR or BSR. Refer to the descriptions of JSR, LBSR, BSR, and PULS for additional information.

Examples:

```
     BSR RND     ;Call a subroutine called "RND"
     JMP AWAY    ;Go somewhere else when done...
RND: ADDD #128   ;Trivial subroutine: Add 128 to register D
     RTS         ;Return from subroutine
```

Encoding: $39

Source Forms: `SBCr p`

Operation: r' <- r - (m) - CC.C

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts both the contents of the memory byte, and the carry bit of the condition code register, from the contents of the A or B register. The result is stored in the specified register.

Addressing Modes:

Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for multi-precision subtraction, since it allows the carry from a previous byte or word subtraction to be factored into a subsequent byte subtraction.

Examples:

```
SUBB #3     ;Subtract from B register - could create a carry
SBCA #0     ;Now the D register has been decremented by 3
```

Encoding:

```
$X2 (+ post-bytes)    Register A or B

X = register and mode. For register A, 8=immediate, 9=direct,
A=indexed, B=extended. For register B, C=immediate, D=direct,
E=indexed, F=extended.
```

SBC(16-bit) — Subtract Memory Contents and Carry from Register

Source Forms: SBCr p

Operation: r' <- r - (m) - CC.C

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts both the contents of the memory word, and the carry bit of the condition code register, from the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for multi-precision subtraction, since it allows the carry from a previous byte or word subtraction to be factored into a subsequent word subtraction.

Examples:

```
SUBE 3,S    ;Subtract value on stack from E register
SBCD #0     ;Accumulate 24 bit result in A.B.E registers
```

Encoding:

```
$10X2 (+ post-bytes)  Register D

X = register and mode. For register D, 8=immediate, 9=direct,
A=indexed, B=extended.
```

SBCR

Source Forms: `SBCR r1,r2`

Operation: r2' <- r2 - r1 - CC.C

Condition Codes:

- H' – Undefined
- N' – 1 if result negative, else 0
- Z' – 1 if result zero, else 0
- V' – 1 if overflow, else 0
- C' – 1 if carry, else 0
- E' – N/C (unless r2 = CC)
- F' – N/C (unless r2 = CC)
- I' – N/C (unless r2 = CC)

Description: Subtracts the contents of two registers, and the carry bit, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be subtracted. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

0000 D (A:B)	0100 SP	1000 A	1100 ---
0001 X	0101 PC	1001 B	1101 ---
0010 Y	0110 W (E:F)*	1010 CC (CCR)	1110 E *
0011 U (US)	0111 V *	1011 DP (DPR)	1111 F *

Addressing Modes: Register (immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
COMB          ;Force carry bit set
LDY #$4567    ;Now Y has $4567
LDX #$1234    ;Now X has $1234
SBCR X,Y      ;Now X still has $1234 and Y has $3332
```

Encoding: `$1033XY`

X = first register, Y = second register (gets result) from table above

SEX

Sign-Extend Register B to Register D

Source Forms: SEX

Operation: A’ <- (unsigned)B.7 * 255

Condition Codes:

H’ – N/C
N’ – Set if result < 0; else clear
Z’ – Set if result 0; else clear
V’ – N/C
C’ – N/C

E’ – N/C
F’ – N/C
I’ – N/C

Description: This instruction converts a signed 8-bit value in B to a signed 16-bit value in D.

Addressing Modes: Inherent

Comments: The SEX instruction is most often used prior to performing an arithmetic operation between an 8-bit value and a 16-bit value.

Examples:

```
LDB 3,X        ;Get an 8-bit value from memory
SEX            ;Convert to 16-bit value
ADDD >OLDSUM   ;Add another 16-bit value (OLDSUM) to it
```

Encoding: $1D

SEXW

Source Forms: SEXW

Operation: E' <- (unsigned)F.7 * 255

Condition Codes:

H' – N/C
N' – Set if result < 0; else clear
Z' – Set if result 0; else clear
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: This instruction converts a signed 8-bit value in F to a signed 16-bit value in W.

Addressing Modes: Inherent

Comments: The SEXW instruction is most often used prior to performing an arithmetic operation between an 8-bit value and a 16-bit value.

Examples:

```
LDE 3,X       ;Get an 8-bit value from memory
SEXW          ;Convert to 16-bit value in W
ADDR W*,D      ;Add 16-bit contents of D register to W
```

Encoding: $14

ST(8-

Store Contents of Register at Memory Address

Source Forms: STr p

Operation: (m)' <- r

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies the contents of the specified register to the memory byte. The condition code register is updated according to the result.

Addressing Modes:
Extended
Direct
Indexed

Comments: The value stored may be either signed or unsigned, depending on how the program interprets the condition code register after executing the ST instruction.

Examples:

```
STA  ,X      ;Store register A at location pointed to by X
```

Encoding:

```
$X7 (+ post-bytes)    Register A or B
$11X7 (+ post-bytes)  Register E* or F*

X = register and mode. For register A or E, 9=direct,
A=indexed, B=extended. For register B or F,  D=direct,
```

Source Forms: STr p

Operation: (m)' <- r

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always 0
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies the contents of the specified 16-bit register to the memory word. The condition code register is updated according to the result.

Addressing Modes:
Extended
Direct
Indexed

Comments: The value stored may be either signed or unsigned, depending on how the program interprets the condition code register after executing the ST instruction.

Examples:

```
STD   ,X       ;Store register D at location pointed to by X
```

Encoding:

```
$XY                     Register D, U or X
$10XY (+ post-bytes)  Register S, W*, or Y

X = mode; for W*, X, or Y, 9=direct, A=indexed, B=eextended;
for D, S or U,  D=direct, E=indexed, F=extended. Y =
```

Store Register Contents at Memory Address

Source Forms: STr p

Operation: (m)’ <- r

Condition Codes:

H’ – N/C
N’ – 1 if result negative, else 0
Z’ – 1 if result zero, else 0
V’ – Always 0
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Copies the contents of the specified 32-bit register to the long memory word. The condition code register is updated according to the result.

Addressing Modes:
Extended
Direct
Indexed

Comments: The value stored may be either signed or unsigned, depending on how the program interprets the condition code register after executing the ST instruction.

Examples:

```
STQ  ,X      ;Store register Q at location pointed to by X
```

Encoding:

```
$10XD (+ post-bytes)  Register Q*

X = mode; for register Q,  D=direct, E=indexed, F=extended.
```

STBT

Store Bit of Register to Bit of Memory

Source Forms: `STBT rr.n,qq.k`

Operation: (DP:qq).k' <- rr.n

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Copies bit (n) of register (rr) to bit (k) of 8-bit direct page memory location (qq). The register and other bits of the memory location are not modified. Two bits in the post-byte specify the register, three bits specify the memory location bit number, and three bits specify the register bit number. The register (rr) is selected from the following table:

```
00 CC (CCR)  01 A         10 B         11 ---
```

Addressing Modes: Bit Function (immed. register number, immed. bit numbers, direct operand)

Comments: The bit number 000 is used for the least-significant bit of an 8-bit value. The bit number 111 is used for the most-significant bit.

Examples:

```
STBT A,7,FLAGS,2 ;Set bit 2 of FLAGS to bit 7 of register A
STBT CC,FLAGS,3  ;Program bit 3 of FLAGS to match carry bit
```

Encoding:

$1137XXYY
XX = post-byte (RRkkknnn); YY = direct page address. RR = register number from above table; kkk = memory bit #; nnn = register bit #

Source Forms:

```
SUBA p
SUBB p
SUBE p
SUBF p
```

Operation: r' <- r - (m)

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts the contents of the memory byte from the contents of the specified register. The result is stored in the specified register.

Addressing Modes:

Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for single-precision subtraction, or for the first byte of multi-precision subtraction, since it allows subtraction of two quantities while ignoring any carry left over from a previous operation.

Examples:

```
SUBB #3     ;Subtract from B register - could create a carry
SBCA #0     ;Now the D register has been decremented by 3
```

Encoding:

```
$X0 + post-bytes    (A and B registers)
$11X0 + post-bytes  (E and F registers)
X = addressing mode (for A or E: 8 = immed., 9 = direct, A =
indexed, B = extd.; for B or F: C = immed., D = direct, E =
indexed, F = extd.)
```

Source Forms:

```
SUBD p
SUBW p
```

Operation: r' <- r - (m)

Condition Codes:

H' – Undefined
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – 1 if overflow, else 0
C' – 1 if carry, else 0
E' – N/C
F' – N/C
I' – N/C

Description: Subtracts the contents of the memory word from the contents of the specified register. The result is stored in the specified register.

Addressing Modes:
Immediate
Extended
Direct
Indexed

Comments: This instruction can be used for single-precision subtraction, or for the first word of multi-precision subtraction, since it allows subtracting two quantities ignoring any carry left over from previous operations.

Examples:

```
LDW  ,X      ;Get a value from memory
SUBW <BASE   ;Subtract predetermined BASE offset
```

Encoding:

```
$X3 (+ post-bytes)     Register D
$11X0 (+ post-bytes)   Register W
```

X = addressing mode. For registers D and W*, 8 = immediate, 9 = direct, A = indexed, B = extended.

SUBR

Subtract Register from Register

Source Forms: `SUBR r1,r2`

Operation: r2' <- r2 - r1

Condition Codes:

- H' – Undefined
- N' – 1 if result negative, else 0
- Z' – 1 if result zero, else 0
- V' – 1 if overflow, else 0
- C' – 1 if carry, else 0
- E' – N/C (unless r2 = CC)
- F' – N/C (unless r2 = CC)
- I' – N/C (unless r2 = CC)

Description: Subtracts the contents of two registers, and stores the result in the second register. The post-byte of this instruction identifies the two registers to be subtracted. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is a 16 bit register, r1 is treated as the equivalent 16 bit register even if it is only 8 bits (e.g. r2=X, r1=A or B forces r1 to D).

If r1 is an 8 bit register and r2 is a 16 bit register, only the 8 LSB's of r1 are used. r2=CC should not be used.

Examples:

```
LDY #$4567  ;Now Y has $4567
LDX #$1234  ;Now X has $1234
SUBR X,Y    ;Now X still has $1234 and Y has $3333
```

Encoding: `$1032XY`

`X = first register, Y = second register (gets result) from table above`

SWI

Source Forms: SWI

Operation: CC.E' <-1; Stack all registers; CC.F' <- 1; CC.I' <- 1; PC' <- ($FFFA)

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – Always 1
F' – Always 1
I' – Always 1

Description: The SWI instructuction pushes the current values of all registers to the system stack. It then transfers control to the address stored at memory location $FFFA. The exact operation of SWI depends on the current processor mode (Emulation or Native) as follows:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFFA normally exits via an RTI instruction. SWI is most often used to implement breakpoints in software debuggers, and to provide "application" programs with access to "operating system" utility routines. SWI may be executed regardless of the state of the CC.I and CC.F bits.

Examples:

```
      SWI       ;Perform Software Interrupt
NEXT: NOP       ;End up here after doing SWI service routine
```

Encoding: $3F

SWI2

Source Forms: `SWI2`

Operation: CC.E' <-1; Stack all registers; PC' <- ($FFF4)

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – Always 1
F' – N/C
I' – N/C

Description: The SWI2 instructuction pushes the current values of all registers to the system stack. It then transfers control to the address stored at memory location $FFF4. The exact operation of SWI2 depends on the current processor mode (Emulation or Native) as follows:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF4 normally exits via an RTI instruction. SWI2 is most often used to provide "application" programs with access to "operating system" utility routines. SWI2 may be executed regardless of the state of the CC.I and CC.F bits. Note that SWI2 does not disable subsequent interrupts.

Examples:

```
      SWI2      ;Perform Software Interrupt 2
NEXT: NOP       ;End up here after doing SWI2 service routine
```

Encoding: $103F

SWI3

Source Forms: `SWI3`

Operation: CC.E' <-1; Stack all registers; PC' <- ($FFF2)

Condition Codes:

H' – N/C	E' – Always 1
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: The SWI3 instructuction pushes the current values of all registers to the system stack. It then transfers control to the address stored at memory location $FFF2. The exact operation of SWI3 depends on the current processor mode (Emulation or Native) as follows:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF2 normally exits via an RTI instruction. SWI3 is most often used to provide “application” programs with access to “operating system” utility routines. SWI3 may be executed regardless of the state of the CC.I and CC.F bits. Note that SWI3 does not disable subsequent interrupts.

Examples:

```
      SWI3      ;Perform Software Interrupt 3
NEXT: NOP       ;End up here after doing SWI3 service routine
```

Encoding: `$113F`

SYNC

Source Forms: `SYNC`

Operation: Stop processing instructions; wait for interrupt; resume

Condition Codes:

H' – N/C	E' – N/C
N' – N/C	F' – N/C
Z' – N/C	I' – N/C
V' – N/C	
C' – N/C	

Description: When the processor executes a SYNC instruction, no other instructions will be executed until after the next hardware interrupt. This synchronizes the processor to the hardware interrupt. This instruction may be executed regardless of the state of the I and F interrupt mask bits.

While waiting for the next hardware interrupt, the processor places its address and data busses in a high-impedance state. On the 6309 and 6309E, the processor also enters a low-power “sleep mode”.

Addressing Modes: Inherent

Comments: If the interrupt is enabled (NMI, or appropriate interrupt mask bit cleared) and the next hardware interrupt lasts more than 3 cycles, the processor responds normally to the interrupt and then continues execution. If the interrupt is masked, or is shorter than 3 cycles, execution continues immediately.

Examples:

```
ORCC #$50  ;Disable interrupts
SYNC       ;Wait for video blanking interrupt
STA 3,Y    ;Store character from A to video screen at Y
```

Encoding: `$13`

TFM

Source Forms:

```
TFM r1+,r2+  ;Form 1 (forward block move M[r1] to M[r2])
TFM r1-,r2-  ;Form 2 (backward block move M[r1] to M[r2])
TFM r1+,r2   ;Form 3 (output bytes at M[r1] to peripheral)
TFM r1,r2+   ;Form 4 (input bytes from peripheral to M[r2])
```

Operation: while (W<>0) [(r2)’ <- (r1); adjust r1; adjust r2; W’ <- W-1]

Condition Codes:

H’ – N/C
N’ – N/C
Z’ – Always set
V’ – N/C
C’ – N/C
E’ – N/C
F’ – N/C
I’ – N/C

Description: Moves W bytes of data from the memory location(s) pointed to by r1 to the memory location(s) pointed to by r2. The post-byte of this instruction identifies the two pointer registers r1, r2.

The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D      0001 X      0010 Y      0011 U (US)      0100 SP
```

Addressing Modes: Register
(immed. register numbers)

Comments: Each incremented or decremented register changes by W. W is cleared at the end of the instruction. The instruction is interruptible. An interrupt during Form 4 re-reads the peripheral referenced by r1 without storing the previous data byte, advancing r2, or decrementing W; use Form 4 only with interrupts disabled.

Examples:

```
LDX #$FF21  ;Address of I/O port
LDY #MYBUFF ;Address of 256 byte buffer
LDW #$0100  ;Byte count (256)
TFM (Y+,X)  ;Send 256 bytes to peripheral from buffer MYBUFF
```

Encoding:

```
$1138JK     (Form 1, J=source, K=destination pointer register
from table)
$1139JK     (Form 2, J=source, K=destination pointer register
from table)
$113AJK     (Form 3, J=source, K=destination pointer register
```

TFR

Transfer Data From One Register to Another

Source Forms: `TFR r1,r2`

Operation: r2' <- r1

Condition Codes:

H' – N/C (unless r2 = CC)
N' – N/C (unless r2 = CC)
Z' – N/C (unless r2 = CC)
V' – N/C (unless r2 = CC)
C' – N/C (unless r2 = CC)
E' – N/C (unless r2 = CC)
F' – N/C (unless r2 = CC)
I' – N/C (unless r2 = CC)

Description: Copies data in register r1 to another register r2 of the same size. The post-byte of this instruction identifies the source (r1) and destination (r2) of the data. The post-byte consists of two 4-bit codes, each of which identifies a register as follows:

```
0000 D (A:B)  0100 SP       1000 A         1100 ---
0001 X        0101 PC       1001 B         1101 ---
0010 Y        0110 W (E:F)* 1010 CC (CCR)  1110 E *
0011 U (US)   0111 V *      1011 DP (DPR)  1111 F *
```

Addressing Modes: Register
(immed. register numbers)

Comments: If r2 is the condition code register (CC), all condition code flags are set to the values of corresponding bits in r1.

