CCASM 2006.2
User's Guide

6809/6309 Cross Assembler for 32-bit Windows

and

HD63B09EP Technical Reference Guide

Distributed by:
http://www.coco3.com
The TRS-80/Tandy Color Computer Resource Site

Table of Contents

Table of Contents (cont.)

Introduction

CCASM is a Windows-based 6809/6309 machine language cross-assembler created with TRS-80 Color Computer and Vectrex users in mind. The command is issuable from any console prompt, batch file, another program, etc. Specifying a source code file and some optional parameters, your programs can be quickly assembled and ready to run on any 6809 or 6309-based computer. For CoCo users, most Tandy EDTASM source code can be assembled without any modifications.

For Beginners

If you've never worked with assembly, many examples are given in this guide and the included source code files for helping you learn how to accomplish common tasks. Once you start putting together small routines and programs, there's no limit to what can be created. Learn the language first and your programming style will build over time.

Ofcourse, there's no certain style required to create great ML programs. CCASM also offers high-level functions to help take the pain away from writing raw assembly programs.

For Experts

You're definately not limited to assembling just EDTASM-compatible source code. Many other powerful psuedo-ops, directives, and instructions are available which will help you create programs that can be bigger, faster, and easier to build.

You have the leisure of namespaces, structures, procedures, procedure libraries and more, allowing you to create much more powerful programs in less time than it would take using a bare-bones assembler.

As CCASM advances, more options, features, and high-level structures will be added making it one of the most powerful 6809/6309 assemblers available.

Summary of Features

program type: 32-bit Windows command prompt
target systems for assembled code: Tandy CoCo 1,2,3; Vectrex, and any 6809 or 6309-based computer
assembled files: 'LOADM' record format, ROM and ROM-like images
accepted source code formats: Tandy EDTASM and variants
source code file compatibility: CoCo text editors, PC text editors, various LF/CR support
maximum source code lines: 32,768
maximum nested include levels: virtually unlimited
assembly passes: 2
nested conditional assembly: yes
expression evaluator: unlimited nesting, logical operations
structures: yes
procedures: yes, nesting & local variable support

Terms Used In This Guide

white space (TABs or SPACEs between source code line fields)
symbol/label (alpha-numeric name that translates into a value or address)
mnemonic (CPU instruction not including any operand)
operand (data used by the mnemonic to form the instruction)
conditional assembly (code segments assembled only if a case is true or false)
PC (the CPU's *program counter* register)
reg. (CPU register/accumulator/pointer)
expression (a way of specifying a simplified or mathematical value)
void (reserved but uninitialized memory)
word (2-byte/16-bit data)
dword (4-byte/32-bit data)
MSB (most-significant byte, leftmost as in MSB/LSB, lower memory address)
LSB (least-significant byte, rightmost as in MSB/LSB, higher memory address)
MSBit (most-significant bit, leftmost as in **b**bbbbbbb)
LSBit (least-significant bit, rightmost as in bbbbbbb**b**)
Boolean (0 means False and <>0 means True)
data structure (related group of data objects)

Command Options

-l	[dump assembly listing]
-s	[dump symbols]
-sa	[dump extended symbols and structures]
-sa	[dump symbols, including automatic & local labels]
-o=	[override default filename for binary output]
-bin	[assemble as Tandy CoCo 'LOADM/EXEC' file (default)]
-sr	[assemble as single-record file having only one origin]
-nr	[assemble with no origin records]
-s19	[assemble as s-record object file]
-rom{=}	[assemble as ROM image of 2k,4k,{8k},16,32,64,128,256]
-h	[show help messages along with any errors]
-d	[show debug messages]
-z	[internal debug listing]
-e	[allow EDTASM .operators.]
-de=	[filename for error reporting]
-q	[quiet mode]
-v	[hide title]
-os9	[output as OS-9 module – NOT AVAILABLE YET]

Example of the -o option

cm array -o=array.sys
(assemble array.asm to array.sys)

Examples of the -rom option

cm mygame -rom
(assemble mygame.asm to mygame.rom of exactly 8192 bytes)

cm newbasic -rom=32k
(assemble newbasic.asm to newbasic.rom of exactly 32768 bytes)

ROM image files are pure data and are compatible with all or most EPROM-burning software, even if you need to rename the files so they will load into your utility.

Example of the -l option

cm mygame -l >listing.txt
(assemble mygame.asm to mygame.bin and send a listing to the file "listing.txt")

Example of the -s option

cm pacman -s
(assemble pacman.asm to pacman.bin and dump the symbol table to the screen)

The CPU Registers

6809 Registers

a [8-bit accumulator]
b [8-bit accumulator]
d [16-bit concatenated register of **a**/**b**]
x [16-bit pointer]
y [16-bit pointer]
u [User Stack or 16-bit pointer]
s [System Stack or 16-bit pointer]
dp [Direct-Page Register]
pc [16-bit Program Counter]
cc [8-bit CPU condition-code register {E-F-H-I-N-Z-V-C}]
cc flags:
- E [Entire State on stack - determines RTI action]
- F [Fast Interrupt mask - set to enable FIRQ-to-CPU]
- H [Half Carry - carry out of bit 3 of arithmetic data]
- I [IRQ interrupt mask - set to enable IRQ-to-CPU]
- N [Negative Code - automatically set if result is negative]
- Z [Zero Code - set if result is zero]
- V [Overflow Code - set for arithmetic overflow]
- C [Carry Code - set for math carries and borrows]

6309-Only Registers

The 6309 CPU has all of the 6809 registers, plus:

e [8-bit accumulator]
f [8-bit accumulator]
w [16-bit concatenated reg. of ***e/f***]
q [32-bit concatentated reg. of **a/b/*e/f***]
v [16-bit accumulator] *
z [Zero reg.]*
0 [Zero reg.] *
00 [alternate Zero reg.] **
md [Mode/Error reg.]

*Note that register names are case-insensitive, meaning **a** is the same as **A**, and **x** is the same as **X**, etc.*
** used by inter-register instructions only*
*** there are two Zero registers in the 6309 CPU*

Source Code Format

A variety of white space methods may be used in your source code. An intelligent parsing routine is used for breaking source code lines down into the fields used to build each instruction. CCASM will generate an error if the required line format is not met or if the combined fields do not form a valid function.