The E, F, W*, and V registers are present only in the HD6309 and HD6309E processor series.

Examples:

```
LDD #$0000  ;Now D has $0000
LDX #$1234  ;Now X has $1234
TFR D,X     ;D still has $0000, so does X now
```

Encoding: $1FXY

X = source register, Y = destination register from table above

Source Forms: `TIM #n,pp`

Operation: (m) & n

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Bit-wise ANDs the 8-bit contents of the addressed memory location with the 8-bit immediate data, and sets the condition code register according to the result. The addressed memory location is not modified.

Addressing Modes:
Direct
Indexed
Extended

Comments: This instruction executes a standard read cycle, rather than an indivisible read-modify-write cycle.

Examples:

```
TIM #$80,<FLAGS ;Clear Z if most-significant bit of FLAGS set
```

Encoding:

```
$XBYY (+ post-bytes for addressing mode)
X = addressing mode (0 = direct, 6 = indexed, 7 = extended);
YY = immediate data
```

Source Forms:

```
TSTr
TST pp
```

Operation: r - 0; {or for (m)}

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – N/C
E' – N/C
F' – N/C
I' – N/C

Description: Compares the specified signed operand to the constant zero, and updates the condition codes. This instruction does not modify the operand.

Addressing Modes:
Inherent
Direct
Indexed
Extended

Comments: When inherent addressing is used, this instruction tests the contents of a register. Any one of the following registers may be specified by r: A, B, E*, F*.

Note that TST does not effect the carry bit.

Examples:

```
     LDB #COUNT  ;Set iteration counter
LOOP TSTB        ;Check for B=0 each time
     BEQ DONE    ; if B=0, exit this loop
     ...
```

Encoding:

```
$XD                  (X = register; 4=A, 5=B)
$1XXD                (XX = register; 14=E, 15=F)
$XD (+ post-bytes) (X=mode; 0=direct, 6=indexed, 7=extended)
```

TST(16-bit) — Test the Operand Against Zero

Source Forms: TSTr

Operation: r - 0

Condition Codes:

H' – N/C
N' – 1 if result negative, else 0
Z' – 1 if result zero, else 0
V' – Always cleared
C' – N/C

E' – N/C
F' – N/C
I' – N/C

Description: Compares the specified signed operand to the constant zero, and updates the condition codes. This instruction does not modify the operand.

Addressing Modes: Inherent

Comments: This instruction tests the contents of a register. Any one of the following registers may be specified by r: D*, W*.

Note that TST does not effect the carry bit.

Examples:

```
     LDD #COUNT  ;Set iteration counter
LOOP TSTD        ;Check for D=0 each time
     BEQ DONE    ; if D=0, exit this loop
     ...
```

Encoding: $1XXD (XX = register; 04=D, 05=W)

DIV0

Division by Zero Trap

Source Forms: (none)

Operation: CC.E' <-1; Stack registers; CC.F', CC.I', MD.7' <- 1; PC' <- ($FFF0)

Condition Codes:

H' – N/C	E' – Always 1
N' – N/C	F' – Always 1
Z' – N/C	I' – Always 1
V' – N/C	
C' – N/C	

Description: The 6309 automatically executes DIV0 whenever the divisor of DIVD or DIVQ is zero. DIV0 pushes the current values of all registers to the system stack, and transfers control to the address stored at memory location $FFF0. The exact operation of DIV0 depends on the current processor mode as follows:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF0 normally exits via an RTI instruction. DIV0 is most often used by software debuggers, to detect runaway programs. DIV0 may be executed regardless of the state of the CC.I and CC.F bits. The DIV0 vector is shared with ILLOP; the 6309 sets bit MD.7 if an DIV0 occured (see BITMD).

Examples:

```
      DIVD #0     ;Force division by 0 - perform DIV0
NEXT: NOP         ;End up here after DIV0 service routine
```

Encoding: (none)

FIRQ

Source Forms: (none)

Operation: CC.E' <- MD.1; Stack registers; CC.F', CC.I' <- 1; PC' <- ($FFF6)

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – Always 1
F' – Always 1
I' – Always 1

Description: The 6309 executes FIRQ as the next instruction whenever external hardware pulls the FIRQ* pin low with CC.F clear. FIRQ transfers control to the address stored at memory location $FFF6. Operation depends on processor and FIRQ modes:

Proc. Mode	FIRQ Mode	Registers Pushed (in order)
Emulation	FIRQ	PC, CC
Emulation	IRQ	PC, U, Y, X, DP, B, A, CC
Native	FIRQ	PC, CC
Native	IRQ	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF6 normally exits via an RTI instruction. FIRQ is most often used to implement high-speed asynchronous I/O functions. FIRQ may be executed regardless of the state of the CC.I bit, but will not be executed while CC.F is set. Refer to descriptions of LDMD and CWAI for additional information.

Examples:

```
START: (any instruction) ;FIRQ occurs during this instruction
                         ;FIRQ routine executes here
NEXT:  (any instruction) ;Come here after doing FIRQ routine.
```

Encoding: (none)

Source Forms: (none)

Operation: CC.E’ <-1; Stack registers; CC.F’, CC.I’, MD.6’ <- 1; PC’ <- ($FFF0)

Condition Codes:

H’ – N/C	E’ – Always 1
N’ – N/C	F’ – Always 1
Z’ – N/C	I’ – Always 1
V’ – N/C	
C’ – N/C	

Description: The 6309 automatically executes ILLOP in place of an otherwise undefined (or illegal) instruction. ILLOP advances PC past the offending opcode, pushes all registers to the system stack, and transfers control to the address stored at memory location $FFF0. The exact operation of ILLOP depends on the current processor mode as follows:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF0 normally exits via an RTI instruction. ILLOP is most often used by software debuggers, to detect runaway programs. ILLOP may be executed regardless of the state of the CC.I and CC.F bits. The ILLOP vector is shared with DIV0; the 6309 sets bit MD.6 if an ILLOP occured (see BITMD).

Examples:

```
      FCB $10,$20    ;Invalid instruction - perform ILLOP
NEXT: NOP            ;End up here after ILLOP service routine
```

Encoding: (none)

IRQ

Source Forms: (none)

Operation: CC.E' <-1; Stack all registers; CC.I' <- 1; PC' <- ($FFF8)

Condition Codes:

H' – N/C
N' – N/C
Z' – N/C
V' – N/C
C' – N/C
E' – Always 1
F' – N/C
I' – Always 1

Description: The 6309 automatically executes IRQ as the next instruction whenever external hardware pulls the IRQ* pin low and CC.I is clear. IRQ pushes the current values of all registers to the system stack, and transfers control to the address stored at memory location $FFF8. The operation of IRQ depends on the processor mode:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF8 normally exits via an RTI instruction. IRQ is most often used to implement asynchronous I/O or timing functions. IRQ may be executed regardless of the state of the CC.F bit, but will not be executed while CC.I is set.

Examples:

```
START: (any instruction) ;IRQ occurs during this instruction
                         ;IRQ routine executes here
NEXT:  (any instruction) ;Come here after doing IRQ routine.
```

Encoding: (none)

Source Forms: (none)

Operation: CC.E' <-1; Stack all registers; CC.F' <- 1; CC.I' <- 1; PC' <- ($FFFC)

Condition Codes:

H' – N/C	E' – Always 1
N' – N/C	F' – Always 1
Z' – N/C	I' – Always 1
V' – N/C	
C' – N/C	

Description: The 6309 automatically executes NMI as the next instruction whenever external hardware pulls the NMI* pin low. NMI pushes the current values of all registers to the system stack, and transfers control to the address stored at memory location $FFFC. The exact operation of NMI depends on the current processor mode:

Mode	Registers Pushed (in order)
Emulation	PC, U, Y, X, DP, B, A, CC
Native	PC, U, Y, X, DP, F, E, B, A, CC

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFFC normally exits via an RTI instruction. NMI is most often used to implement high-speed I/O and hardware debugging functions. NMI may be executed regardless of the state of the CC.I and CC.F bits.

Examples:

```
START: (any instruction) ;NMI occurs during this instruction
                         ;NMI routine executes here
NEXT:  (any instruction) ;Come here after doing NMI routine.
```

Encoding: (none)

Source Forms: (none)

Operation: CC.I’ <- 1; CC.F’ <- 1; DP’ <- 0; V’ <- V; MD’ <- 0; PC’ <- ($FFFE)

Condition Codes:

H’ – Undefined
N’ – Undefined
Z’ – Undefined
V’ – Undefined
C’ – Undefined
E’ – Undefined
F’ – Always 1
I’ – Always 1

Description: The 6309 executes RESET as the next instruction whenever external hardware pulls the RESET* pin low. RESET transfers control to the address stored at memory location $FFFE.

RESET always forces the processor into Emulation Mode, clears the DP register, and sets both CC.I and CC.F to disable interrupts. The value of the V register is preserved. All other register values are undefined after RESET.

Addressing Modes: Inherent

Comments: The routine pointed to by memory location $FFF0 normally initializes hardware peripherals and the system stack pointer. RESET is most often used to force the processor into a known state at power-up. RESET may be executed regardless of the state of the CC.I and CC.F bits. See also LDMD for additional information.

Examples:

```
BUSY: (any instruction)  ;RESET occurs during this instr.
...
DORST: (any instruction) ;RESET routine executes here.
                         ; No going back!
```

Encoding: (none)

This page intentionally left blank.

SECTION 4
APPLICATION INFORMATION

4.1 INTRODUCTION

This section provides information that will help programmers, already familiar with the 6809, to fully take advantage of the new 6309 features.

Some of the information in this section takes the form of sample programs. To make the sample programs easier to understand, we've left out some optimization and programming tricks that would be obvious to experienced machine language programmers. You are welcome to optimize or modify the sample programs for your own use in any way that you see fit.

4.2 Detecting the 6309

Some programmers may want their software to run on both the 6309 and the 6809. Since the new features of the 6309 aren't available on the MC68B09E, software that runs on either processor must either:

A) Not use any of the new 6309 features, or
B) Determine which processor is present, and use the new features only when executing on an 6309.

The following subroutine determines which processor is present:

```
* Determine whether processor is 6309 or 6809
* Returns Z clear if 6309, set if 6809
CHK309  PSHS  D       ;Save Reg-D
        FDB   $1043   ;6309 COMD instruction (COMA on 6809)
        CMPB  1,S     ;not equal if 6309
        PULS  D,PC    ;exit, restoring D
```

The subroutine relies on the 6809 treating undefined instructions as 1-byte NOPs. When executed on a 6809, the subroutine complements A, but not B; the result of the comparison will be that B equals its saved value. When executed on a 6309, the subroutine complements both A and B; the result of the comparison will be that B does not equal its saved value. The subroutine preserves all register values.

4.3 Detecting 6309 Native Mode

Most programs run without modification in both Emulation Mode and Native Mode. Programs that process software interrupts, include timing loops, or use interrupts to terminate polled I/O loops, are the exception. To run in both modes, these exceptional programs need to modify the behavior of their stack access routines and timing loops, based on the current mode.

It would be easy to determine the processor's operating mode, if the BITMD instruction could access bit 0 of the mode register (bit MD.0). Unfortunately this is not allowed, so we need to resort to more devious means.

The following subroutine determines whether the processor is in Emulation Mode or Native Mode:

```
* Determine whether processor is in Emulation Mode or Native Mode
* Works for 6809 or 6309.
* Returns Z clear if Emulation (or 6809), Z set if Native
CHKNTV  PSHSW                   ;Ignored on 6809 (no stack data)
        PSHS  U,Y,X,DP,D,CC     ;Save all registers
        LEAU  CHKX68,PCR        ;Special exit for 6809 processor
        LDY   #0
        PSHS  U,Y,X,D           ;Push 6809 trap, Native marker, PC temps
        ORCC  #$D0              ;Set CC.E (entire), no interrupts
        PSHS  U,Y,X,DP,D,CC     ;Save regs
        LEAX  CHKXIT,PCR
        STX   10,S              ;Preset Emulation mode PC slot
        STX   12,S              ;Preset Native mode PC slot
        RTI                     ;End up at CHKXIT next
CHKXIT  LDX   ,S++              ;In NATIVE, get 0; in EMULATION, non-zero
         BEQ  CHKNTV9
        LEAS  2,S               ;Discard native marker in EMULATION mode
CHKNTV9 TFR   CC,A
        ANDA  #$0F              ;Keep low CC value
        AIM   #$F0,0,S          ;Keep high bits of stacked CC
        ORA   2,S               ;Combine CC values (skip over 6809 trap)
        STA   2,S               ; and save on stack
        PULSW                   ;Pull bogus W (does RTS to CHKX68 on 6809)
        PULS  CC,D,DP,X,Y,U     ;Restore 6309 registers and return
        PULSW
        RTS
```

```
CHKX68  PULS  CC,D,DP,X,Y,U,PC  ;Restore 6809 registers and return
```

The subroutine relies on the 6809 treating undefined instructions as 1-byte or 2-byte NOPs. It preserves all register values.

Since the subroutine takes relatively long to execute, you may want your programs to call it only once. If a program sets or clears some memory location based on the subroutine's returned condition codes, it can make future mode determinations quickly by examining the memory location. This technique relies on the processor not changing modes once you've called the subroutine.

4.4 Switching Between Emulation Mode and Native Mode

The 6309 powers up in Emulation Mode, and reverts to Emulation Mode when reset. Programs that use Native Mode must explicitly change the processor mode via the LDMD instruction.

The LDMD instruction loads an immediate value into the M (mode) register. Only the two least-significant bits of this value effect processor operation. Bit 0 selects between Emulation Mode (0) and Native Mode (1). Bit 1 selects between 6809-like FIRQ handling (0) and 6309 IRQ-like FIRQ handling (1).

For most applications, you'll set the processor mode and FIRQ handling mode no more than once — at the beginning of the program. In this case, you know exactly which processor and FIRQ modes you want and a simple LDMD instruction sets up the desired modes. For example, the instruction

```
LDMD  #$01
```

forces 6809-like FIRQ handling and places the processor in Native Mode.

If your program is running under an operating system, you may want to leave control of the processor mode to the operating system. Changing the contents of the MD register from within an application could confuse the operating system, since the operating system won't be aware of the change.

This section describes two subroutines that you can use to set the processor mode. The 1st subroutine selects either Native Mode or Emulation Mode and forces the FIRQ handling mode bit to a predetermined value. The 2nd subroutine selects either processor mode, preserving the current FIRQ mode.

Here's the 1st subroutine. It sets the 6309 processor mode to either Native Mode or Emulation Mode, depending on the value in Register A:

```
* Force processor to Emulation Mode or Native Mode,
* depending on value in Register A
* A=0  Emulation Mode
* A<>0 Native Mode
* Works for 6309 only.
SETPMD  TSTA
        BNE   SETNTV
        LDMD  #$00           ;Force Emulation Mode, normal FIRQ
        BRA   SETPND
SETNTV  LDMD  #$01           ;Force Native Mode, normal FIRQ
SETPND  RTS
```

The subroutine directly loads the 6309 mode (M) register with either $00 for Emulation Mode or $01 for Native Mode. This forces the processor into the specified mode, but also resets the Register MD bit controlling how the processor handles FIRQ interrupts. The values $00 and $01 tell the 6309 to handle FIRQ interrupts exactly as they are handled on the 6809. If your program wants the processor to handle FIRQ interrupts just like IRQ interrupts, you should change the values in the subroutine to $02 for Emulation Mode and $03 for Native Mode.

We wouldn't have to worry about which value (e.g. $00 or $02 for Emulation Mode) to store in the MD register, if only we could read the current contents of the MD Register. We could then read the current contents of M, strip off the least-significant bit, update the least-significant bit for to select the desired mode, and write the updated value to MD.

The 6309 provides the BITMD instruction specifically to read the MD register. Unfortunately, this instruction can read only the two most-significant bits or MD. There's no way to read the contents of the two least-significant bits, as we would like.

This type of problem occurs frequently when interfacing microprocessors to peripheral hardware (such as parallel output ports): often, some or all memory locations used by the peripheral are write-only. One way that programmers to work around this problem is by having the program remember the most recent value written to the peripheral. The value is remembered by storing a copy of it in some "normal" memory location. Programmers often refer to a memory location used to store a copy of write-only peripheral data as a "register image".