Source code lines:

1) are separated into fields by SPACEs or TABs
2) can optionally have a line number in the first field
3) can optionally have a label in the first field (second field if a line number is present)
4) must have a SPACE or TAB before all mnemonics, psuedo-ops, and trailing comments.

The following examples show the typical layout of any given source code line. The '-' character represents a SPACE or TAB used to separate fields.

Label-Mnemonic-Operand-Comment
Label-Mnemonic--Comment
Label-Mnemonic
-Mnemonic-Operand
-Mnemonic--Comment
LineNumber-Label-Mnemonic-Operand-Comment
LineNumber--Mnemonic-Operand-Comment

A TAB-formatted line might look like this:

start	jsr	subroutine	this is a comment

Or, since line numbers are supported:

00010	start	jsr	subroutine	this is a comment

A SPACE-formatted line might look like this:

00010 start jsr subroutine ;this is a comment

Labels and Symbols

Label and symbol names:

1) should generally be kept under 32 character long
2) should not be named the same as any reserved symbol
3) should not contain any mathematical characters or names used by the expression evaluator

Although the CCASM preference is to use lowercase-oriented source code, capitol letters are welcome if that is what you prefer. However, symbol names are case-sensitive. In other words, the symbol "color" is not the same as the symbol "Color".

Automatic Symbols

The following symbols and their values are automatically set by the assembler.

*	[returns the address of the Program Counter]
.	[returns the offset into the operand]
sizeof{struct}	[returns the size of a data structure]

Standard Labels:

```
jmp label
bsr some_routine
```

Local Labels:

Local labels are resusable labels containing at least one '@' character or '?' character and generally kept short. Local labels may be used to save symbol table space or to avoid having to think of many unique label names in large programs.

You can reuse the same local label name many times as long as a blank line separates them. This scheme can be pictured as *local blocks* of source code, each possibly containing local labels used in other blocks. Local blocks cannot access local labels used in other blocks.

```
lbra a@
bra ?b
jmp @@exit
```

Branch Points:

Branch Points are very similar to local labels but they are much more efficient and easier to type. They can also save you lots of time thinking of *named* labels.

Using the single-character label called '**!**', you can branch forward and backward in your source code to the nearest *Branch Point*. Debugging your programs can be more difficult if you use too many Branch Points; therefore, they are best for short code segments.

```
        bra   <     branch backward to nearest Branch Point label
        bra   >     branch forward to nearest Branch Point label
```

example:

```
!       lda   ,x+   grab a byte from table
        bne   <     branch upwards to last "!" label
        bra   >     branch downwards to next "!" label
        nop
!       rts         exit
```

Psuedo-Ops and Directives

The following list of assembler commands are used in the mnemonic/operand fields just like regular instructions, only they generate data or perform special assembler functions; they do not automatically create CPU instructions.

title {string} [set the title of the source code]
org {address} [set/change program origin address]
include {filename*[.asm]*} [insert/include another source file at the current line]
includebin {filename*[.bin]*} [insert any file into the codestream]
proc {parameter:type,parameter:type...} [define a procedure]
call {procedure,param1,param2,param3...} [call a procedure]
namespace {label} [causes {label} to prefix to all subsequent labels]
endnamespace [end all namespaces in effect]
struct [start a data structure containing fields]
endstruct [end a structure]
union [start a union structure where the PC doesn't advance per object]
endunion [end a union structure]
page [inject a FORM-FEED character into the assembly listing]
setdp {0-255} [inform the assembler of the Direct Page register value]
{label} **equ** {expression} [assign a value to a label, becoming a symbol]
{label} **=** {expression} [assign a value to a label, becoming a symbol]
{label} **set** {expression} [reassign a value to a label, becoming a symbol]
even [align the PC on an even address]
odd [align the PC on an odd address]
align [align the PC on any boundary]
fcc {"string"} [form constant character string]
fcn {"string"} [form null-terminated string, adds (0) to end]
fcs {"string"} [form sign-terminated string, sets bit 7 of last character]
fcr {"string"} [form carriage-return/null-terminated string, adds 13,0 to end]
fcb {value,expression...} [form constant byte, 8-bit data]
fdb {value,expression...} [form double-byte/word/16-bit data]
fqb {value,expression...} [form quad-byte/dword/32-bit data]
fzb/**rzb** {number of cleared bytes} [form # of initialized byte(s)]
fzd/**rzd** {number of cleared words} [form # of initialized double-byte(s)]
fzq/**rzq** {number of cleared dwords} [form # of initialized quad-byte(s)]
rmb {number of voided bytes} [reserved memory, creates void]
rmd {number of voided words} [reserved memory, creates void]
rmq {number of voided dwords} [reserved memory, creates void]
end {address} [marks the end of assembly, used ***only once*** in master source file]

Conditional Assembly

Source lines between a *condition test* and an *end condition* statement are assembled only if the condition is true.

if {boolean expression} [start conditional assembly segment if condition=***true***]
ifeq [assemble segment if expression evaluates to ***zero***]
ifne [assemble segment if expression evaluates to ***nonzero***]
iflt [assemble segment if expression yields a ***negative*** result]
ifgt [assemble segment if expression yields a ***positive*** result]
ifle [assemble segment if expression yields a ***negative*** or ***zero*** result]
ifge [assemble segment if expression yields a ***positive*** or ***zero*** result]
cond {boolean expression} [start conditional assembly segment if result=***true***]
ifp1 [assemble source segment only if in assembly pass #1]
ifp2 [assemble source segment only if in assembly pass #2]
endif {end an **if** conditional assembly segment]
endc [end a **cond** conditional assembly segment]
endp [end an **ifp1/ifp2** conditional assembly segment]

Important notes*: Make sure all symbols to be used in conditional assembly expressions are predefined. Forward references are not supported within conditional assembly expressions. Nesting is supported up to 32 levels (virtually unlimited). Also… COMMENTS are NOT ALLOWED on the same line as a conditional statement, like so:*

```
if      coco.equ.3          this comment is not allowed
endif
```

Mnemonics

All legal 6809 mnemonics are supported by the 6309 CPU. Mnemonics and registers in ***italics*** are supported only by the 6309 CPU.

Loading & Moving Data Around

ld{**a,b,d,x,y,u,s,*e,f,w,q,md***} {memory,value} [load data into a reg.]
st{**a,b,d,x,y,u,s,*e,f,q,w***} {memory} [store reg. contents to mem.]
ldbt {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [transfer mem. bit into reg. bit]
stbt {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [transfer reg. bit into mem. bit]
band {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [AND mem. bit into reg.]
biand {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [AND complemented mem. bit into reg.]
bor {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [OR mem. bit into reg.]
bior {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [OR complemented mem. bit into reg.]
beor {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [EOR mem. bit into reg.]
bieor {**a,b**} , {source bit} , {dest. bit} , {DP mem.} [EOR complemented mem. bit into reg.]

copy {source reg.,destination reg.} [copy block of memory to another address]
copy- {source reg.,destination reg.} [copy block of memory in reverse]
imp {source reg.,destination reg.} [implode block of memory into one address]
exp {source reg.,destination reg.} [expand target into block of memory]
tfrp [same as ***copy***] *
tfrm [same as ***copy-***] *
tfrs [same as ***imp***] *
tfrr [same as ***exp***] *

** Used by the "EDTASM6309" assembler created by Robert Gault.*
*** The HD63B09EP Reference Guide by Chet Simpson and Alan Dekok mentions a single mnemonic not used in CCASM, called "TFM" for doing memory block operations. TFM R,R+ translates into **exp r,r**; TFM R+,R translates into **imp r,r;** TFM R-,R- translates into **copy- r,r**; and TFM R+,R+ translates into **copy r,r**.*

Comparing, Testing, And Clearing

clr{**a,b,*d,e,f,w***} [clear register]
clr {memory,index} [clear byte at memory location]
tst{**a,b,*d,e,f,w***} [test the target reg., setting reg. **cc**]
tst {memory} [test the target memory, setting reg. **cc**]
bit{**a,b,*d,md***} {memory,value} [test target bits with bits of a reg.]
cmp{a,b,d,x,y,u,s,*e,f,w*} [compare a reg. with memory data]

Saving And Restoring Registers On The Stacks

pshs {register list} [push registers onto **S**ystem stack}
puls {register list} [pull registers from **S**ystem stack}
pshu {register list} [push registers onto **U**ser stack}
pulu {register list} [pull registers from **U**ser stack}
pshsw [push reg. ***w*** onto **S**ystem Stack]
pulsw [pull reg. ***w*** register from **S**ystem stack]
pshuw [push reg. ***w*** onto **U**ser stack]
puluw [pull reg. ***w*** from **U**ser stack]

Doing Arithmetic

abx [add reg. **b** to reg. **x**]
add{**a,b,d,*e,f,w***} {memory,value} [add memory to reg.]
sub{**a,b,d,*e,f,w***} {memory,value} [subtract target from reg.]
adc{**a,b,d**} {memory,value} [add memory plus carry to reg.]
sbc{**a,b,*d***} {memory,value} [subtract target & carry from reg.]
daa [decimal-adjust contents of reg. **a**]
mul [multiply reg. **a** by reg. **b**, becoming reg. **d**]
muld {memory,value} [multiply **d** * operand, becoming **d**]
divd {memory,value} [divide register **d** by target, becoming **d**]
divq [divide register ***q*** by target]
inc{**a,b,*d,e,f,w***} [increment (add 1) to reg.]
inc {memory} [increment memory]
dec{**a,b,*d,e,f,w***} [decrement (subtract 1 from) reg.]
dec {memory} [decrement byte at memory location]
neg{**a,b,*d***} [negate (2's complement) a reg.]
neg {memory} [negate the target]
sexw [sign-extend reg. ***w*** (bit 15) into reg. **d**]
sex [sign-extend reg. **b** (bit 7) into reg. **a**]
asr{**a,b,*d***} [shift reg. bits to the right, retaining sign bit]
asr {memory} [shift memory bits to the right, retaining sign bit]
asl{**a,b,*d***} [shift reg. bits to the left, filling LSBit with zero]
asl {memory} [shift memory bits to the left, filling LSBit with zero]

Moving Around Within Your Programs

jmp {memory} [jmp to a direct/indirect address]
jsr {memory} [jump to a direct/indirect subroutine]
rts [return from subroutine (**jsr** or **bsr**); same as **puls pc**]
rti [return from interrupt (CPU- or **swi**-generated interrupt]
nop [no operation, code that does nothing]

Doing Bit-Based Operations

com{a,b,***d,e,f,w***} [1's-complement a CPU reg.]
com {memory} [1's-complement a byte of memory]
and{**a,b,cc,*d***} {memory,value} [logical AND of memory bits with a reg.]
or{**a,b,cc,*d***} {memory,value} [OR the bits of the target byte into a reg.]
eor{**a,b,*d***} {memory,value} [exclusive OR of target memory bits with reg.}
rol{**a,b,*d,w***} [rotate reg. bits to the left, filling LSBit with Carry]
rol {memory} [rotate memory bits to the left, filling LSBit with Carry]
ror{**a,b,*d,w***} [rotate reg. bits to the right, filling MSBit with Carry]
ror {memory} [rotate memory bits to the right, filling MSBit with Carry]
lsl{**a,b,*d***} [logical shift reg. bits to the left, filling LSBit with zero]
lsl {memory} [logical shift memory bits to the left, filling LSBit with zero]
lsr{**a,b,*d,w***} [logical shift reg. bits to the right, filling MSBit with zero]
lsr {memory} [logical shift memory bits to the right, filling MSBit with zero]
aim {value;memory} [AND the bits of the value with the bits of the memory byte]
eim {value;memory} [EOR/XOR the bits of the value with the bits of the memory byte]
oim {value;memory} [OR the bits of the value with the bits of the memory byte]
tim {value;memory} [TEST the bits of the value with the bits of the memory byte]

Operating Between Two Registers

exg {reg.,reg.} [exchange contents of two registers]
tfr {src. reg.,dest. reg.} [transfer src. reg. into dest. reg.]
lea{**x,y,u,s**} {offset,pointer} [load effective address]
adcr {source reg,destination reg} [add source reg. plus carry to destination reg.]
addr {source reg,destination reg} [add source reg. to destination reg.]
andr {source reg,destination reg} [AND of source reg. with the destination reg.]
cmpr {source reg,destination reg} [compare source reg. with destination reg.]
eorr {source reg,destination reg} [Exclusive OR of source reg. with destination reg.]
orr {source reg,destination reg} [OR of source reg. with destination reg.]
sbcr {source reg,destination reg} [subtract source reg. and carry from dest. reg.]
subr {source reg,destination reg} [subtract source reg. from destination reg.]