We can use the “register image” technique to improve our subroutine to set the 6309 processor mode. Here’s an improved version:

```
* Change processor to Emulation Mode or Native Mode,
* depending on value in Register A
* A=0  Emulation Mode
* A<>0 Native Mode
*
* Assumes direct page location <D.MDREG contains an
* accurate image of the MD register contents (The
* program must initialize <D.MDREG to $00 at start-up).
*
* Since LDMD accepts only an immediate operand, we
* push the appropriate LDMD / RTS sequence onto the
* stack and call it as a subroutine.
* Works for 6309 only.
SETPMD  PSHS  X,D,CC         ;Save registers
        ORCC  #$50           ;Make operation indivisible
        LDB   <D.MDREG       ;Get mode register image
        ANDB  #$FE           ; strip mode selection bit (Emulation)
        TSTA
        BEQ   SETMD2         ;Skip next part if want Emulation
        ORB   #$01           ;Set Native mode bit (INCB lacks clarity)
SETMD2  STB   <D.MDREG       ;B has right value — update register image
        LDA   #$39           ;RTS op-code
        EXG   B,A            ;Now A = LDMD’s immed. operand, B = RTS
        LDX   #$103D         ;X has LDMD’s 2-byte op-code
        EXG   D,X            ;Now D:X = 10 3D <value> 39
        PSHS  X,D            ;Put subroutine on stack
        JSR   ,S             ;Call subroutine, setting mode
        LEAS  4,S            ; throw away subroutine when done.
        PULS  CC,D,X,PC      ; and return to caller.
```

This subroutine uses several advanced machine language programming techniques. First, it saves the condition code register and explicitly disables interrupts using ORCC; this ensures that no interrupt service routine can change the value of D.MDREG once our subroutine has read it. Second, we use direct page location D.MDREG (you may use any available memory location in your programs) to keep a “register image” of the 6309’s write-only MD register; note that we read D.MDREG to obtain the current value of the MD register, and also that we update D.MDREG to match the new value that we write to the MD register. Finally, we use the system stack to store a custom-built subroutine that loads the MD register with the desired value; this is necessary because LDMD requires the immediate addressing mode, and we wished to avoid self-modifying code in the subroutine.

4.5 Selecting and Using the 6309 FIRQ Mode

Among popular 8-bit microprocessors, only the 6809 family has the unique FIRQ (Fast Interrupt ReQuest) interrupt. FIRQ has its own interrupt vector ($FFF6 vs. $FFF8 for IRQ) and its own interrupt mask bit in the condition code register (CC.F vs. CC.I for IRQ).

On the 6809, FIRQ behaves differently than any other type of interrupt. Most interrupts (NMI, IRQ, SWI, etc.) stack processor registers PC, U, Y, X, DP, B, A, and CC. FIRQ stacks only PC and CC. It takes time to stack registers; since FIRQ stacks fewer registers than any other interrupt, the processor can transfer control to the service routine faster. This makes the 6809's FIRQ ideal for use in high-speed interrupt-driven I/O applications.

The 6309 exactly emulates the 6809's handling of FIRQ, but only when mode bit MD.1 is zero. The processor sets this bit to zero at reset, allowing the 6309 to drop directly into a 6809 system.

When bit MD.1 is one, the 6309 changes the operation of FIRQ. FIRQ retains its separate interrupt vector and condition code register mask bit, but when MD.1 is one each FIRQ stacks the same set of registers as any other interrupt. You might wonder why this is a useful feature, if it just makes the interrupt take longer.

We've already mentioned that FIRQ is well suited for high-speed interrupt-driven I/O. Many computer systems just use FIRQ as another distinct interrupt input, and aren't concerned about its added speed. On these systems, the FIRQ interrupt service routine often begins by stacking some or all of the registers that FIRQ doesn't. These service routines actually use more time manually stacking and unstacking registers, than would be spent if FIRQ automatically stacked all of the registers. Here's an example:

```
* 6809-style FIRQ routine, assuming all registers
* must be preserved.
*
* This routine uses 34 cycles for register stacking
* overhead; the FIRQ itself takes 10 cycles, for a
* total overhead of 44 cycles.
FIRQVC  PSHS U,Y,X,DP,B,A     ;Save registers — 14 cycles
        LDA  >DEVSTS          ;Clear the interrupt source
        ...                   ; (do rest of service here)
        PULS A,B,DP,X,Y,U     ;Restore registers — 14 cycles
        RTI                   ;Restore CC and PC — 6 cycles
```

The PSHS and PULS instructions use 28 machine cycles, and require 4 extra bytes in the interrupt service routine. The 6309's MD.1 mode bit overcomes this problem by changing FIRQ to automatically stack the PC, U, Y, X, DP, B, A, and CC registers. Here's an example of an equivalent modified FIRQ routine for the 6309:

```
* 6309-style FIRQ routine, assuming FIRQ automatically
* preserves all registers.
*
* This routine uses 15 cycles for register stacking
* overhead; the FIRQ itself takes 19 cycles, for a
* total overhead of 34 cycles.
FIRQVC  LDA  >DEVSTS          ;Clear the interrupt source
        ...                   ; (do rest of service here)
        RTI                   ;Restore all registers — 15 cycles
```

In this example, the new FIRQ mode of the 6309 saves 4 bytes and 10 cycles.

You can select either mode of FIRQ handling on the 6309, by using the LDMD instruction to modify bit MD.1. Many of the techniques described in Section 4.4 for the MD.0 bit apply equally to MD.0. For example, the instruction

```
LDMD  #$02
```

forces IRQ-like FIRQ handling and places the processor in Emulation Mode.

If your program is running under an operating system, you may want to leave control of FIRQ handling to the operating system. Changing the contents of the MD register from within an application could confuse the operating system, since the operating system won't be aware of the change.

4.6 Using the W Register in Emulation Mode

All of the 6309's new instructions and registers may be accessed from both Emulation Mode and Native Mode.

Using the W register in Emulation Mode presents a special challenge. Since this register is not stacked during Emulation Mode interrupts, interrupt service routines must take care not to change the value of the W register. Furthermore, time-sliced systems based on the 6309 must take care to save each task's W register before switching to another task; otherwise, switching to a new task that modifies the W register will destroy the W register value expected by the old task when it resumes.

Here's an example of a simple program and interrupt routine, each using the W register.

```
* Main program. Decrements W register until 0, then
* executes an RTS
MAIN    LDW   #1000          ;Delay value
DELAY   DECW                 ; (loop until zero)
        BNE   DELAY
        RTS                  ;Exit
* Interrupt service routine. This routine
* transfers 16 bytes of data from IOADDR to
* the buffer pointed to by >BFADDR
IRQSVC  LDX   #IOADDR
        LDY   >BFADDR
        LDW   #16            ;byte count
        TFM   X,Y+
        RTI                  ;now W is zero
```

In Native Mode, these routines would work completely independently of one another. The Native Mode IRQ would stack and restore the contents of the W register, avoiding any impact of the IRQSVC routine on the MAIN routine.

In Emulation Mode, an interrupt occurring during execution of MAIN's delay loop will extend the delay to at least 64 times its intended length: the interrupt service routine exits with W = 0, causing MAIN's delay loop to execute 65,536 iterations before the DECW instruction sets the Z bit of the condition code register. Even worse, frequent interrupts in this example will make the delay loop last forever, since each new interrupt resets W. This is only one example of the kind of problems that can occur when using the W register in Emulation Mode.

In this example, we've modified the troublesome interrupt service routine to preserve the W register, eliminating the problem:

```
* Main program. Decrements W register until 0, then
* executes an RTS
MAIN    LDW   #1000          ;Delay value
DELAY   DECW                 ; (loop until zero)
        BNE   DELAY
        RTS                  ;Exit
* Revised interrupt service routine. This routine
* transfers 16 bytes of data from IOADDR to the
* buffer pointed to by >BFADDR, preserving W.
IRQSVC  PSHSW                ;save W
        LDX   #IOADDR
        LDY   >BFADDR
```

```
        LDW   #16            ;byte count
        TFM   X,Y+
        PULSW
        RTI                  ;now W is zero
```

This version of the interrupt service routine works correctly for both Native Mode and Emulation Mode.

You don't always have the luxury of modifying interrupt service routines to preserve the W register. In this case, the safest way to use the W register while in Emulation Mode is with interrupts disabled. Even so, the programmer must make sure that the service routine for non-maskable NMI interrupts preserves W. Here's an example of "safe" use of W from Emulation Mode:

```
* Routine to use W "safely" as a parameter
* to the TFM instruction. Since TFM is
* interruptible, we explicitly mask them
* for the entire time that the W register
* contents are important.
BLKCLR  CLR   ,-S            ;Get a ZERO to the stack
        LDY   >BFADDR        ;Address of buffer to clear
        ORCC  #$50           ;No IRQ, no FIRQ (hope NMI is OK!)
        LDW   #2048          ; buffer size is 2K
*   An interrupt that changed W during
*   execution of the TFM would be disastrous!
        TFM   X,Y+           ; clear the buffer
        ANDCC #$AF           ;FIRQ, IRQ OK now.
        LEAS  1,S            ;clean the stack
        RTS
```

The example notes a potential trouble spot; and uses the ORCC instruction to prevent this type of trouble.

4.7 Native Mode Interrupt Processing

One of the most important differences between the 6309's Emulation Mode and Native Mode is the way in which interrupts are processed. In Emulation Mode, the 6309 processes interrupts identically to the 6809, while in Native Mode the 6309 pushes the W register onto the stack (in addition to the standard 6809 registers) during interrupt processing.

Unless a program modifies the behavior of FIRQ by setting mode bit MD.1, the FIRQ interrupt works identically in both Native Mode and Emulation mode. This interrupt pushes the program counter (PC) and the condition code register (CC) to the system stack, and vectors to the address stored at $FFF6. The interrupt service routine is responsible for preserving any registers (including W) that it modifies.

The remaining interrupts (NMI, SWI, IRQ, TRAP) operate slightly differently in each mode.

In Emulation Mode, these interrupts push 12 bytes of registers to the system stack, in the following order:

```
(SP before interrupt) ->                            Stack Offset
                              PC.LO
                              PC.HI                       10
                              U.LO
                              U.HI                        8
                              Y.LO
                              Y.HI                        6
                              X.LO
                              X.HI                        4
                              DP                          3
                              B                           2
                              A                           1
(SP after interrupt) ->       CC                          0
```

Note that the contents of each register are stored at a predetermined offset from the ending value of the stack pointer. For example, examining the byte at (2,S) reveals the contents of register B just before the interrupt, while examining the word at (10,S) reveals the contents of the program counter.

In Native Mode, these interrupts push 14 bytes of registers to the system stack, in the following order:

```
(SP before interrupt) ->                            Stack Offset
                              PC.LO
                              PC.HI                       12
                              U.LO
                              U.HI                        10
                              Y.LO
                              Y.HI                        8
                              X.LO
                              X.HI                        6
                              DP                          5
                              F                           4
                              E                           3
                              B                           2
                              A                           1
```

```
(SP after interrupt) ->       CC                     0
```

Note that the contents of each register are still stored at a predetermined offset from the ending value of the stack pointer, but that the Native Mode offsets for registers DP, X, Y, U, and PC differ from those in Emulation Mode. For example, the word at (10,S) now reveals the contents of register U; we must examine the word at (12,S) to determine the contents of the program counter.

Differences in the interrupt stack offsets for various registers aren't usually important; what matters in most cases is that the contents of all registers are restored to their original values upon completion of interrupt service. This allows the interrupted program to operate without malfunction, even if many interrupts occur.

Two important programming techniques are effected significantly by the register stacking differences between Native Mode and Emulation Mode. These techniques are Service Calls and Interrupt-Terminated I/O.

Service Calls

Some 6809-compatible operating systems (including OS9) use software interrupts to transfer control from a program to an operating system service routine. This technique conceals operating system details (such as specific service subroutine addresses) from the application, making applications more portable and maintainable.

When using this technique, the application accesses an operating system service routine by loading information into 6809 registers (such as D, X, Y, and U) and executing a SWI - type instruction. The corresponding SWI vector points at the operating system service routine, which performs a requested service using the caller's register values (stacked by the SWI instruction) as arguments.

In some cases, the service routine returns information to the application by modifying the values of the stacked registers; this causes the registers to take on different values when application execution resumes. The service routine terminates by executing an RTI instruction, which restores all register values and causes execution to resume at the address indicated by the stacked copy of register PC. Unless the service routine modified the stacked PC, execution resumes at the instruction immediately following the SWI.

Native Mode changes the interrupt stack offsets of registers DP, X, Y, U, and PC. This impacts SWI-type service routines that examine or modify caller-supplied register values. When updating software to work in Native Mode, service routines of this type must be rewritten to either:

- Examine or modify only the caller's CC, A, and B registers, or
- Manually adjust the stack frame to match Emulation Mode, or
- Work only in Native Mode, using the new stack offsets, or
- Dynamically determine which stack offsets to use, by sensing the processor operating mode.

Often, only the 3rd and 4th alternatives are practical. Note that the 2nd alternative does not preserve the W register.

Interrupt-Terminated I/O

A less common technique involves using the NMI (non-maskable interrupt) to terminate a time-critical I/O operation. This eliminates the need for the I/O operation to keep track of the number of bytes transferred; instead, the peripheral hardware generates an NMI when the transfer is complete. This technique is used by the "HALT-type" floppy disk controllers supplied with some models of 6809-based PCs.

When using this technique, the application stores the address of a "completion routine" at an address known to the NMI service routine. The application then enters an infinite data transfer loop, copying data to or from the peripheral. When the transfer is complete, an NMI occurs. The NMI service routine changes the value of the stacked program counter, making it the address of the specified "completion routine". This causes execution to resume at the completion routine, rather than in the infinite data transfer loop, terminating the transfer.

Native Mode changes the interrupt stack offset of the program counter. This impacts interrupt-terminated I/O, since the NMI service routine must change the stacked program counter value. When updating software to work in Native Mode, service routines of this type must be rewritten to either:

- Manually adjust the stack frame to match Emulation Mode, or
- Work only in Native Mode, using the new stack offsets, or
- Dynamically determine which stack offsets to use, by sensing the processor operating mode.

Often, only the 2nd and 3rd alternatives are practical. Note that the 1st alternative does not preserve the W register.

4.8 Using the TRAP Vector

Motorola's 6809 literature describes addresses $FFF0-$FFF1 as a "reserved" interrupt vector. On most existing 6809 systems, the value of the "reserved" vector is either garbage data, $0000, or $FFFF.

On 6309 systems, $FFF0 is known as the TRAP vector. The 6309 uses this vector to point at the service routine for a new type of interrupt, called a "trap". The 6309 provides two kinds of TRAP interrupts: the Illegal Opcode Trap, and the Division By Zero Trap.

Illegal Opcode Trap

Even though the 6309 defines many new instructions, not all combinations of binary data make valid 6309 machine language instructions. We call the invalid combinations "illegal op-codes".

The 6809 has many more illegal op-codes than the 6309. The designers of the 6809 chose to ignore these illegal op-codes, since they never occur in a bug-free program. When the 6809 encounters an illegal op-code, it usually just advances to the next instruction.

The designers of the 6309 took a different approach, reasoning that since illegal op-codes occur only as program bugs, the programmer or user wants to know when an illegal op-code occurs. To accomplish this, every illegal op-code in the 6309 causes a new kind of software interrupt: the Illegal Opcode Trap.

The Illegal Opcode Trap uses the new 6309 TRAP interrupt vector. This vector must point at a valid interrupt service routine to take advantage of the 6309's Illegal Opcode Trap.

When the 6309 encounters an illegal op-code, it attempts to use the TRAP vector as the address of the Illegal Opcode Trap service routine. The processor tries to execute whatever code or data it finds there. If the TRAP vector doesn't point at a valid interrupt service routine, the program that encountered the illegal op-code will malfunction unpredictably.

Programmers can take advantage of the Illegal Opcode Trap, by reprogramming the system's TRAP vector (usually stored in an internal boot ROM) to match the SWI vector. Most 6809-based systems use the SWI software interrupt for debugging. When the debugger is running on these systems, the SWI interrupt service routine handles "breakpoints" created when the debugger replaces the first byte of an instruction with the 1-byte SWI op-code ($3F). Setting the TRAP vector to match the SWI vector effectively causes additional "breakpoints" whenever the processor encounters an illegal op-code. The programmer can distinguish between a legitimate breakpoint and an illegal op-code breakpoint by checking the breakpoint address displayed by the debugger.