Handling Interrupts

cwai {#byte} [clear and wait for interrupt]
swi{**2,3**} [software (manual) interrupt types 2 and 3]
swi [software interrupt type 1]
sync [synchronize to interrupt]

Unconditional Relative Branches (always performed)

bra {address} [branch]
lbra {address} [long branch]
brn {address} [branch never]
lbrn {address} [long branch never]
bsr {address} [branch to a subroutine]
lbsr {address} [long branch to a subroutine]

Conditional Relative Branches based on (reg. cc) flags

bhs {address} [branch if higher or same, **C=0**]	**unsigned**
lbhs {address} [long branch if higher or same, **C=0**]	**unsigned**
blo {address} [branch if lower, **C=1**]	**unsigned**
lblo {address} [long branch if lower, **C=1**]	**unsigned**
bhi {address} [branch if higher]	**unsigned**
lbhi {address} [long branch if higher]	**unsigned**
bls {address} [branch if less than or same]	**unsigned**
lbls {address} [long branch if less than or same]	**unsigned**
blt {address} [branch if less than, **N XOR V=1**]	**signed**
lblt {address} [long branch if less than, **N XOR V=1**]	**signed**
ble {address} [branch if less than or equal, **Z=1 or N XOR V=1**]	**signed**
lble {address} [long branch if less than or equal]	**signed**
bgt {address} [branch if greater than, **N XOR V=0**]	**signed**
lbgt {address} [long branch if greater than, **N XOR V=0**]	**signed**
bge {address} [branch if greater than or equal, **Z=1 or N XOR V=0**]	**signed**
lbge {address} [long branch if greater than or equal to]	**signed**

Branches based on a CPU Condition Code

bne {address} [branch if not equal]	**Z=0**
lbne {address} [long branch if not equal]	**Z=0**
beq {address} [branch if equal]	**Z=1**
lbeq {address} [long branch if equal]	**Z=1**
bcc {address} [branch if carry is clear]	**C=0**
lbcc {address} [long branch if carry is clear]	**C=0**
bcs {address} [branch if carry is set]	**C=1**
lbcs {address} [long branch if carry is set]	**C=1**
bmi {address} [branch if minus]	**N=1**
lbmi {address} [long branch if minus]	**N=1**
bpl {address} [branch if plus]	**N=0**
lbpl {address} [long branch if plus]	**N=0**
bvc {address} [branch if no overflow]	**V=0**
lbvc {address} [long branch if no overflow]	**V=0**
bvs {address} [branch if overflow]	**V=1**
lbvs {address} [long branch if overflow]	**V=1**

Operands

When a direct value is expected by an instruction

#%010101 [binary value]
#100 [decimal value]
#$7F [hexidecimal value]
#symbol_name [use symbol's equate]
#expression

When memory access is expected

%address [binary address]
$address [hexidecimal address]
symbol_name [use symbol's equate]
address [decimal address]
<address [LSB of address, reg. **dp** is the MSB]
>address [full 16-bit address]

When a string or character is expected

“a string”
/a string/
'c a character
'b' a character

Indexed memory

,{**x**,**y**,**u**,**s**,**pc**,***w***} (access memory pointed to by reg.)
[,{**x**,**y**,**u**,**s**,**pc**,***w***}**]** (indirect access)
{**a**,**b**,**d**,***e***,***f***,***w***},{**x,y,u,s,pc,*w***}
[address**]** (indirect address)
offset,{**x**,**y**,**u**,**s**,**pc**,***w***} (use 5-bit offset from pointer if possible)
<offset,{**x**,**y**,**u**,**s**,**pc**,***w***} (force 8-bit offset from pointer if possible)
>offset,{**x**,**y**,**u**,**s**,**pc**,***w***} (force 16-bit offset from pointer if possible)

typical examples of indexed memory access:

,**x**	offset,**x**	,**x**+	,**x**++	,-**x**	,--**x**
a,**x**	**b**,**x**	**d**,**x**	***e***,**x**	***f***,**x**	***w***,**x**
,**y**	offset,**y**	,**y**+	,**y**++	,-**y**	,--**y**
a,**y**	**b**,**y**	**d**,**y**	***e***,**y**	***f***,**y**	***w***,**y**
,**u**	offset,**u**	,**u**+	,**u**++	,-**u**	,--**u**
a,**u**	**b**,**u**	**d**,**u**	***e***,**u**	***f***,**u**	***w***,**u**
,**s**	offset,**s**	,**s**+	,**s**++	,-**s**	,--**s**
a,**s**	**b**,**s**	**d**,**s**	***e***,**s**	***f***,**s**	***w***,**s**
,***w***	offset,***w***	,***w***++,--***w***		,**pc**	offset,**pc**
[,**x**]	[offset,**x**]	[,**x**++]	[,--**x**]		
[**a**,**x**]	[**b**,**x**]	[**d**,**x**]	[***e***,**x**]	[***f***,**x**]	[***w***,**x**]
[,**y**]	[offset,**y**]	[,**y**++]	[,--**y**]		
[**a**,**y**]	[**b**,**y**]	[**d**,**y**]	[***e***,**y**]	[***f***,**y**]	[***w***,**y**]
[,**u**]	[offset,**u**]	[,**u**++]	[,--**u**]		
[**a**,**u**]	[**b**,**u**]	[**d**,**u**]	[***e***,**u**]	[***f***,**u**]	[***w***,**u**]
[,**s**]	[offset,**s**]	[,**s**++]	[,--**s**]		
[**a**,**s**]	[**b**,**s**]	[**d**,**s**]	[***e***,**s**]	[***f***,**s**]	[***w***,**s**]
[,***w***]	[offset,***w***]	[,***w***++]	[,--***w***]	[,**pc**]	[offset,**pc**]

Indexed memory using 6309 AIM, TIM, EIM, OIM instructions

#100;**5,x**
#65;**a,y**

Expressions

Values, offsets, addresses, and any other type of parameter may be defined as simple or complex mathematical expressions.

Operators

***** [multiply]
/ [divide]
% [modulas]
+ [add] (also unary)
- [subtract] (also unary)
^ [1's complement, logical NOT] (also unary)
& [logical AND]
! [logical OR]
| [logical OR]
~ [logical Exclusive OR]

Comparisons

The result of these operations will be of the Boolean type (either 0 for False or 1 for True). You compare mathematical expressions on either side of the operation, and get a True or False result.