Another technique for using the Illegal Opcode Trap is to set the TRAP vector to point at an RTI (Return From Interrupt) instruction. This causes the processor to resume execution at the next instruction after the illegal op-code, effectively skipping over the illegal op-code. Note that this will not make the 6309 handle illegal op-codes identically to the 6809; the number of bytes skipped may differ between the 6809 and the 6309.

A few undocumented, but working, 6809 instructions generate an Illegal Opcode Trap on the 6309. For example, the sequence $16XXXX is the documented coding of the 6809 LBRA instruction; this sequence works identically on each processor. The undocumented sequence $1020XXXX is a working alternative coding of LBRA on the 6809. On the 6309, this sequence generates an Illegal Opcode Trap. Very, very few programs use undocumented 6809 instructions, so the slight difference between processors causes few, if any, problems.

Division By Zero Trap

The 6309 uses the TRAP interrupt vector for a second purpose, unrelated to illegal op-codes. The second use for the TRAP vector is the Division By Zero Trap, which occurs when the divisor of the DIVD or DIVQ instruction is zero.

In this case, the 6309 advances the program counter past the entire division instruction before calling the TRAP interrupt service routine. Division by zero almost always represents a program bug, so setting the TRAP vector to match the SWI vector provides a valuable debugging tool for the Division By Zero Trap interrupt. If your program uses hardware division, be sure to check the divisor for a value of zero before executing DIVD or DIVQ.

TRAP Interrupt Service Routine

Up to this point we've discussed setting the TRAP vector to match the SWI vector, as a valuable debugging tool.

In some cases, the system designer might prefer having a dedicated interrupt service routine for TRAP interrupts. For example, under a 6809 operating system like OS9, SWI software interrupts invoke operating system services (such as file I/O) when the debugger isn't running. If an Illegal Instruction Trap occurs when the debugger isn't running, and the TRAP vector matches the SWI vector, OS9 will treat the illegal op-code as an attempt to access an operating system service. If the illegal op-code happens to match a valid service request, OS9 will process the bogus request — possibly corrupting a file or performing some other undesirable "service".

A dedicated TRAP service routine avoids problems caused by misinterpretation of illegal op-codes. Since the routine only handles TRAP interrupts, it can more effectively recover from them — possibly by aborting the current program as soon as it encounters an illegal op-code.

When writing a dedicated TRAP service routine, the programmer must remember that the processor uses the TRAP vector for both Illegal Opcode Trap and Division By Zero Trap interrupts. The programmer can tell the difference between these two interrupts by examining the 6309's mode (MD) register.

The Illegal Opcode Trap interrupt sets mode bit MD.6, while the Division By Zero Trap sets mode bit MD.7. The interrupt service routine can use the BITMD instruction to determine which type of TRAP has occurred, as shown in the example:

```
* Example of a TRAP interrupt service routine.
* This routine dispatches ILLEGAL OPCODE interrupts
* to a ROM debugger at address DBGENT, while it
* handles DIVISION BY ZERO interrupts by setting the
* carry bit (CC.C) on the stack frame. Note that the
* routine works for both Native and Emulation modes.
SVTRAP  BITMD #%01000000     ;MD.6 non-zero if ILLEGAL OPCODE
        LBNE  >DBGENT        ; so go to ROM debugger
        OIM   #$01,0,S       ;DIVISION BY ZERO; set LSB of stacked CC
        RTI                  ;Return past DIVD or DIVQ, with carry set.
```

The BITMD instruction automatically clears each bit that it tests; in the example, the LDMD instruction clears bit MD.6 after testing it. This the TRAP interrupt service routine from trying to handle an error that has already been detected and serviced.

Once set by a TRAP interrupt, bits MD.6 and MD.7 remain set until explicitly cleared by the LDMD instruction or by RESET.

4.9 New 16-Bit Operations

The 6809, although considered an 8-bit microprocessor, can perform some 16 bit operations. For example, the LDD instruction loads two bytes (16 bits) of data at once, while the LEAX instruction can add a 16-bit offset to the contents of the 16-bit X register.

The 6809 also provides instructions that allow manipulation of 16-bit (or larger) data, one byte at a time. Thus, we can complement a 16-bit value in the D register by executing the sequence COMA / COMB. To negate a 16-bit value, we use the sequence COMA / COMB / ADDD #1; part of this sequence uses 8-bit operations, while part of it uses 16-bit operations.

The old 6809-style sequences still work on the 6309, but the 6309 can often perform equivalent operations using less memory or less time through the use of new 16-bit instructions. Table 4.9 shows several examples of old 6809 sequences and their more efficient 6309 counterparts.

6809 Sequence	6309 Sequence	Notes
PSHS D TFR Y,D LEAX D,X PULS D	ADDR Y,X	16-bit register addition
COMA COMB ADDD #1	NEGD	16-bit negate
ASLB ROLA	ASLD	16-bit shift or logic
SUBD #1	DECD	16-bit decrement
CMPD #0	TSTD	16-bit zero test
ADCB #0 ADCA #0 ADDD 2,X	ADCD 2,X	16-bit addition with carry
<subroutine>	DIVD 2,X	16-bit by 8-bit division

In addition to new 16-bit instructions, the 6309 adds two new 16-bit registers: the W register, and the V register.

The W register is slightly less flexible than the D register when used as an accumulator . For example, you can't perform logical functions like AND, OR, and ASL on register W, but you can use it for load, store and arithmetic operations using the same addressing modes as register D. In many cases, having the W register is like having a second D register to work with.

The W register compensates for its limited accumulator functions by providing limited pointer register functions. Used as a pointer register, W is slightly less flexible than the Y register. For example, you can't use W as a pointer register for certain addressing modes: `LDX ,W+` is not allowed, but `LDX ,W++` is. In many cases, having the W register is like having an additional pointer register to work with.

The V register is useful primarily for register-to-register operations. You can transfer 16-bit values into and out of V, and you can use the V register as the source or destination operand of any register-to-register instruction except TFM. The V register maintains its most recent value even after RESET; the values of other 6809 or 6309 registers are undefined after RESET.

4.10 New 32-Bit Operations

Some writers describe the 6809 as an "8/16-bit processor", basing their description on the 6809's 8-bit data bus and 16-bit internal registers. Both the 6809 and the 6309 can perform 8-bit and 16-bit operations. In addition, the 6309 can perform 32-bit operations using the new Q register.

The Q register is a 32-bit register made up of the 16-bit D register and the 16-bit W register. Only a few 32-bit operations are allowed on register Q: you can load it (LDQ), store it (STQ), divide its contents by a 16-bit value (DIVQ), or multiply together two 16-bit values to obtain a 32-bit result in Q (MULD).

You can use combinations of 16-bit operations to implement more 32-bit operations on the Q register. Here are a few examples:

Sequence of Instructions	32-Bit Result

`EXG  D,W`
`ADDD 2,X`
`EXG  D,W`
	Add 32-bit number to Q

```
ADCD 0,X

CMPD 0,U                                    Unsigned 32-bit compare
BNE  L1
CMPW 2,U
L1 EQU *

COMW                                        Complement Q
COMD

INCW                                        Increment Q
BNE L2
INCD
L2 EQU *
```

You can use any addressing mode with instructions that manipulate the Q register, but some of the 6809 addressing modes are less useful with Q than with other registers. The 6309 doesn't define any new 32-bit addressing modes. For example, the sequence

```
LDQ  ,X++
```

loads the Q register but only advances register X by 2. It would be nice if we could automatically advance by 4 (e.g. LDQ ,X++++), but the 6309 can't do this. The best alternatives for this example are:

```
LDQ  ,X++
LEAX 2,X
```

and

```
LDD  ,X++
LDW  ,X++
```

Each alternative uses 5 bytes of memory. In Emulation Mode, the second alternative is actually faster.

With the exception of the auto-increment load and store operations shown in the previous example, the 6309's 32-bit operations are significantly faster than equivalent combinations of 8-bit or 16-bit operations.

4.11 Using the TFM Block Move Instruction

The 6309's TFM (TransFer Multiple) instruction moves blocks of data from one memory location to another at high speed. TFM is about 4 times faster than the simplest equivalent byte-by-byte data copying loop.

The TFM instruction uses any two 16-bit pointer registers (S, U, Y, X, or D) to point at the source and destination memory locations for the transfer. It uses the W register to count the number of bytes transferred.

The 6309 has four types of TFM instruction. Each type handles a different kind of memory transfer. Before executing any type of TFM, register W contains the number of bytes to be moved; two pointer registers point at the first byte to be moved, and the location to which TFM will move it. After executing TFM, the W register contains zero and the values in the pointer registers depend on the type of TFM that was executed.

Memory-to-Memory Transfer

Two types of TFM allow programmers to rapidly copy memory from one location to another. These types are the Post-Increment TFM and the Post-Decrement TFM.

The Post-Increment TFM copies bytes from one location to another in ascending order. The 6309 decrements the value of the W register, and increments the value of each pointer register, after copying each byte. When the copy is complete, the source pointer register points past the last original byte and the destination pointer register points past the last copy byte.

The Post-Decrement TFM copies bytes from one location to another in descending order. The 6309 decrements the value of the W register, and decrements the value of each pointer register, after copying each byte. When the copy is complete, the source pointer register points before the last original byte and the destination pointer register points before the last copy byte.

Both types of TFM move a block of data from one location to another, but there's a very important reason for having two ways of accomplishing the move. The reason has to do with moving overlapping blocks of data.

Consider a block of 16 bytes of data, containing the numbers 0..15, stored in computer memory at address START. Let's say we want to move this data to address START-1 for some reason. We might use Post-Increment TFM to do it this way:

```
* Example to move 16 bytes from START to START-1
* using Post-Increment TFM.
MOVEIT  LDX  #START          ;Source pointer
        LDU  #START-1        ;Destination pointer
        LDW  #16             ;Byte count
        TFM  X+,U+           ;Post-Increment TFM
        RTS                  ;Return when done.

        RMB  1               ;This is START-1
START   FCB  0,1,2,3,4,5,6,7 ;Data to move
        FCB  8,9,10,11,12,13,14,15
XEND    EQU  *
```

The TFM instruction would begin by loading the value 0 (pointed to by X), storing it at START-1 (pointed to by U). It would then increment X and U, decrement W, and continue with the values 1..15. At the end of the instruction, X will have the value XEND and U will have the value XEND-1; W will have the value 0. Even though the source and destination areas overlap, TFM copies the data correctly.

Next, consider moving the same block of data to address START+1. A programmer's first impulse might be to use Post-Increment TFM again:

```
* Example to move 16 bytes from START to START+1
* using Post-Increment TFM. (A bad idea!)
MOVEIT  LDX  #START          ;Source pointer
        LDU  #START+1        ;Destination pointer
        LDW  #16             ;Byte count
        TFM  X+,U+           ;Post-Increment TFM
        RTS                  ;Return when done.

START   FCB  0,1,2,3,4,5,6,7 ;Data to move
        FCB  8,9,10,11,12,13,14,15
XEND    RMB  1               ; last byte gets moved here.
```

The TFM instruction would again begin by loading the value 0 (pointed to by X), storing it at START+1 (pointed to by U). It would then increment X and U, decrement W, and load the next byte. The next byte happens to be stored at address START+1, which TFM just set to the value 0. TFM loads this byte, stores it at START+2, and so on until 16 bytes have been "moved". When the TFM instruction completes, the results are not at all what the programmer had in mind: the value of the first byte of the source data has been replicated 16 times at the destination. In this case, overlapping source and destination areas led to disastrous Post-Increment TFM results.

The right approach, for this second example, is to use Post-Decrement TFM:

```
* Example to move 16 bytes from START to START+1
* using Post-Decrement TFM.
MOVEIT  LDX  #(START+16-1)   ;Source pointer
        LDU  #(START+16-1)+1 ;Destination pointer
        LDW  #16             ;Byte count
        TFM  X-,U-           ;Post-Decrement TFM
        RTS                  ;Return when done.

START   FCB  0,1,2,3,4,5,6,7 ;Data to move
        FCB  8,9,10,11,12,13,14,15
XEND    RMB  1               ; last byte gets moved here.
```

The TFM instruction would begin by loading the value 15 (pointed to by X), storing it at XEND (pointed to by U). It would then decrement X and U, decrement W, and continue with the values 14..0. At the end of the instruction, X will have the value START-1 and U will have the value START; W will have the value 0. Post-Decrement TFM copies the data correctly.

It's important to realize that while Post-Decrement TFM worked for the second example, it would have failed in the first example. That's why the 6309 provides both Post-Increment TFM and Post-Decrement TFM.

Here's a "rule of thumb" to follow when deciding whether to use Post-Increment TFM or Post-Decrement TFM:

> *If the starting source address is greater than or equal to the starting destination address, use Post-Increment TFM. Otherwise, use Post-Decrement TFM.*

The rule always works, but a programmer can safely choose either type of TFM when the source and destination regions don't overlap. For example, some programmers prefer to use Post-Increment TFM, regardless of the relationship between the starting source and destination addresses, when working on an application in which the source and destination regions never overlap.

Peripheral Data Input

Another type of TFM allows programmers to rapidly read any number of bytes from a single memory address, storing them in consecutive memory locations. This type is Peripheral Input TFM.

Computers based on the 6809 use memory-mapped I/O, meaning that every hardware peripheral (such as a serial or parallel port, and A/D or D/A converter) has a unique memory address. Programs obtain data from a memory-mapped input peripheral by reading one or more of the peripheral's memory locations. Similarly, programs send data to a memory-mapped output peripheral by writing to one or more of the peripheral's memory locations.

Peripheral Input TFM is intended specifically for use with certain kinds of memory-mapped input peripherals. By specifying the "peripheral data" memory address as the source, and the address of a buffer as the destination, a programmer can use Peripheral Input TFM to input entire blocks of peripheral data at high speed.

Some memory-mapped input peripherals require handshaking sequences, in which the program always checks a "peripheral status" location for a predetermined value before reading a new input byte from a "peripheral data" location. Peripheral Input TFM blindly reads data as quickly as possible, and doesn't work for this type of peripheral.

Here's an example using Peripheral Input TFM. It reads 256 bytes of data from the sector buffer of a no-handshake hard disk controller.

```
* Example to read 256 bytes from the HDDATA
* (sector buffer) peripheral input register of a
* no-handshake hard disk controller to the
* buffer at register X
HDREAD  PSHS U,X,CC      ;Save pointers
        LDU  #HDDATA     ;Source pointer
        LDW  #256        ;Byte count
        ORCC #$50        ;No interrupts!
        TFM  U,X+        ;Peripheral input TFM
        PULS CC,X,U,PC   ;Return when done.
```

After executing the TFM instruction, register U has the value HDDATA and register X has its original value plus 256. The example restores X and U to their original values before returning to the caller.

This example assumes that the hard disk controller can transfer a new data byte from the sector buffer every 3 clock cycles (the rate at which TFM copies bytes). Programmers should always make sure that the peripheral can operate at TFM's transfer rate, and that the transfer is handshake-free, before using Peripheral Input TFM.

Note that the example disables interrupts before executing the Peripheral Input TFM instruction. This is necessary, for Peripheral Input TFM only, because of the way the TFM instruction continues execution after an interrupt. An interrupt occurring during execution of the TFM instruction causes TFM to "forget" the last byte read from the peripheral. TFM "recovers" this byte after returning from the interrupt service routine, by re-reading the peripheral. Since the peripheral doesn't know that an interrupt occurred, it outputs the "next" data byte rather than the "forgotten" data byte. This causes two problems: TFM loses one byte of data, and the peripheral's internal counter or pointer is off by one byte. To avoid these problems, always disable interrupts before executing Peripheral Input TFM (and be sure to restore interrupts to their original state when done).

Peripheral Data Output

The fourth type of TFM allows programmers to rapidly write any number of bytes to a single memory address, reading them from consecutive memory locations. This type is Peripheral Output TFM.

Peripheral Output TFM is intended specifically for use with certain kinds of memory-mapped output peripherals. By specifying the "peripheral data" memory address as the destination, and the address of a buffer as the source, a programmer can use Peripheral Output TFM to output entire blocks of peripheral data at high speed.

Some memory-mapped output peripherals require handshaking sequences, in which the program always checks a "peripheral status" location for a predetermined value before writing a new output byte to a "peripheral data" location. Peripheral Output TFM blindly writes data as quickly as possible, and doesn't work for this type of peripheral.

This example uses Peripheral Output TFM to write 256 bytes of data to the sector buffer of a no-handshake hard disk controller.

```
* Example to write 256 bytes to the HDDATA
* (sector buffer) peripheral input register of a
* no-handshake hard disk controller from the
* buffer at register X
HDWRIT  PSHS U,X         ;Save pointers
        LDU  #HDDATA     ;Destination pointer
        LDW  #256        ;Byte count
        TFM  X+,U        ;Peripheral output TFM
        PULS X,U,PC      ;Return when done.
```

After executing the TFM instruction, register U has the value HDDATA and register X has its original value plus 256. The example restores X and U to their original values before returning to the caller.

Like the previous example, this example assumes that the hard disk controller can operate without handshake at the data transfer rate demanded by TFM.

Interrupts and TFM

Each type of TFM instruction performs an indivisible transfer. If any interrupts (including NMI) occur during execution of a TFM instruction, the 6309 will not call the appropriate interrupt service routine until the entire transfer is complete.

Large block transfers using TFM can take as long as 200ms (at 1 MHz E-clock speed). In contrast, many types of video control hardware generate a “vertical retrace” interrupt every 17ms. If a program disables interrupts during execution of a long TFM instruction, the computer can easily miss one or more video (or disk I/O, or other time-critical) interrupts, leading to problems that range from annoying (display flicker) to severe (unreliable disk operation).

If you have to disable interrupts during a block move (for example, when using Peripheral Input TFM), it’s best to break up the block move into smaller pieces, as shown in this example:

```
* Example to move 4096 bytes from (X) to (Y)
* using Peripheral Input TFM and a loop.
MOVEIT  PSHS Y,X,B           ;Save register values
        LDB  #(4096/256)     ;Number of loops
MOVE2   PSHS CC
        ORCC #$50            ;No interrupts
        LDW  #256            ;Byte count per loop
        TFM  X+,Y+           ;Post-Increment TFM (< 1ms at 1MHz)
        PULS CC              ;Enable interrupts!
        DECB                 ;Decrement loop counter
        BNE  MOVE2
        PULS B,X,Y,PC        ;Return when done.
```

Even though the example uses a loop to execute several TFM instructions, it’s still much faster than a byte-by-byte copy loop.

Other Uses of TFM

Both Post-Increment TFM and Peripheral Input TFM can clear or initialize a block of memory. Suppose we want to clear 256 bytes of memory at (X). We might do it this way with Peripheral Input TFM:

```
* Example to clear 256 bytes at (X)
* using Peripheral Input TFM.
* This example uses 15 bytes.
MCLEAR  PSHS X              ;Save register values
        CLR  ,-S            ;ZERO on stack
        LDW  #256           ;Byte count
        TFM  S,Y+           ;Copy 256 Zeros to (X)
        LEAS 1,S            ;Clean up stack
        PULS X,PC           ;Return when done.
```

A less obvious solution takes advantage of a "problem" we described earlier, in which Post-Increment TFM will duplicate overlapping data if the destination address is higher than the source address:

```
* Example to clear 256 bytes at (X)
* using Post-Increment TFM.
* This example also uses 15 bytes.
MCLEAR2 PSHS Y,X            ;Save register values
        LEAY 1,X            ;Duplicate pointer + 1
        CLR  ,X             ;ZERO 1st byte
        LDW  #255           ;Byte count - 1
        TFM  X+,Y+          ;Duplicate 255 Zeros to (Y)
        PULS X,Y,PC         ;Return when done.
```

Each of these examples takes about the same time to execute.

Sometimes a program needs to initialize a number of locations of the stack with predetermined constant values. This might occur, for instance, at the beginning of a subroutine compiled from a high-level language. The Post-Decrement TFM instruction can rapidly initialize stack variables, often using less memory than equivalent load and push operations, as shown in this example:

```
* Example showing 16 bytes of stack data
* initialized using Post-Decrement TFM.
SAMPLE  LEAX IDATA+15,PCR   ;Point at stack initializers
        LDW  #16            ;Byte count
        LEAS -1,S           ;Adjust SP to point at 1st byte pushed
        TFM  X-,S-          ;Post-Decrement TFM
        LEAS 1,S            ;Adjust SP to point at last byte pushed
        ...                 ;Rest of routine goes here.
```

```
IDATA   FCB  0,1,2,3,4,5,6,7 ;Data to initialize stack with
        FCB  8,9,10,11,12,13,14,15
```

In this example, it's important to understand the LEAS -1,S and LEAS 1,S instructions that bracket the TFM instruction. We need these extra instructions because the 6309's stack pointer always points at the most recently pushed item of data. The first LEAS adjusts the stack pointer to point at the 1st location to be initialized by TFM, while the second LEAS adjusts the stack pointer to point at the last byte initialized by TFM.

You might suspect that Post-Increment TFM could be useful to initialize a stack. This turns out to be incorrect. If we had used Post-Increment TFM, an interrupt occurring just after we initialized the stack would overwrite the initialized data. Even if we disabled interrupts during initialization, an NMI could still corrupt the stack. Here's an example of what NOT to do:

```
* Example showing 16 bytes of stack data
* initialized (unwisely) using Post-Increment TFM.
SAMPLE  LEAX IDATA,PCR       ;Point at stack initializers
        LDW  #16             ;Byte count
        LEAS -16,S           ;Adjust SP, then copy upwards
        TFM  X+,S+           ;Post-Increment TFM
*** NMI or other interrupt here overwrites initialized stack!
        LEAS -16,S           ;Move SP back to 1st byte pushed
        ...                  ;Rest of routine goes here.

IDATA   FCB  0,1,2,3,4,5,6,7 ;Data to initialize stack with
        FCB  8,9,10,11,12,13,14,15
```

4.12 Hardware Multiplication and Division

On the 6809, multiplication and division took a long time — even with the 6809's 8-bit MUL instruction. The next example shows a 6809 routine to multiply two 16-bit unsigned numbers, one in the D register, the other at memory location (0,X):

```
* 16 x 16 unsigned multiplication using 6809
* MUL instruction. Multiplies D by (0,X).
* Returns 32-bit result in D:X
MUL16   CLR  ,-S        ;temp. results
        CLR  ,-S
        CLR  ,-S
        CLR  ,-S
        PSHS D
        LDA  1,X
        MUL             ;D lo byte * (X) lo byte
        STD  2+2,S      ; <61 cycles so far>
```

```
        LDA  1,S
        LDB  0,X
        MUL             ;D lo byte * (X) hi byte
        ADDD 2+1,S      ; (accumulate result)
        STD  2+1,S
        LDA  2+0,S
        ADCA #0
        STA  2+0,S      ; <46 more cycles>
        LDA  0,S
        LDB  1,X
        MUL             ;D hi byte * (X) lo byte
        ADDD 2+1,S      ; (accumulate result)
        STD  2+1,S
        LDA  2+0,S
        ADCA #0
        STA  2+0,S      ; <46 more cycles>
        LDA  0,S
        LDB  0,X
        MUL             ;D hi byte * (X) hi byte
        ADDD 2+0,S      ; (accumulate result)
        STD  2+0,S      ; <34 more cycles>
        LEAS 2,S        ;discard old value of D
        PULS D,X,PC     ;Exit with result in D:X <16 more>
```

The subroutine takes about 203 cycles to execute. Thanks to the 6309's MULD instruction, similar (but signed) multiplication code looks like this:

```
* 16 x 16 signed multiplication using 6309
* MULD instruction. Multiplies D by (0,X).
* Returns 32-bit result in D:X
MUL16   MULD 0,X        ;results to D:W
        TFR  W,X        ; now results in D:X
        RTS
```

This version takes only 43 cycles, and uses only 6 bytes of memory.

The 6309 can also perform division in hardware. The DIVD instruction divides the 16-bit number in D by a specified 8-bit operand, while the DIVQ instruction divides the 32-bit number in Q by a specified 16-bit operand. Each division operation produces both a quotient (in W) and a remainder (in D).

4.13 Using Bit Manipulation Instructions

Several new 6309 instructions simplify operations involving individual bits of registers or memory locations. Programs sometimes store and manipulate data one bit at a time, to conserve memory. Peripheral devices often provide up to eight different control functions using individual bits of a single memory location.

The 6309's new bit manipulation instructions include:

Mnemonic	Description
OIM	OR with immediate data (set several bits)
AIM	AND with immediate data (clear several bits)
EIM	Exclusive-OR with immediate data (invert bits)
TIM	Test bits using immediate data (test several bits)
ORR	OR register-to-register (set several bits)
ANDR	AND register-to-register (clear several bits)
EORR	Exclusive-OR register-to-register (invert bits)
BAND	AND memory bit to register bit (single bits)
BIAND	AND inverse memory bit to register bit (single bits)
BOR	OR memory bit to register bit (single bits)
BIOR	OR inverse memory bit to register bit (single bits)
BEOR	Exclusive-OR memory bit to register bit (single bits)
BIEOR	Exclusive-OR inverse memory bit to register bit
LDBT	Load memory bit to register bit (single bits)
STBT	Store register bit to memory bit (single bits)
BITMD	Test bits of MD (mode) register

These 16 new instructions fall into four basic categories based on the kind of data they manipulate: in-place instructions, register instructions, single-bit instructions, and special instructions.

In-Place Instructions

The in-place bit manipulation instructions (OIM, AIM, EIM, TIM) allow a programmer to manipulate individual bits of data stored in memory, without using a register. Each of these instructions includes 8 bits of immediate data, which are used as a bit mask for the specified memory location.

To set one or more bits in memory, use the OIM instruction with an immediate operand having set bits in the to-be-set bit positions of the memory location. For example, the instruction:

```
OIM #%10010011,3,X
```

sets bit positions 7, 4, 1, and 0 of the data at memory location (3,X). The remaining bits of the memory location are not changed.

To clear one or more bits in memory, use the AIM instruction with an immediate operand having clear bits in the to-be-cleared bit positions of the memory location. For example, the instruction:

```
AIM #%10010011,3,X
```

clears bit positions 6, 5, 3 and 2 of the data at memory location (3,X). The remaining bits of the memory location are not changed.

To invert one or more bits in memory, use the EIM instruction with an immediate operand having set bits in the to-be-inverted bit positions of the memory location. For example, the instruction:

```
EIM #%10010011,3,X
```

inverts bit positions 7, 4, 1, and 0 of the data at memory location (3,X). The remaining bits of the memory location are not changed.

You can use direct, indexed, or extended addressing to specify the in-place memory operand of these instructions, but the bit mask must always be immediate data.

The last instruction in this group, TIM, allows the program to test a memory location for one or more set bits. To test for set bits, use the TIM instruction with an immediate operand having set bits in the to-be-tested bit positions of the memory location. The instruction will set flag CC.Z if none of the specified bits are set, and will clear flag CC.Z if any of the specified bits are set. For example, the instruction:

```
TIM #%10010011,>IOPORT
```

tests for a set bit in position 7, 4, 1, or 0 of the data at memory location >IOPORT. The instruction doesn't change any bits of the memory location.

OIM, AIM, and EIM each execute an indivisible read-modify-write operation on memory. This means that within a single instruction, the 6309 reads data from memory, processes it, and writes modified data to the same location in memory.

Register Instructions

Sometimes a program needs to perform bit-wise logical operations on data stored in 16-bit registers. The 6309 provides the instructions ORR, ANDR, and EORR for this purpose.

These instructions operate on the bits of one register, using data stored in the other register. For example, the instruction:

```
ORR  X,Y
```

ORs the 16-bit data in the Y register with the 16-bit data in the X register, placing the result in Y. We can think of this as setting any bits in Y that are already set in X. Similarly, the instruction:

```
ANDR D,X
```

clears any bits in X that are already cleared in D.

Single-Bit Instructions

The 6309 has eight single-bit manipulation instructions. These instructions update a single bit of a specified register, based on a single bit of a specified direct page memory location, or program a single bit of a specified direct page memory location, based on a single bit of a specified register.

The first three single-bit instructions provide logical AND, OR, and Exclusive-OR operations (BAND, BOR, BEOR) for 1-bit data types. Three more instructions (BIAND, BIOR, BIEOR) invert the bit read from direct page memory before performing the logical operation. The LDBT instruction loads a single bit of data from direct page memory, while the STBT instruction stores a single bit of data to direct page memory. Logic “truth tables” for these instructions are shown below.

BAND Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**0**
0	1	**0**
1	0	**0**
1	1	**1**

BIAND Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**0**
0	1	**0**
1	0	**1**
1	1	**0**

BOR Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**1**

BIOR Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**1**

BEOR Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

BIEOR Instruction

Register Bit Value	Memory Bit Value	Result Value
0	0	**1**
0	1	**0**
1	0	**0**
1	1	**1**

LDBT Instruction

Register Bit Value	Memory Bit Value	Result Value
-	0	**0**
-	1	**1**
-	0	**0**
-	1	**1**

STBT Instruction

Register Bit Value	Memory Bit Value	Stored Value
0	-	**0**
0	-	**0**
1	-	**1**
1	-	**1**

Single-bit instructions can access data anywhere in memory, if the program has previously stored the most-significant 8 bits of the desired memory address in the DP register. For example, this piece of code accesses the variable FLAGS using single-bit instructions, regardless of where it is located in memory:

```
* Example to use single-bit instructions
* anywhere in memory. This example sets
* carry based on bit 4 of variable FLAGS.
        PSHS A,DP          ;Save DP
        LDA  #(FLAGS/256)  ;Set up new DP register
        TFR  A,DP
        LDBT CC.C,<FLAGS.4 ;Load CC.C from FLAGS.4
        PULS DP,A
```

Because of the overhead involved in setting up the DP register, you may want to use the above technique only when you have several bit manipulation operations to perform in the same 256-byte memory block.

The single-bit instructions can be useful when manipulating peripheral devices, which occupy memory locations $FFXX on many systems. After storing the value $FF in the DP register, the single-bit instructions can access any location in the range $FF00—$FFFF.

Be careful when using single-bit instructions to access peripherals. Each of the single-bit instructions (including STBT) reads an entire direct page memory location, and not just one bit, when executed. Many peripheral devices provide multiple status flags in a single memory location. Reading that location may automatically clear all set flags. If you use an instruction like LDBT to check on a single status flag, it reads the entire status byte — clearing other set flags before your program has a chance to detect them. This could cause lost data or other program malfunctions.

Here's an example that uses single-bit instructions in a different way, to initialize a peripheral device from a data byte:

```
* Example to use single-bit instructions to initialize
* a peripheral from a data byte. Each bit position of
* the data byte represents a pair of peripheral addresses.
* If the bit is clear, we initialize the peripheral by
* writing to the lower address. If the bit is set, we
* initialize the peripheral by writing to the higher
* address.
*
* Some types of video peripherals use similar
* initialization sequences.
*
* Enter with value in A, peripheral at VIDADR.
* Uses direct page variable TEMP as scratch.
VCINIT  STA  <TEMP
        LDD  #$0800        ;Byte count, offset
        LDX  #VIDADR       ;Note: must be even!
VCLOOP  LDBT B.0,<TEMP.0   ;Make B #0 or #1
        STA  B,X           ;Write anything to correct address
        LSR  <TEMP         ;Line up next TEMP.0
        LEAX 2,X           ;Line up next peripheral address
        DECA
        BNE  VCLOOP        ;Loop for each of 8 bits.
        RTS
```

Single-bit instructions can quickly move groups of bits data from one location to another . This example copies data from one 2-bit field (beginning at bit 3 of location TEMP1) to a new field (beginning at bit 5 of location TEMP2):

```
* Example to use 6309 single-bit instructions to
* move a 2-bit field from TEMP1 to TEMP2. Note that
* no registers are changed, and that the field
* starts at a different bit position in each TEMP.
MOVFLD  LDBT CC.C,<TEMP1.3  ;1st bit to carry
        STBT CC.C,<TEMP2.5  ; carry to destination.
        LDBT CC.C,<TEMP1.4  ;2nd bit to carry
        STBT CC.C,<TEMP2.6  ; carry to destination.
        RTS

* Here's the same thing, in old 6809 instructions.
MOVFLD  PSHS A
        LDA  <TEMP2
        ANDA #%10011111     ;Strip old field value (use AIM on 6309)
        STA  <TEMP2
        LDA  <TEMP1
        ANDA #%00011000     ;Get just field being copied
        ASLA                ;Align to destination bit position
        ASLA                ; (shift bit 3 to bit 5, etc.).
        ORA  <TEMP2         ;OR copied data into TEMP1
        STA  <TEMP2
        PULS A,PC
```

The version using 6309 single-bit instructions is slightly shorter, and much easier to understand, than the equivalent 6809 version.