= [is equal to]
< [is less than]
> [is greater than]
< [is less than or equal to]
> [is greater than or equal to]
<> [is not equal to]

Order Of Operations

1) parenthesis (innermost **(**first**)**)
2) unaries (like '**-**', '**+**', and '**^**')
2) multiplies and divides (*****, **/**, **%**)
3) adds and subtracts (**+**, **-**)
4) logical operations (&, !, ~, ^)
5) comparisons (**=**, **<**, **>**, **<>**, **<=**, **>=**)

You can always use parenthesis to control the order or to enhance the clarity of an expression.

Expression Examples

-64
+101
100**+**5
-symbol_5
$2000**+**$100
$3120-$ab
-255**<=**254
timercount**>**3600
symbol**=**anothersymbol
label**<>**anotherlabel
^255
label_c**+^**5
^symbol [return 1's complement of "symbol"]
port**!**enableDAC [return both values OR'ed into one value]
sample**&**%11111100 [mask out the lower 2 bits of "sample"]
%11111**%**%1000 [1st binary value modulas the 2nd binary value)
50*4/2
1+2*(3+4)+5 ; notice the order of operations (1 + 2*7 + 5 = 20)
(1024+32)*15+31
(52**-**2)*****2
+-5
-(+5)
-100/5*2 ; automatically orders as -(100/(5*2))
100+-100/10
apple+200/2 ; return ("apple" plus 100)
1*2+3*4+5*6
-254**<=**255
1000**>**-1000
-2000**>**2000
true**&**true ; returns true if both cases are true
true**&**false
false**&**true
false**&**false
true**!**true ; returns true if either case is true
true**!**false
false**!**true
false**!**false

See the Portal-9 or Rainbow IDE 'TESTS' project for many more examples of CCASM's powerful expression evaluator.

Structures, Unions, and Namespaces

Structures

A CCASM structure is a segment of data or code separated into fields or offsets from the structure beginning. By using the format "structurename.structurefield" you can access any field of any structure. These fields translate into their own offset from the beginning of the structure.

An example of a simple structure is:

```
color   struct
red     rmb  1
green   rmb  1
blue    rmb  1
        ends
```

To access the "green" field, you would reference the symbol "color.green".

Database applications can rely heavily on structures. Using pointers to objects, you can access records by name and field fairly easily in a large table or database. Because each structure field is an offset, it can be used as the offset for indexed memory instructions or anywhere else an offset is expected.

```
        ldx   #colors          start of database memory
        ldy   #256             records in database
a@      lda   color.green,x    load "green" field of this record
        ldb   color.blue,xload "blue" field of this record
        lde   color.red,x      load "red" field of this record
        jsr   plot
        leax  3,x              point to next record (skip structure size)
        leay  -1,y
        bne   a@
```

To automatically compute the size of a structure, use the following compile-time symbol:

example:

```
        ldy   #sizeof{color}
        ldx   #sizeof{transaction}
```

To declare a structure that inherits the fields of another structure, and possibly appends new fields to the new structure, the following syntax is used:

```
apple       structfruit
diameter    rmb         1
            ends
```

To generate data in the code stream (like FCB, FDB, FCC, etc. does) based on a structure, use the syntax:

```
label  apple
```

The label is required, and the mnemonic (psuedo-op) is whatever the structure name is. The above example generates initialized data the size of the source structure (apple).

Note that label inherits all of apple's structure fields. You can now directly access this data *area* using direct and exctended addressing.

```
start       lda    label.diameter     actual address of field
```

Unions

A union structure allows overlapping objects or data fields. The program counter does not advance as usual inside of a union structure for each object. The total size of a union is the size of the largest object in the union. Ending a union causes the program counter to advance by the size of the union (the largest object inside).

```
variant     union
byte        rmb   1
word        rmb   2
            endu
```

To automatically compute the size of a union, use the following compile-time symbol:

example:

```
            ldy    #sizeof{variant}
```

It's beyond the scope of this document to go into detail about all of the uses for union structures, but several uses will be mentioned briefly.

1) allows variable name aliasing
2) allows the reuse of variable memory by placing all union symbols at the same PC address
3) allows different data types to exist at the same location

A named union inside of a parent structure will cause all of its fields to take on the form *parent_structure.union_name.union_field*. You may optionally wish to use another method for accessing the union.

CCASM also supports anonymous unions. Anonymous (unnamed) unions must be declared within a structure. Because the union resides inside of a named structure, no name for the union is required. The resulting dot notation name for the union fields will be *parent_structure.union_field.*

Namespaces

Using the *namespace* directive, a constant prefix label will be assigned to all subsequent labels; thus, allowing composite labels to be formed. This feature might come in handy more when you are attempting to merge or include foreign source code into your programs.

```
foo         namespace
start       rts
            endname          close namespace

            jmp   foo.start
```

Procedures

Introduction to CCASM Procedures

Procedures in assembly language? Ofcourse! You can create procedures that use formal parameters, then call your procedures along with the required parameters. Code generation and stack management is handled automatically.

Procedures are declared using the ***proc/begin/endproc*** directives. The ***proc*** directive is required to name the procedure and list the required parameters and their types. Procedures are ended using the ***endproc*** directive.

Declaring Procedures

```
fillmem     proc  top:word,length:word,filler:byte
            begin fillmem
            ldx   top,u       get parameter
            ldy   length,u    get parameter
            lda   filler,u    get parameter
a@          sta   ,x+
            leay  -1,y
            bne   a@
            endproc
```

The first required field is the procedure name ('fillmen' in this example). The second field (always called ***proc***) is also required. The third field is optional and lists any parameters required by the procedure. Procedures do not have to have parameters. Then why use a procedure instead of the ***jsr*** instruction? Procedures can reserve local named variables on the stack automatically. This helps isolate your procedures or subroutines from the rest of the program.

The ***begin*** directive marks the entry point into your procedure. This allows static and local memory to be reserved between the ***proc*** and ***begin*** directives. Static memory will be placed at the current program counter inside of the procedure while local memory gets allocated on the stack at run-time. The code for this is generated automatically by the assembler.

You define formal parameters by listing any number of symbol names along with their types (such as byte, word, dword, int8, int16, etc.). The format is ***symbol:type,symbol:type,...*** for as many parameters as you need.

Note: No spaces are allowed in a procedures's parameter list.