So far, the examples in this section have used LDBT and STBT. You can use any of the other single-bit instructions similarly. This example ORs together the two fields used in the previous example:

```
* Example to use 6309 single-bit instructions to
* move a 2-bit field from TEMP1 to TEMP2. Note that
* no registers are changed, and that the field
* starts at a different bit position in each TEMP.
ORBFLD  LDBT CC.C,<TEMP1.3  ;1st bit to carry
        BOR  CC.C,<TEMP2.5  ; OR in current value
        STBT CC.C,<TEMP2.5  ; carry to destination.
        LDBT CC.C,<TEMP1.4  ;2nd bit to carry
        BOR  CC.C,<TEMP2.6  ; OR in current value
        STBT CC.C,<TEMP2.6  ; carry to destination.
        RTS

* Here's the same thing, in old 6809 instructions.
ORBFLD  PSHS A
        LDA  <TEMP1
```

```
ANDA #%00011000     ;Get just field being copied
ASLA                ;Align to destination bit position
ASLA                ; (shift bit 3 to bit 5, etc.).
ORA  <TEMP2         ;OR copied data into TEMP1
STA  <TEMP2
PULS A,PC
```

In this case, the version using 6309 single-bit instructions is longer, but still easier to understand, than the equivalent 6809 version.

Special Instructions

The special BITMD instruction is the only way provided for a 6309 program to read the mode (MD) register.

The MD register contains two flags: the Illegal Opcode Trap Flag (MD.6), and the Division By Zero Trap Flag (MD.7). BITMD checks these flags, either one at a time or together. BITMD can’t check any other bits in the MD register.

When BITMD accesses the MD register, it automatically clears any flag bits that it checks. It’s best to check one bit at a time - checking two flags with a single BITMD instruction will clear both flags, without telling you which one was set.

Most programs use BITMD only in TRAP interrupt service routines.

4.14 Capabilities of the D and W Registers

Although the D and W registers are both general-purpose accumulators, the 6309 allows more operations on the D register than on the W register.

This is also true of the individual 8-bit registers that make up W and D. The 6309 has more instructions to manipulate the A and B registers, than it has to manipulate the E and F registers. Very few instructions can manipulate the Q register, which is made up of A, B, E, and F.

The Table summarizes major instruction types of the 6309, and indicates which registers those instructions can manipulate.

Register

Mnem.	Description	Q	D	W	A	B	E	F
LD	load	X	X	X	X	X	X	X
ST	store	X	X	X	X	X	X	X
INC,DEC	add / sub 1		X	X	X	X	X	X
TST	zero test		X	X	X	X	X	X
COM	complement		X	X	X	X	X	X
CLR	zero set		X	X	X	X	X	X
ADD	add		X	X	X	X	X	X
SUB	subtract		X	X	X	X	X	X
CMP	compare		X	X	X	X	X	X
SBC	sub. w/ carry		X	+	X	X	+	+
AND,BIT	bit-wise AND		X	+	X	X	+	+
EOR	bit-wise XOR		X	+	X	X	+	+
ADC	add w/ carry		X	+	X	X	+	+
OR	bit-wise OR		X	+	X	X	+	+
LSR	rt. shift		X	X	X	X		
ROR,ROL	rotate		X	X	X	X		
ASR,ASL	arith. shift		X		X	X		
NEG	negate		X		X	X		

Each “X” indicates an instruction that works in all of the standard addressing modes for the given register; each “+” indicates an instruction that works only in register-to-register mode. Blank spaces indicate that the instruction cannot be used to manipulate the given register.

The 6309 supports all of the above instructions for registers A, B, and D. You can usually use the EXG instruction, in conjunction with another instruction, to effectively perform that instruction on register W, E, or F. Here’s an example that performs the equivalent of an exclusive-OR into the W register:

```
* Perform equivalent of EORW #$1234.
FAKEOR  EXG  W,D           ;swap registers
        EORD #$1234        ;perform operation on D
        EXG  D,W           ;swap registers again
        RTS
```

Note that the order of registers doesn’t matter for the EXG instruction; EXG D,W performs exactly the same operation as EXG W,D, even though the 6309 encodes the two instructions differently. The example uses two different instructions to emphasize to a human reader that W is the register of interest.

The W register is also a pointer register, and can be used with many of the 6309 indexed addressing modes. The Table summarizes major addressing modes of the 6309, and indicates which addressing modes can be used with the W, E, and F registers.

Mnem.	Description	Register Y	W	E	F
,r	No offset	X	X		
n5,r	5-bit fixed offset	X	X		
n8,r	8-bit fixed offset	X	X		
n16,r	16-bit fixed offset	X	X		
r8,r	8-bit register offset	X		+	+
r16,r	16-bit register offset	X	+		
,r+	Post-increment by 1	X			
,r++	Post-increment by 2	X	X		
,-r	Pre-decrement by 1	X			
,--r	Pre-decrement by 2	X	X		
n8,PCR	8-bit PC-relative				
n16,PCR	16-bit PC-relative				

Mnem.	Description	Register Y	W	E	F
[,r]	Indirect, No offset	X	X		
[n8,r]	Indirect, 8-bit fixed offset	X			
[n16,r]	Indirect, 16-bit fixed offset	X	X		
[r8,r]	Indirect, 8-bit register offset	X		+	+
[r16,r]	Indirect, 16-bit register offset	X	+		
[,r++]	Indirect, Post-increment by 2	X	X		
[,--r]	Indirect, Pre-decrement by 2	X	X		
[n8,PCR]	Indirect, 8-bit PC-relative				
[n16,PCR]	Indirect, 16-bit PC-relative				
[n16]	Indirect, Extended				

The “Y” column shows addressing modes usable with the Y register (a full-fledged pointer register) for comparison. Each “X” indicates an addressing mode in which the specified register can be used as a pointer; each “+” indicates an addressing mode in which the specified register can be used as an offset. Blank spaces indicate that the specified register cannot be used with the addressing mode. Note that the W register can be used with many of the same addressing modes as the Y register.

4.15 Uses for the V Register

The 6309’s V register can hold any 16-bit value. Very few instructions can manipulate the contents of the V register, but this register has several important uses.

The V register has one special feature: any value in the V register stays there, even when the computer is reset. Only turning off the computer erases the contents of the V register. Furthermore, the V register is automatically initialized to all 1’s ($FFFF) at power-up. Programs can use the value in the V register for “reset protection”: by checking the value of

V in the RESET routine, a program can determine whether to preserve in-memory data structures:

```
* RESET routine.
*
* If this is initial power-up, initialize all peripherals and memory.
* If this is a RESET, just initialize the peripherals
RESET   JSR  PINIT          ;Initialize peripherals
        TFR  V,X            ;Check V register for $FFFF
        CMPX #$FFFF
        BNE  WARMST         ;If not $FFFF, this is just a reset
* Here, we know it's a power-up. Init memory and store "reset" flag
* value in V
COLDST  JSR  MINIT          ;Initialize memory
        LDD  #$1234         ; Set V register (any value but $FFFF)
        TFR  D,V
WARMST  JMP >PROGRM         ;RESET complete. Go do the program!
```

On the 6809, programmers often performed a similar reset protection test by storing a special value in a predetermined memory location, instead of the V register. This older technique works most of the time, but there's always a chance that the predetermined memory location will accidentally already have the special value at power-up. This causes the program to skip power-up memory initialization, leading to program malfunctions. Using the 6309 V register as described above, there's no uncertainty.

Reset protection is one of the most effective uses for the V register, but it has other uses too. When using the V register for both reset protection and other purposes, you must be careful not to leave a value of $FFFF in the register accidentally. The easiest solution is to use V either for reset protection, or as a useful calculation register, but not both.

A second use for the V register is to hold a frequently used value. You can then use the 6309's register-to-register instructions to quickly load this value or use it in a calculation. The value can be a constant, or some number that varies from time to time, but is used in a lot of calculations. After storing a value in V, you can load it into any register using the 2-byte TFR instruction; equivalent 6809 register loading instructions require 3-4 bytes if the number is constant, or 2-4 bytes and a memory location if the number varies.

```
* Example: loading another register from V

* Start by initializing V early in the program
        LDD  #$1234
        TFR  D,V
        ....
```

```
* Much later in the program, we need the value $1234 in a
* lot of places. Two of the places might look like this:
        TFR  V,X            ;Get $1234 to X
        ...
        ADDR V,U            ;Add $1234 to U
        ...
```

The 6309 doesn't stack the V register during interrupts, and (unlike the W register) there's no way to directly push the V register to the stack. This means that in a multi-tasking environment, several different programs using the V register at the same time are likely to corrupt each other's V register value. You may want to limit or carefully control your use of the V register. Remember, in a multi-tasking environment any other task (including ones beyond your control) could change the value of your V register unless the operating system preserves it.

4.16 Register-to-Register Operations

While most of the 6309's instructions manipulate a single register, or a register and a memory location, several of the new instructions manipulate two registers at once. These are called register-to-register instructions.

Some of these instructions are present in both the 6309 and the 6809. The register-to-register instructions common to both processors are most useful for address calculations. For example,

```
LEAX D,X
```

adds the contents of the D register to the contents of the X register (usually, D is used as an offset into a data structure that X points at). The two instructions below each set the Y register equal to the U register:

```
TFR  U,Y       ;Transfer U to Y (Y <- U)

LEAY ,U        ;Load U with “effective address” Y
```

Only a few instructions like this exist on the 6809. The 6309 adds many new ones, as shown in the Table.

		Processor Type	
Mnem.	Description	6809	6309
TFR	transfer	X	X
EXG	exchange	X	X

```
LEA      calculate address    X        X
ABX      add B to X           X        X
ADDR     add registers                 X
ADCR     add w/ carry                  X
SUBR     subtract regs.                X
SBCR     subtract w/ carry             X
ANDR     bit-wise AND                  X
ORR      bit-wise OR                   X
EORR     bit-wise XOR                  X
CMPR     compare registers             X
```

Each of the register-to-register instructions (except ABX) takes any two register names as arguments.

Most 6809 programs use the D register for arithmetic, with other registers (X, Y, U) for temporary storage and as pointers. These programs often copy other registers into D for manipulation, and then copy the result back to the original register. For example, a 6809 program that needs to add the X and Y registers might use this sequence:

```
EXG  Y,D          ;Get Y into D (stash D in Y)
LEAX D,Y          ;Add Y to X
EXG  Y,D          ;Put back Y and D as they were
```

A 6809 program that needs to subtract Y from D has a more complex task:

```
EXG  X,D          ;Get X into D (stash D in X)
PSHS Y            ;Copy Y to where we can use it on stack
SUBD ,S++         ;Now D = X+Y, stack is clean
EXG  X,D          ;Put back X and D as they were
```

The 6309's register-to-register instructions simplify each of these operations:

```
ADDR Y,X          ;Add Y to X

SUBR Y,X          ;Subtract Y from X
```

Register-to-register instructions make programs that perform arithmetic on registers both smaller and faster.

4.17 Application Summary

In this Chapter, we've examined most of the 6309's new programming features. Most of the new features use familiar instruction mnemonics and work with all of the 6809's addressing modes.

Some of the new instructions add new addressing modes (e.g. single-bit instructions), while others greatly expand the use of an existing mode (e.g. register-to-register instructions).

Because the instructions were added to the 6809's basic set, and because the designers of the 6309 needed a drop-in replacement for the 6809, some of the new features (TFM, the W and V registers) aren't fully integrated with the 6809's interrupt system. These features greatly enhance the 6809, but must be used with the proper precautions to avoid missed interrupts or data corruption.

The new instructions of the 6309 provide a basis for building 6809-family programs that are smaller and faster than ever before. We hope the information in this Chapter has given you some ideas and guidelines for programming the 6309.

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
00	NEG	DIRECT	2	6	5	20	BRA	RELATIVE	2	3	3
01	OIM	DIRECT	3	6	6	21	BRN	RELATIVE	2	3	3
02	AIM	DIRECT	3	6	6	22	BHI	RELATIVE	2	3	3
03	COM	DIRECT	2	6	5	23	BLS	RELATIVE	2	3	3
04	LSR	DIRECT	2	6	5	24	BHS/BCC	RELATIVE	2	3	3
05	EIM	DIRECT	3	6	6	25	BLO/BCS	RELATIVE	2	3	3
06	ROR	DIRECT	2	6	5	26	BNE	RELATIVE	2	3	3
07	ASR	DIRECT	2	6	5	27	BEQ	RELATIVE	2	3	3
08	ASL/LSL	DIRECT	2	6	5	28	BVC	RELATIVE	2	3	3
09	ROL	DIRECT	2	6	5	29	BVS	RELATIVE	2	3	3
0A	DEC	DIRECT	2	6	5	2A	BPL	RELATIVE	2	3	3
0B	TIM	DIRECT	3	6	6	2B	BMI	RELATIVE	2	3	3
0C	INC	DIRECT	2	6	5	2C	BGE	RELATIVE	2	3	3
0D	TST	DIRECT	2	6	4	2D	BLT	RELATIVE	2	3	3
0E	JMP	DIRECT	2	3	2	2E	BGT	RELATIVE	2	3	3
0F	CLR	DIRECT	2	6	5	2F	BLE	RELATIVE	2	3	3
10	(PREBYTE)	---	---	---	---	30	LEAX	INDEXED	2+	4+	4+
11	(PREBYTE)	---	---	---	---	31	LEAY	INDEXED	2+	4+	4+
12	NOP	INHERENT	1	2	1	32	LEAS	INDEXED	2+	4+	4+
13	SYNC	INHERENT	1	>=4	>=4	33	LEAU	INDEXED	2+	4+	4+
14	SEXW	INHERENT	1	2	1	34	PSHS	IMMEDIATE	2	5+	4+
15	---	---	---	---	---	35	PULS	IMMEDIATE	2	5+	4+
16	LBRA	RELATIVE	3	5	4	36	PSHU	IMMEDIATE	2	5+	4+
17	LBSR	RELATIVE	3	9	7	37	PULU	IMMEDIATE	2	5+	4+
18	---	---	---	---	---	38	---	---	---	---	---
19	DAA	INHERENT	1	2	1	39	RTS	INHERENT	1	5	4
1A	ORCC	IMMEDIATE	2	3	2	3A	ABX	INHERENT	1	3	1
1B	---	--	---	---	---	3B	RTI	INHERENT	1	6/15	6/17
1C	ANDCC	IMMEDIATE	2	3	3	3C	CWAI	INHERENT	2	>=20	>=22
1D	SEX	INHERENT	1	2	1	3D	MUL	INHERENT	1	11	10
1E	EXG	REGISTER	2	8	5	3E	---	---	---	---	---
1F	TFR	REGISTER	2	6	4	3F	SWI	INHERENT	1	19	21