The following parameter list defines 5 bytes used by the procedure, composed of two 16-bit values and one 8-bit value.

top:word,length:word,filler:byte

Calling Procedures

After defining a procedure, it's ready to call using the ***call*** function. When you call a procedure, you must pass the same number of parameters into the procedure that are defined in the formal parameter list. However, the names or values you pass in are separate (outside) objects. This information is copied into the formal parameter names used only by the procedure.

Here's an example of how we would call the fillmem procedure:

```
            org   3584

start       call  fillmem,1024,512,128
            rts

            end   start
```

Here's what happens when the ***call*** function is invoked:

First, the supplied actual parameters (1024, 512, 128) are pushed onto the S stack starting from the last parameter (128) and ending with the first parameter (1024). The above example pushes the following parameters onto the S stack in the order of *byte, word, word*. The parameters are pushed onto the S stack automatically (at run-time) using code generated by the assembler (at compile-time).

The parameter values that are pushed onto the S stack occupy the same number of bytes as the formal parameter's type states. If you try to pass in a 16-bit value for an 8-bit formal parameter, only the LSB of the parameter will be passed to the procedure.

Inside of Procedures

So, what goes on inside of a procedure? The quick answer is: anything you like! The other answer explains what is generated by the assembler to make the procedure do what it is supposed to do.

First there is a small bit of automatic code that finishes creating the procedure's *activation record* (stack frame).

The previous activation record pointer (,U) is pushed to the S stack, then the current value of the S stack is copied to the U register so that parameters and local memory can be accessed as offsets from ,U. This is the base address of the procedure's activation record. Parameters are accessed from positive sides of ,U while local memory is accessed from negative sides of ,U. As long as we preserve the U register during the procedure, everything is ok. However, if there's no parameters or local variables, you can use U for whatever you like.

Now the S stack is moved down in memory one byte for each byte of local memory required by the procedure. This stack adjustment is done using one instruction which subtracts the total local memory requirement from the value of the S stack.

Inside of a procedure, the current location of the S stack base is not that important. In other words, since ,U now points to the activation record which also holds information used to restore the S stack to where it was before the procedure call, you can use S to play around with some. However, be careful not to destroy anything on the plus side of the stack since there's likely to be an activation record (or more) sitting there at any given time.

At this time, there is currently no “display”. All labels and symbols are local to the procedure, meaning you can't access any symbols that were defined outside of the procedure.

Accessing Procedure Parameters

You can access the parameters that the ***call*** function passed in by using the following syntax:

```
lda   parameter1,u      normal
ldd   parameter2,u      normal
ldx   [parameter3,u]    indirect (pass-by-reference support)
lda   parameter1+1,u    offset of parameter + 1
```

Simple enough, all procedure *parameters* are accessed as offsets from the U register. That is, the parameter values are pushed onto the S stack before the procedure is called, then the U register is pointed to this base pointer of the S stack.

The assembler automatically computes parameter offsets, so you don't have to really worry too much about where your data is on the stack. Just use the formal parameter name (defined in the procedure declaration) and append the ",U" indexed register.

You can also place static data (RMBs, FCB's, FCC's, etc.) inside of your procedures.

Local Variables

You can reserve local variables inside of a procedure by using the ***var*** directive, like so:

```
fillmem     proc  start:word,length:word,filler:byte
aa          var   1       reserve 1 byte of local memory
bb          var   2       reserve 2 bytes of local memory
            begin fillmem
            ...
            lda   aa,u    access local mem
            ldd   bb,u    access local mem
            endproc
```

You access local variables the same way you access procedure parameters, using the ,U indexing mode. Local memory is accessed on the negative side of ,U while parameters are accessed on the positive side of ,U. For example:

```
lda   local,u     translates to lda -offset,u
lda   param,u     translates to lda offset,u
```

The offsets for both parameters and local variables are automatically computed at compile-time. These offsets into the procedure's activation record will be explained next.

Procedure Activation Records

Every procedure has an activation record that is created at run-time and stored on the S stack. The code for creating the activation record is generated by the assembler automatically, based on a procedure's optional parameters and local variables, etc. A procedure with both parameters and local variables will have an activation record similar to the one below. Note that {address} is given as an example of where the S stack was originally at (32768) before the procedure call.

Local Variable 2 (MSB)	{32759} +0,s	-3,u
Local Variable 2 (LSB)	{32760} +1,s	-2,u
Local Variable 1 (byte)	{32761} +2,s	-1,u
Register U MSB	{32762} +3,s	<-- Record Base (,u)
Register U LSB	{32763} +4,s	1,u
Program Counter MSB	{32764} +5,s	2,u
Program Counter LSB	{32765} +6,s	3,u
Parameter2 (LEVEL)	{32766} +7,s	4,u
Parameter1 (COLOR)	{32767} +8,s	5,u

A procedure having parameters but no local variables will have an activation record similar to the one below.

Register U MSB	{32762} +0,s	<-- Record Base (,u)
Register U LSB	{32763} +1,s	1,u
Program Counter MSB	{32764} +2,s	2,u
Program Counter LSB	{32765} +3,s	3,u
Parameter2 (LEVEL)	{32766} +4,s	4,u
Parameter1 (COLOR)	{32767} +5,s	5,u

A procedure having no parameters and no local variables will have an activation record similar to the one below. Note that this is basically a pointless activation record unless you plan to do some manual allocation of local memory, etc. by adjusting the S stack yourself from within the procedure.

Register U MSB	{32762} +0,s	<-- Record Base (,u)
Register U LSB	{32763} +1,s	1,u
Program Counter MSB	{32764} +2,s	2,u
Program Counter LSB	{32765} +3,s	3,u

Instruction Examples

6809 Examples

orcc #80 [disable IRQ and FIRQ interrupts]
andcc #175 [enable IRQ and FIRQ interrupts]
orcc #%00000001 [manually set the Carry conditon code]
andcc #%11111110 [manually clear the Carry condition code]
pshs x,d [push reg. **x**, reg. **b**, and reg. **a** onto S stack]
puls d,x,pc [pull regs. from stack then simulate an rts]
leay -1**,y** [subtract 1 from reg. y]
leau 2,**x** [load reg.x + 2 into reg.u]
leax d,x [reg. **x** = reg. **x** + reg. **d**]
leax table,**pc** [load relative address of "table" into reg. **x**]
here **equ *** ['*' translates into the address where "here" is or will be]
fdb 1024,**.** ['.' translates into the *address of* the 2nd operand value]
fcc "this is a basic ASCII string"
fcn "this string automatically gets a NULL added to it!"
fcs "this is a bit7-terminated ASCII string"
fcr "this string automatically gets a CR+NULL added to it"
fcb 1,2,3,4,5 [store 5 8-bit values]
fdb 10,20,30 [store 3 16-bit values]
fqb 5,10,15,20 [store 4 32-bit values]
rmb 200 [reserve/void 200 bytes of memory, for use at run-time]
lda ,x [get data at address pointed to by reg. **x**]
lda [,x] [get data at address pointed to by address in reg. **x**]
lda -5,u [get data at 5 bytes above address in reg. **u**]
adca #0 [add Carry result (0 or 1) into reg. **a**]
adcb #10 [add Carry result plus 10 into reg. **b**]
asrb [divide the *signed* contents of reg. **b** by 2]
lsrb [divide the *unsigned* contents of reg. **a** by 2]
rora [done consecutively, 9-bit right rotation is possible]
rola [9-bit left rotation through the Carry condition code]

6309-Only Examples

ldmd #1 [enable full 6309 CPU operation mode]
sexw [converts signed reg. ***w*** into signed reg. ***q***]
oim 64;1024 [OR the value 64 into address 1024]
oim 128;**,u** [OR the value 128 into the memory pointed to by reg. **u**]
aim 254;2**,u** [AND the value 254 into offsetted mem. pointed to by reg. **u**]
aim 191;1024 [AND the value 191 into address 1024]
tim $80;65280 [TEST bit #7 of address 65280]
tim %11;[1000] [TEST bits #0&1 of indirect address 1000]
eim 85;255 [XOR the value 85 into address 255]
bor **a**,1,7,255 [OR bit #1 in reg. **a** with bit #7 from address 255]
ldbt **a**,2,6,200 [load bit #2 in reg. **a** with bit #6 from address 200]
ldq #98765 [load reg. ***q*** with a 32-bit integer]
ldq #$A4B2C3D9 [load reg. ***q*** with a 32-bit hex. value]
ldq #%10110010110000111010100011101011 [32-bit binary value]

Sample Program

This program prints a message to your Color BASIC screen:

```
          org       16384     run at this address
start     leax      msg,pcr   point to our message
!         lda       ,x+       get ASCII byte in msg
          beq       done      stop at null byte
          jsr       [40962]   print using BASIC ROM's STDOUT
          bra       <         loop back to "!"
done      rts                 return to BASIC
msg       fcn       "HELLO WORLD"
          end       start     set BASIC "EXEC" address
```

This program echos your keystrokes to the Color BASIC screen (hit <BREAK> to exit):

```
          org       16384     run at this address
getkey    jsr       [40960]   get key from BASIC ROM's STDIN
          tsta                is it a NULL character?
          beq       getkey    yes, ignore it
          cmpa      #3        is it the BREAK key?
          beq       done2     yes, so exit
          jsr       [40962]   no, so print the char to STDOUT
          bra       getkey    keep checking keys
done2     rts                 return to BASIC
          end       getkey    set BASIC "EXEC" address
```

This program clears the Color BASIC screen:

```
          org       16384     run at this address
filler    equ       $6060     "filler = $6060"
cls       ldx       #1024     point to top of screen
          ldy       #512      set # of bytes to clear
          ldd       #filler   use 2 bytes of $60
!         std       ,x++      clear the 2 characters
          leay      -2,y      subtract them from count
          bne       <         count not 0, so repeat
          rts                 return to BASIC
          end       cls
```

This example combines the above routines into one program:

```
          org       16384     run at this address
start     ldx       #1024     point to top of screen
          ldy       #512      set # of bytes to clear
          ldd       #$6060    use 2 blank characters
!         std       ,x++      clear the 2 characters
          leay      -2,y      subtract them from count
          bne       <         go back to "!" until count=0
          leax      msg,pcr   point to our message
!         lda       ,x+       get ASCII byte in msg
          beq       getkey    stop at null byte
          jsr       [40962]   print using BASIC ROM
          bra       <         loop back to "!"
getkey    jsr       [40960]   get keystroke using BASIC ROM
          tsta                is it a NULL character?
          beq       getkey    yes, ignore it
          cmpa      #3        is it the BREAK key?
          beq       done      yes, so exit
          jsr       [40962]   no, so print the character
          bra       getkey    keep checking keys
done      rts                 return to BASIC
msg       fcr       "HELLO WORLD OF ASSEMBLY"
          end       start
```

File Formats

Multi-record files:

1) are created automatically based on the structure of your source code
2) can be LOADMed by Disk BASIC or similar loaders
3) have a beginning ORG record defining where the code should loading into RAM
4) have subsequent ORG records causing the loader to jump somewhere else
5) have an END record signifying there are no more records

This type of file can contain sub origins and any mix of voided memory, etc. An example of a multi-record file would be one that has the ability to load 3 different programs into 3 different locations of RAM, all done by the loader based on information found in the embedded records. Another example would be a program that automatically executes after being loaded, by embedding a small segment of code that overwrites a system area of Disk BASIC.

Single-record files:

1) are created automatically based on the structure of your source code
2) can be LOADMed by Disk BASIC or similar loaders
3) have a beginning LOAD record defining where the code should loading into RAM
4) have an END record signifying there are no more records

An example of a single-record binary file would be a file created by BASIC after typing SAVEM "SCREEN",1024,1535,0. The resulting file would 522 bytes long because a 5-byte LOAD record begins, then 512 bytes of screen data, then a 5-byte END record.

You can also force a single-record file output (-sr option) which has an additional effect of translating any RMB statements in your source into initialized data (rather than voided memory).

Because of the translation of voided memory areas into initialized data, a continuous stream of code is generated from the first ORG statement to the END statement of your source code. No other embedded ORG statements should be used in your source code that will be assembled in single-record format.

No-records files:

1) must be force-assembled using the -nr option
2) are similar to ROM images
3) have no beginning or subsequent ORG records
4) have no END record

This type of file can be viewed as a variable-sized ROM image where the file consists of only program opcode or data and no loader control structures. Such ROM-like files must be structured correctly before assembly. Multiple ORG statements are allowed in the *source code*, but should be used very carefully. No opcode or initialized data should be placed after any RMB statement in a program to be assembled in no-records format. In other words, voided memory is not assembled, because a record is not generated to tell the loader to advance past or load around any voided memory.