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
40	NEGA	INHERENT	1	2	1	60	NEG	INDEXED	2+	6+	6+
41	---	---	---	---	---	61	OIM	INDEXED	3+	7+	7+
42	---	---	---	---	---	62	AIM	INDEXED	3+	7+	7+
43	COMA	INHERENT	1	2	1	63	COM	INDEXED	2+	6+	6+
44	LSRA	INHERENT	1	2	1	64	LSR	INDEXED	2+	6+	6+
45	---	---	---	---	---	65	EIM	INDEXED	3+	6+	6+
46	RORA	INHERENT	1	2	1	66	ROR	INDEXED	2+	6+	6+
47	ASRA	INHERENT	1	2	1	67	ASR	INDEXED	2+	6+	6+
48	ASLA/LSLA	INHERENT	1	2	1	68	ASL/LSL	INDEXED	2+	6+	6+
49	ROLA	INHERENT	1	2	1	69	ROL	INDEXED	2+	6+	6+
4A	DECA	INHERENT	1	2	1	6A	DEC	INDEXED	2+	6+	6+
4B	---	---	---	---	---	6B	TIM	INDEXED	3+	7+	7+
4C	INCA	INHERENT	1	2	1	6C	INC	INDEXED	2+	6+	6+
4D	TSTA	INHERENT	1	2	1	6D	TST	INDEXED	2+	6+	5+
4E	---	---	---	---	---	6E	JMP	INDEXED	2+	3+	3+
4F	CLRA	INHERENT	1	2	1	6F	CLR	INDEXED	2+	6+	6+
50	NEGB	INHERENT	1	2	1	70	NEG	EXTENDED	3	7	6
51	---	---	---	---	---	71	OIM	EXTENDED	4	7	7
52	---	---	---	---	---	72	AIM	EXTENDED	4	7	7
53	COMB	INHERENT	1	2	1	73	COM	EXTENDED	3	7	6
54	LSRB	INHERENT	1	2	1	74	LSR	EXTENDED	3	7	6
55	---	---	---	---	---	75	EIM	EXTENDED	4	7	7
56	RORB	INHERENT	1	2	1	76	ROR	EXTENDED	3	7	6
57	ASRB	INHERENT	1	2	1	77	ASR	EXTENDED	3	7	6
58	ALSB/LSLB	INHERENT	1	2	1	78	ASL/LSL	EXTENDED	3	7	6
59	ROLB	INHERENT	1	2	1	79	ROL	EXTENDED	3	7	6
5A	DECB	INHERENT	1	2	1	7A	DEC	EXTENDED	3	7	6
5B	---	--	---	---	---	7B	TIM	EXTENDED	4	5	5
5C	INCB	INHERENT	1	2	1	7C	INC	EXTENDED	3	7	6
5D	TSTB	INHERENT	1	2	1	7D	TST	EXTENDED	3	7	5
5E	---	---	---	---	---	7E	JMP	EXTENDED	3	4	3
5F	CLRB	INHERENT	1	2	1	7F	CLR	EXTENDED	3	7	6

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
80	SUBA	IMMEDIATE	2	2	2	A0	SUBA	INDEXED	2+	4+	4+
81	CMPA	IMMEDIATE	2	2	2	A1	CMPA	INDEXED	2+	4+	4+
82	SBCA	IMMEDIATE	2	2	2	A2	SBCA	INDEXED	2+	4+	4+
83	SUBD	IMMEDIATE	3	4	3	A3	SUBD	INDEXED	2+	6+	5+
84	ANDA	IMMEDIATE	2	2	2	A4	ANDA	INDEXED	2+	4+	4+
85	BITA	IMMEDIATE	2	2	2	A5	BITA	INDEXED	2+	4+	4+
86	LDA	IMMEDIATE	2	2	2	A6	LDA	INDEXED	2+	4+	4+
87	---	---	---	---	---	A7	STA	INDEXED	2+	4+	4+
88	EORA	IMMEDIATE	2	2	2	A8	EORA	INDEXED	2+	4+	4+
89	ADCA	IMMEDIATE	2	2	2	A9	ADCA	INDEXED	2+	4+	4+
8A	ORA	IMMEDIATE	2	2	2	AA	ORA	INDEXED	2+	4+	4+
8B	ADDA	IMMEDIATE	2	2	2	AB	ADDA	INDEXED	2+	4+	4+
8C	CMPX	IMMEDIATE	3	4	3	AC	CMPX	INDEXED	2+	6+	5+
8D	BSR	RELATIVE	2	7	6	AD	JSR	INDEXED	2+	7+	6+
8E	LDX	IMMEDIATE	3	3	3	AE	LDX	INDEXED	2+	5+	5+
8F	---	---	---	---	---	AF	STX	INDEXED	2+	5+	5+
90	SUBA	DIRECT	2	4	3	B0	SUBA	EXTENDED	3	5	4
91	CMPA	DIRECT	2	4	3	B1	CMPA	EXTENDED	3	5	4
92	SBCA	DIRECT	2	4	3	B2	SBCA	EXTENDED	3	5	4
93	SUBD	DIRECT	2	6	4	B3	SUBD	EXTENDED	3	7	5
94	ANDA	DIRECT	2	4	3	B4	ANDA	EXTENDED	3	5	4
95	BITA	DIRECT	2	4	3	B5	BITA	EXTENDED	3	5	4
96	LDA	DIRECT	2	4	3	B6	LDA	EXTENDED	3	5	4
97	STA	DIRECT	2	4	3	B7	STA	EXTENDED	3	5	4
98	EORA	DIRECT	2	4	3	B8	EORA	EXTENDED	3	5	4
99	ADCA	DIRECT	2	4	3	B9	ADCA	EXTENDED	3	5	4
9A	ORA	DIRECT	2	4	3	BA	ORA	EXTENDED	3	5	4
9B	ADDA	DIRECT	2	4	3	BB	ADDA	EXTENDED	3	5	4
9C	CMPX	DIRECT	2	6	4	BC	CMPX	EXTENDED	3	7	5
9D	JSR	DIRECT	2	7	6	BD	JSR	EXTENDED	3	8	7
9E	LDX	DIRECT	2	5	4	BE	LDX	EXTENDED	3	6	5
9F	STX	DIRECT	2	5	4	BF	STX	EXTENDED	3	6	5

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
C0	SUBB	IMMEDIATE	2	2	2	E0	SUBB	INDEXED	2+	4+	4+
C1	CMPB	IMMEDIATE	2	2	2	E1	CMPB	INDEXED	2+	4+	4+
C2	SBCB	IMMEDIATE	2	2	2	E2	SBCB	INDEXED	2+	4+	4+
C3	ADDD	IMMEDIATE	3	4	3	E3	ADDD	INDEXED	2+	6+	5+
C4	ANDB	IMMEDIATE	2	2	2	E4	ANDB	INDEXED	2+	4+	4+
C5	BITB	IMMEDIATE	2	2	2	E5	BITB	INDEXED	2+	4+	4+
C6	LDB	IMMEDIATE	2	2	2	E6	LDB	INDEXED	2+	4+	4+
C7	---	---	---	---	---	E7	STB	INDEXED	2+	4+	4+
C8	EORB	IMMEDIATE	2	2	2	E8	EORB	INDEXED	2+	4+	4+
C9	ADCB	IMMEDIATE	2	2	2	E9	ADCB	INDEXED	2+	4+	4+
CA	ORB	IMMEDIATE	2	2	2	EA	ORB	INDEXED	2+	4+	4+
CB	ADDB	IMMEDIATE	2	2	2	EB	ADDB	INDEXED	2+	4+	4+
CC	LDD	IMMEDIATE	3	3	3	EC	LDD	INDEXED	2+	5+	5+
CD	LDQ	IMMEDIATE	5	5	5	ED	STD	INDEXED	2+	5+	5+
CE	LDU	IMMEDIATE	3	3	3	EE	LDU	INDEXED	2+	5+	5+
CF	---	---	---	---	---	EF	STU	INDEXED	2+	5+	5+
D0	SUBB	DIRECT	2	4	3	F0	SUBB	EXTENDED	3	5	4
D1	CMPB	DIRECT	2	4	3	F1	CMPB	EXTENDED	3	5	4
D2	SBCB	DIRECT	2	4	3	F2	SBCB	EXTENDED	3	5	4
D3	ADDD	DIRECT	2	6	4	F3	ADDD	EXTENDED	3	7	5
D4	ANDB	DIRECT	2	4	3	F4	ANDB	EXTENDED	3	5	4
D5	BITB	DIRECT	2	4	3	F5	BITB	EXTENDED	3	5	4
D6	LDB	DIRECT	2	4	3	F6	LDB	EXTENDED	3	5	4
D7	STB	DIRECT	2	4	3	F7	STB	EXTENDED	3	5	4
D8	EORB	DIRECT	2	4	3	F8	EORB	EXTENDED	3	5	4
D9	ADCB	DIRECT	2	4	3	F9	ADCB	EXTENDED	3	5	4
DA	ORB	DIRECT	2	4	3	FA	ORB	EXTENDED	3	5	4
DB	ADDB	DIRECT	2	4	3	FB	ADDB	EXTENDED	3	5	4
DC	LDD	DIRECT	2	5	4	FC	LDD	EXTENDED	3	6	5
DD	STD	DIRECT	2	5	4	FD	STD	EXTENDED	3	6	5
DE	LDU	DIRECT	2	5	4	FE	LDU	EXTENDED	3	6	5
DF	STU	DIRECT	2	5	4	FF	STU	EXTENDED	3	6	5

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
1000	---	---	---	---	---	1020	---	---	---	---	---
1001	---	---	---	---	---	1021	LBRN	RELATIVE	4	5	5
1002	---	---	---	---	---	1022	LBHI	RELATIVE	4	5/6	5
1003	---	---	---	---	---	1023	LBLS	RELATIVE	4	5/6	5
1004	---	---	---	---	---	1024	LBHS/LBCC	RELATIVE	4	5/6	5
1005	---	---	---	---	---	1025	LBLO/LBCS	RELATIVE	4	5/6	5
1006	---	---	---	---	---	1026	LBNE	RELATIVE	4	5/6	5
1007	---	---	---	---	---	1027	LBEQ	RELATIVE	4	5/6	5
1008	---	---	---	---	---	1028	LBVC	RELATIVE	4	5/6	5
1009	---	---	---	---	---	1029	LBVS	RELATIVE	4	5/6	5
100A	---	---	---	---	---	102A	LBPL	RELATIVE	4	5/6	5
100B	---	---	---	---	---	102B	LBMI	RELATIVE	4	5/6	5
100C	---	---	---	---	---	102C	LBGE	RELATIVE	4	5/6	5
100D	---	---	---	---	---	102D	LBLT	RELATIVE	4	5/6	5
100E	---	---	---	---	---	102E	LBGT	RELATIVE	4	5/6	5
100F	---	---	---	---	---	102F	LBLE	RELATIVE	4	5/6	5
1010	---	---	---	---	---	1030	ADDR	REGISTER	3	4	4
1011	---	---	---	---	---	1031	ADCR	REGISTER	3	4	4
1012	---	---	---	---	---	1032	SUBR	REGISTER	3	4	4
1013	---	---	---	---	---	1033	SBCR	REGISTER	3	4	4
1014	---	---	---	---	---	1034	ANDR	REGISTER	3	4	4
1015	---	---	---	---	---	1035	ORR	REGISTER	3	4	4
1016	---	---	---	---	---	1036	EORR	REGISTER	3	4	4
1017	---	---	---	---	---	1037	CMPR	REGISTER	3	4	4
1018	---	---	---	---	---	1038	PSHSW	INHERENT	2	6	6
1019	---	---	---	---	---	1039	PULSW	INHERENT	2	6	6
101A	---	---	---	---	---	103A	PSHUW	INHERENT	2	6	6
101B	---	---	---	---	---	103B	PULUW	INHERENT	2	6	6
101C	---	---	---	---	---	103C	---	---	---	---	---
101D	---	---	---	---	---	103D	---	---	---	---	---
101E	---	---	---	---	---	103E	---	---	---	---	---
101F	---	---	---	---	---	103F	SWI2	INHERENT	2	20	22

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ˜	NM ˜
1040	NEGD	INHERENT	2	2	1
1041	---	---	---	---	---
1042	---	---	---	---	---
1043	COMD	INHERENT	2	2	1
1044	LSRD	INHERENT	2	2	1
1045	---	---	---	---	---
1046	RORD	INHERENT	2	2	1
1047	ASRD	INHERENT	2	2	1
1048	ASLD/LSLD	INHERENT	2	2	1
1049	ROLD	INHERENT	2	2	1
104A	DECD	INHERENT	2	2	1
104B	---	---	---	---	---
104C	INCD	INHERENT	2	2	1
104D	TSTD	INHERENT	2	2	1
104E	---	---	---	---	---
104F	CLRD	INHERENT	2	2	1
1050	NEGW	INHERENT	2	2	1
1051	---	---	---	---	---
1052	---	---	---	---	---
1053	COMW	INHERENT	2	3	2
1054	LSRW	INHERENT	2	3	2
1055	---	---	---	---	---
1056	RORW	INHERENT	2	3	2
1057	ASRW	INHERENT	2	2	1
1058	ASLW/LSLW	INHERENT	2	2	1
1059	ROLW	INHERENT	2	3	2
105A	DECW	INHERENT	2	3	2
105B	---	--	---	---	---
105C	INCW	INHERENT	2	3	2
105D	TSTW	INHERENT	2	3	2
105E	---	---	---	---	---
105F	CLRW	INHERENT	2	3	1

OP	MNEM	MODE	#	EM ˜	NM ˜
1060	---	---	---	---	---
1061	---	---	---	---	---
1062	---	---	---	---	---
1063	---	---	---	---	---
1064	---	---	---	---	---
1065	---	---	---	---	---
1066	---	---	---	---	---
1067	---	---	---	---	---
1068	---	---	---	---	---
1069	---	---	---	---	---
106A	---	---	---	---	---
106B	---	---	---	---	---
106C	---	---	---	---	---
106D	---	---	---	---	---
106E	---	---	---	---	---
106F	---	---	---	---	---
1070	---	---	---	---	---
1071	---	---	---	---	---
1072	---	---	---	---	---
1073	---	---	---	---	---
1074	---	---	---	---	---
1075	---	---	---	---	---
1076	---	---	---	---	---
1077	---	---	---	---	---
1078	---	---	---	---	---
1079	---	---	---	---	---
107A	---	---	---	---	---
107B	---	---	---	---	---
107C	---	---	---	---	---
107D	---	---	---	---	---
107E	---	---	---	---	---
107F	---	---	---	---	---

Appendix A: 6309 Programming Card

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
1080	SUBW	IMMEDIATE	4	5	4	10A0	SUBW	INDEXED	3+	7+	6+
1081	CMPW	IMMEDIATE	4	5	4	10A1	CMPW	INDEXED	3+	7+	6+
1082	SBCD	IMMEDIATE	4	5	4	10A2	SBCD	INDEXED	3+	7+	6+
1083	CMPD	IMMEDIATE	4	5	4	10A3	CMPD	INDEXED	3+	7+	6+
1084	ANDD	IMMEDIATE	4	5	4	10A4	ANDD	INDEXED	3+	7+	6+
1085	BITD	IMMEDIATE	4	5	4	10A5	BITD	INDEXED	3+	7+	6+
1086	LDW	IMMEDIATE	4	4	4	10A6	LDW	INDEXED	3+	6+	6+
1087	---	---	---	---	---	10A7	STW	INDEXED	3+	6+	6+
1088	EORD	IMMEDIATE	4	5	4	10A8	EORD	INDEXED	3+	7+	6+
1089	ADCD	IMMEDIATE	4	5	4	10A9	ADCD	INDEXED	3+	7+	6+
108A	ORD	IMMEDIATE	4	5	4	10AA	ORD	INDEXED	3+	7+	6+
108B	ADDW	IMMEDIATE	4	5	4	10AB	ADDW	INDEXED	3+	7+	6+
108C	CMPY	IMMEDIATE	4	5	4	10AC	CMPY	INDEXED	3+	7+	6+
108D	---	---	---	---	---	10AD	---	---	---	---	---
108E	LDY	IMMEDIATE	4	4	4	10AE	LDY	INDEXED	3+	6+	6+
108F	---	---	---	---	---	10AF	STY	INDEXED	3+	6+	6+
1090	SUBW	DIRECT	3	7	5	10B0	SUBW	EXTENDED	4	8	6
1091	CMPW	DIRECT	3	7	5	10B1	CMPW	EXTENDED	4	8	6
1092	SBCD	DIRECT	3	7	5	10B2	SBCD	EXTENDED	4	8	6
1093	CMPD	DIRECT	3	7	5	10B3	CMPD	EXTENDED	4	8	6
1094	ANDD	DIRECT	3	7	5	10B4	ANDD	EXTENDED	4	8	6
1095	BITD	DIRECT	3	7	5	10B5	BITD	EXTENDED	4	8	6
1096	LDW	DIRECT	3	6	5	10B6	LDW	EXTENDED	4	7	6
1097	STW	DIRECT	3	6	5	10B7	STW	EXTENDED	4	7	6
1098	EORD	DIRECT	3	7	5	10B8	EORD	EXTENDED	4	8	6
1099	ADCD	DIRECT	3	7	5	10B9	ADCD	EXTENDED	4	8	6
109A	ORD	DIRECT	3	7	5	10BA	ORD	EXTENDED	4	8	6
109B	ADDW	DIRECT	3	7	5	10BB	ADDW	EXTENDED	4	8	6
109C	CMPY	DIRECT	3	7	5	10BC	CMPY	EXTENDED	4	8	6
109D	---	---	---	---	---	10BD	---	---	---	---	---
109E	LDY	DIRECT	3	6	5	10BE	LDY	EXTENDED	4	7	6
109F	STY	DIRECT	3	6	5	10BF	STY	EXTENDED	4	7	6

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
10C0	---	---	---	---	---	10E0	---	---	---	---	---
10C1	---	---	---	---	---	10E1	---	---	---	---	---
10C2	---	---	---	---	---	10E2	---	---	---	---	---
10C3	---	---	---	---	---	10E3	---	---	---	---	---
10C4	---	---	---	---	---	10E4	---	---	---	---	---
10C5	---	---	---	---	---	10E5	---	---	---	---	---
10C6	---	---	---	---	---	10E6	---	---	---	---	---
10C7	---	---	---	---	---	10E7	---	---	---	---	---
10C8	---	---	---	---	---	10E8	---	---	---	---	---
10C9	---	---	---	---	---	10E9	---	---	---	---	---
10CA	---	---	---	---	---	10EA	---	---	---	---	---
10CB	---	---	---	---	---	10EB	---	---	---	---	---
10CC	---	---	---	---	---	10EC	LDQ	INDEXED	3+	8+	8+
10CD	---	---	---	---	---	10ED	STQ	INDEXED	3+	8+	8+
10CE	LDS	IMMEDIATE	4	4	4	10EE	LDS	INDEXED	3+	6+	6+
10CF	---	---	---	---	---	10EF	STS	INDEXED	3+	6+	6+
10D0	---	---	---	---	---	10F0	---	---	---	---	---
10D1	---	---	---	---	---	10F1	---	---	---	---	---
10D2	---	---	---	---	---	10F2	---	---	---	---	---
10D3	---	---	---	---	---	10F3	---	---	---	---	---
10D4	---	---	---	---	---	10F4	---	---	---	---	---
10D5	---	---	---	---	---	10F5	---	---	---	---	---
10D6	---	---	---	---	---	10F6	---	---	---	---	---
10D7	---	---	---	---	---	10F7	---	---	---	---	---
10D8	---	---	---	---	---	10F8	---	---	---	---	---
10D9	---	---	---	---	---	10F9	---	---	---	---	---
10DA	---	---	---	---	---	10FA	---	---	---	---	---
10DB	---	---	---	---	---	10FB	---	---	---	---	---
10DC	LDQ	DIRECT	3	8	7	10FC	LDQ	EXTENDED	4	9	8
10DD	STQ	DIRECT	3	8	7	10FD	STQ	EXTENDED	4	9	8
10DE	LDS	DIRECT	3	6	5	10FE	LDS	EXTENDED	4	7	6
10DF	STS	DIRECT	3	6	5	10FF	STS	EXTENDED	4	7	6

OP	MNEM	MODE	#	EM ~	NM ~
1100	---	---	---	---	---
1101	---	---	---	---	---
1102	---	---	---	---	---
1103	---	---	---	---	---
1104	---	---	---	---	---
1105	---	---	---	---	---
1106	---	---	---	---	---
1107	---	---	---	---	---
1108	---	---	---	---	---
1109	---	---	---	---	---
110A	---	---	---	---	---
110B	---	---	---	---	---
110C	---	---	---	---	---
110D	---	---	---	---	---
110E	---	---	---	---	---
110F	---	---	---	---	---
1110	---	---	---	---	---
1111	---	---	---	---	---
1112	---	---	---	---	---
1113	---	---	---	---	---
1114	---	---	---	---	---
1115	---	---	---	---	---
1116	---	---	---	---	---
1117	---	---	---	---	---
1118	---	---	---	---	---
1119	---	---	---	---	---
111A	---	---	---	---	---
111B	---	---	---	---	---
111C	---	---	---	---	---
111D	---	---	---	---	---
111E	---	---	---	---	---
111F	---	---	---	---	---

OP	MNEM	MODE	#	EM ~	NM ~
1120	---	---	---	---	---
1121	---	---	---	---	---
1122	---	---	---	---	---
1123	---	---	---	---	---
1124	---	---	---	---	---
1125	---	---	---	---	---
1126	---	---	---	---	---
1127	---	---	---	---	---
1128	---	---	---	---	---
1129	---	---	---	---	---
112A	---	---	---	---	---
112B	---	---	---	---	---
112C	---	---	---	---	---
112D	---	---	---	---	---
112E	---	---	---	---	---
112F	---	---	---	---	---
1130	BAND	SINGLE BIT	4	7	6
1131	BIAND	SINGLE BIT	4	7	6
1132	BOR	SINGLE BIT	4	7	6
1133	BIOR	SINGLE BIT	4	7	6
1134	BEOR	SINGLE BIT	4	7	6
1135	BIEOR	SINGLE BIT	4	7	6
1136	LDBT	SINGLE BIT	4	7	6
1137	STBT	SINGLE BIT	4	8	7
1138	TFM	REGISTER	3	6+3W	6+3W
1139	TFM	REGISTER	3	6+3W	6+3W
113A	TFM	REGISTER	3	6+3W	6+3W
113B	TFM	REGISTER	3	6+3W	6+3W
113C	BITMD	IMMEDIATE	3	4	4
113D	LDMD	IMMEDIATE	3	5	5
113E	---	---	---	---	---
113F	SWI3	INHERENT	2	20	22

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
1140	---	---	---	---	---	1160	---	---	---	---	---
1141	---	---	---	---	---	1161	---	---	---	---	---
1142	---	---	---	---	---	1162	---	---	---	---	---
1143	COME	INHERENT	2	2	2	1163	---	---	---	---	---
1144	---	---	---	---	---	1164	---	---	---	---	---
1145	---	---	---	---	---	1165	---	---	---	---	---
1146	---	---	---	---	---	1166	---	---	---	---	---
1147	---	---	---	---	---	1167	---	---	---	---	---
1148	---	---	---	---	---	1168	---	---	---	---	---
1149	---	---	---	---	---	1169	---	---	---	---	---
114A	DECE	INHERENT	2	2	2	116A	---	---	---	---	---
114B	---	---	---	---	---	116B	---	---	---	---	---
114C	INCE	INHERENT	2	2	2	116C	---	---	---	---	---
114D	TSTE	INHERENT	2	2	2	116D	---	---	---	---	---
114E	---	---	---	---	---	116E	---	---	---	---	---
114F	CLRE	INHERENT	2	2	2	116F	---	---	---	---	---
1150	---	---	---	---	---	1170	---	---	---	---	---
1151	---	---	---	---	---	1171	---	---	---	---	---
1152	---	---	---	---	---	1172	---	---	---	---	---
1153	COMF	INHERENT	2	2	2	1173	---	---	---	---	---
1154	---	---	---	---	---	1174	---	---	---	---	---
1155	---	---	---	---	---	1175	---	---	---	---	---
1156	---	---	---	---	---	1176	---	---	---	---	---
1157	---	---	---	---	---	1177	---	---	---	---	---
1158	---	---	---	---	---	1178	---	---	---	---	---
1159	---	---	---	---	---	1179	---	---	---	---	---
115A	DECF	INHERENT	2	2	2	117A	---	---	---	---	---
115B	---	---	---	---	---	117B	---	---	---	---	---
115C	INCF	INHERENT	2	2	2	117C	---	---	---	---	---
115D	TSTF	INHERENT	2	2	2	117D	---	---	---	---	---
115E	---	---	---	---	---	117E	---	---	---	---	---
115F	CLRF	INHERENT	2	2	2	117F	---	---	---	---	---

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
1180	SUBE	IMMEDIATE	3	3	3	11A0	SUBE	INDEXED	3+	5+	5+
1181	CMPE	IMMEDIATE	3	3	3	11A1	CMPE	INDEXED	3+	5+	5+
1182	---	---	---	---	---	11A2	---	---	---	---	---
1183	CMPU	IMMEDIATE	4	5	4	11A3	CMPU	INDEXED	3+	7+	6+
1184	---	---	---	---	---	11A4	---	---	---	---	---
1185	---	---	---	---	---	11A5	---	---	---	---	---
1186	LDE	IMMEDIATE	3	3	3	11A6	LDE	INDEXED	3+	5+	5+
1187	---	---	---	---	---	11A7	STE	INDEXED	3+	5+	5+
1188	---	---	---	---	---	11A8	---	---	---	---	---
1189	---	---	---	---	---	11A9	---	---	---	---	---
118A	---	---	---	---	---	11AA	---	---	---	---	---
118B	ADDE	IMMEDIATE	3	3	3	11AB	ADDE	INDEXED	3+	5+	5+
118C	CMPS	IMMEDIATE	4	5	4	11AC	CMPS	INDEXED	3+	7+	6+
118D	DIVD	IMMEDIATE	3	25	25	11AD	DIVD	INDEXED	3+	27+	27+
118E	DIVQ	IMMEDIATE	4	34	34	11AE	DIVQ	INDEXED	3+	36+	36+
118F	MULD	IMMEDIATE	4	28	28	11AF	MULD	INDEXED	3+	30+	30+
1190	SUBE	DIRECT	3	5	4	11B0	SUBE	EXTENDED	4	6	5
1191	CMPE	DIRECT	3	5	4	11B1	CMPE	EXTENDED	4	6	5
1192	---	---	---	---	---	11B2	---	---	---	---	---
1193	CMPU	DIRECT	3	7	5	11B3	CMPU	EXTENDED	4	8	6
1194	---	---	---	---	---	11B4	---	---	---	---	---
1195	---	---	---	---	---	11B5	---	---	---	---	---
1196	LDE	DIRECT	3	5	4	11B6	LDE	EXTENDED	4	6	5
1197	STE	DIRECT	3	5	4	11B7	STE	EXTENDED	4	6	5
1198	---	---	---	---	---	11B8	---	---	---	---	---
1199	---	---	---	---	---	11B9	---	---	---	---	---
119A	---	---	---	---	---	11BA	---	---	---	---	---
119B	ADDE	DIRECT	3	5	4	11BB	ADDE	EXTENDED	4	6	5
119C	CMPS	DIRECT	3	7	5	11BC	CMPS	EXTENDED	4	8	6
119D	DIVD	DIRECT	3	27	26	11BD	DIVD	EXTENDED	4	28	27
119E	DIVQ	DIRECT	3	36	35	11BE	DIVQ	EXTENDED	4	37	36
119F	MULD	DIRECT	3	30	29	11BF	MULD	EXTENDED	4	31	30

OP	MNEM	MODE	#	EM ~	NM ~	OP	MNEM	MODE	#	EM ~	NM ~
11C0	SUBF	IMMEDIATE	3	3	3	1lE0	SUBF	INDEXED	3+	5+	5+
11C1	CMPF	IMMEDIATE	3	3	3	1lE1	CMPF	INDEXED	3+	5+	5+
11C2	---	---	---	---	---	1lE2	---	---	---	---	---
11C3	---	---	---	---	---	1lE3	---	---	---	---	---
11C4	---	---	---	---	---	1lE4	---	---	---	---	---
11C5	---	---	---	---	---	1lE5	---	---	---	---	---
11C6	LDF	IMMEDIATE	3	3	3	1lE6	LDF	INDEXED	3+	5+	5+
11C7	---	---	---	---	---	1lE7	STF	INDEXED	3+	5+	5+
11C8	---	---	---	---	---	1lE8	---	---	---	---	---
11C9	---	---	---	---	---	1lE9	---	---	---	---	---
11CA	---	---	---	---	---	1lEA	---	---	---	---	---
11CB	ADDF	IMMEDIATE	3	3	3	1lEB	ADDF	INDEXED	3+	5+	5+
11CC	---	---	---	---	---	1lEC	---	---	---	---	---
11CD	---	---	---	---	---	1lED	---	---	---	---	---
11CE	---	---	---	---	---	1lEE	---	---	---	---	---
11CF	---	---	---	---	---	1lEF	---	---	---	---	---
11D0	SUBF	DIRECT	3	5	4	11F0	SUBF	EXTENDED	4	6	5
11D1	CMPF	DIRECT	3	5	4	11F1	CMPF	EXTENDED	4	6	5
11D2	---	---	---	---	---	11F2	---	---	---	---	---
11D3	---	---	---	---	---	11F3	---	---	---	---	---
11D4	---	---	---	---	---	11F4	---	---	---	---	---
11D5	---	---	---	---	---	11F5	---	---	---	---	---
11D6	LDF	DIRECT	3	5	4	11F6	LDF	EXTENDED	4	6	5
11D7	STF	DIRECT	3	5	4	11F7	STF	EXTENDED	4	6	5
11D8	---	---	---	---	---	11F8	---	---	---	---	---
11D9	---	---	---	---	---	11F9	---	---	---	---	---
11DA	---	---	---	---	---	11FA	---	---	---	---	---
11DB	ADDF	DIRECT	3	5	4	11FB	ADDF	EXTENDED	4	6	5
11DC	---	---	---	---	---	11FC	---	---	---	---	---
11DD	---	---	---	---	---	11FD	---	---	---	---	---
11DE	---	---	---	---	---	11FE	---	---	---	---	---
11DF	---	---	---	---	---	11FF	---	---	---	---	---

INDEXED MODE	FORM	+#	+~	Post-Byte	W Post-Byte
No Offset	,r	0	0	1RR00100	10001111
5-Bit Offset	n5,r	0	1	0RRnnnnn	---
8-Bit Offset	n8,r	1	1	1RR01000	---
16-Bit Offset	n16,r	2	4	1RR01001	10101111
A Reg. Offset	A,r	0	1	1RR00110	---
B Reg. Offset	B,r	0	1	1RR00101	---
D Reg. Offset	D,r	0	4	1RR01011	---
E Reg. Offset	E,r	0	1	1RR00111	---
F Reg. Offset	F,r	0	1	1RR01010	---
W Reg. Offset	W,r	0	4	1RR01110	---
Post-Increment by 1	,r+	0	2	1RR00000	---
Post-Increment by 2	,r++	0	3	1RR00001	11001111
Pre-Decrement by 1	,-r	0	2	1RR00010	---
Pre-Decrement by 2	,--r	0	3	1RR00011	11101111
8-Bit PC Relative	n8,PCR	1	1	1XX01100	---
16-Bit PC-Relative	n16,PCR	2	5	1XX01101	---
Indirect, No Offset	[,r]	0	3	1RR10100	10010000
Indirect, 8-Bit Offset	[n8,r]	1	4	1RR11000	---
Indirect, 16-Bit Offset	[n16,r]	2	7	1RR11001	10110000
Indirect, A Reg. Offset	[A,r]	0	4	1RR10110	---
Indirect, B Reg. Offset	[B,r]	0	4	1RR10101	---
Indirect, D Reg. Offset	[D,r]	0	7	1RR11011	---
Indirect, E Reg. Offset	[E,r]	0	7	1RR10111	---
Indirect, F Reg. Offset	[F,r]	0	7	1RR11010	---
Indirect, W Reg. Offset	[W,r]	0	7	1RR11110	---
Indirect, Post-Increment by 2	[,r++]	0	6	1RR10001	11010000
Indirect, Pre-Decrement by 2	[,--r]	0	6	1RR10011	11110000
Indirect, 8-Bit PC-Relative	[n8,PCR]	1	4	1RR11100	---
Indirect, 16-Bit PC-Relative	[n16,PCR]	2	8	1XX11101	---
Indirect, Extended	[n16]	2	5	10011111	---

RR is register used. XX=don't care,
00=X, 01=Y, 10=U, 11=S

Appendix A: 6309 Programming Card

Stack (PSH / PUL) Post-Byte

7	6	5	4	3	2	1	0
PC	**S/U**	**Y**	**X**	**DP**	**B**	**A**	**CC**

Register-to-Register (TFR) Post-Byte

7	6	5	4	3	2	1	0
Source				**Destination**			

For Source or Destination:

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

Bit Manipulation (BAND) Post-Byte

7	6	5	4	3	2	1	0
Register		**Memory bit #(0..7)**			**Register bit #(0..7)**		

For Register:

%00	CC	%10	B
%01	A	%11	---

Inside Back Cover.
Intentionally left blank.