Multiple ORG statments followed by sets of RMBs are generally used for enumerating variable addresses, etc. Large buffers and uninitialized tables and can also be reserved this way so long as no opcode or data appears after any RMB statements. Doing so would cause those stray opcodes to be loaded into unintended locations in RAM.

6809 Opcode Summary

Mnemon.	Op	IHNZVC	IEXD#R	~	Description	Notes
ABX	3A	------	X	3	Add to Index Register	X=X+B
ADCa s	B9	-*****	XXXXX	5	Add with Carry	a=a+s+C
ADDa s	BB	-*****	XXXXX	5	Add	a=a+s
ADDD s	F3	-*****	XXX*X	7	Add to Double acc.	D=D+s
ANDa s	B4	--**0-	XXXXX	5	Logical AND	a=a&s
ANDCC s	1C	?????1	X	3	Logical AND with CCR	CC=CC&s
ASL d	78	--****	XXX X	7	Arithmetic Shift Left	d=d*2
ASLa	48	--****	X	2	Arithmetic Shift Left	a=a*2
ASR d	77	--****	XXX X	7	Arithmetic Shift Right	d=d/2
ASRa	47	--****	X	2	Arithmetic Shift Right	a=a/2
BCC m	24	------	x	3	Branch if Carry Clear	If C=0
BCS m	25	------	x	3	Branch if Carry Set	If C=1
BEQ m	27	------	x	3	Branch if Equal	If Z=1
BGE m	2C	------	x	3	Branch if Great/Equal	If NxV=0
BGT m	2E	------	x	3	Branch if Greater Than	If Zv{NxV}=0
BHI m	22	------	x	3	Branch if Higher	If CvZ=0
BHS m	24	------	x	3	Branch if Higher/Same	If C=0
BITa s	B5	--**0-	XXXXX	5	Bit Test accumulator	a&s
BLE m	2F	------	x	3	Branch if Less/Equal	If Zv{NxV}=1
BLO m	25	------	x	3	Branch if Lower	If C=1
BLS m	23	------	x	3	Branch if Lower/Same	If CvZ=1
BLT m	2D	------	x	3	Branch if Less Than	If NxV=1
BMI m	2B	------	x	3	Branch if Minus	If N=1
BNE m	26	------	x	3	Branch if Not Equal	If Z=0
BPL m	2A	------	x	3	Branch if Plus	If N=0
BRA m	20	------	x	3	Branch Always	PC=m
BRN m	21	------	x	3	Branch Never	NOP
BSR m	8D	------	x	7	Branch to Subroutine	-[S]=PC,BRA
BVC m	28	------	x	3	Branch if Overflow Clr	If V=0
BVS m	29	------	x	3	Branch if Overflow Set	If V=1
CLR d	7F	--0100	XXX X	7	Clear	d=0
CLRa	4F	--0100	X	2	Clear accumulator	a=0
CMPa s	B1	--****	XXXXX	5	Compare	a-s
CMPD s	B3	--****	XXX*X	8	Compare Double acc.	D-s (10H)
CMPS s	BC	--****	XXX*X	8	Compare Stack pointer	S-s (11H)
CMPU s	B3	--****	XXX*X	8	Compare User stack ptr	U-s (11H)
CMPi s	BC	--****	XXX*X	7	Compare	i-s (Y ~s=8)
COM d	73	--**01	XXX X	2	Complement	d=~d
COMa	43	--**01	X	7	Complement accumulator	a=~a
CWAI n	3C	E?????	X	K	AND CCR, Wait for int.	CC=CC&n,E=1,
DAA	19	--****	X	2	Decimal Adjust Acc.	A=BCD format
DEC d	7A	--***-	XXX X	7	Decrement	d=d-1
DECa	4A	--***-	X	2	Decrement accumulator	a=a-1
EORa s	B8	--**0-	XXXXX	5	Logical Exclusive OR	a=axs
EXG r,r	1E	------	X	8	Exchange (r1 size=r2)	r1<->r2
INC d	7C	--***-	XXX X	7	Increment	d=d+1
INCa	4C	--***-	X	2	Increment accumulator	a=a+1
JMP s	7E	------	XXX X	4	Jump	PC=EAs

6809 Opcode Summary (cont.)

Mnemon.	Op	IHNZVC	IEXD#R	~	Description	Notes
JSR s	BD	------	XXX X	8	Jump to Subroutine	-[S]=PC,JMP
LBcc nn	10	------	x	5	Long cond. Branch(~=6)	If cc LBRA
LBRA nn	16	------	x	5	Long Branch Always	PC=nn
LBSR nn	17	------	x	9	Long Branch Subroutine	-[S]=PC,LBRA
LDa s	B6	--**0-	XXXXX	5	Load accumulator	a=s
LDD s	FC	--**0-	XXX*X	6	Load Double acc.	D=s
LDS s	FE	--**0-	XXX*X	7	Load Stack pointer	S=s (10H)
LDU s	FE	--**0-	XXX*X	6	Load User stack ptr	U=s
LDi s	BE	--**0-	XXX*X	6	Load index register	i=s (Y ~s=7)
LEAp s	3X	---i--	xX X	4	Load Effective Address	p=EAs(X=0-3)
LSL d	78	--0***	XXX X	7	Logical Shift Left	d={C,d,0}<-
LSLa	48	--0***	X	2	Logical Shift Left	a={C,a,0}<-
LSR d	74	--0***	XXX X	7	Logical Shift Right	d=->{C,d,0}
LSRa	44	--0***	X	2	Logical Shift Right	d=->{C,d,0}
MUL	3D	---*-*	X	B	Multiply	D=A*B
NEG d	70	-?****	XXX X	7	Negate	d=-d
NEGa	40	-?****	X	2	Negate accumulator	a=-a
NOP	12	------	X	2	No Operation	
ORa s	BA	--**0-	XXXXX	5	Logical inclusive OR	a=avs
ORCC n	1A	??????	X	3	Inclusive OR CCR	CC=CCvn
PSHS r	34	------	X	2	Push reg(s) (not S)	-[S]={r,...}
PSHU r	36	------	X	2	Push reg(s) (not U)	-[U]={r,...}
PULS r	35	??????	X	2	Pull reg(s) (not S)	{r,...}=[S]+
PULU r	37	??????	X	2	Pull reg(s) (not U)	{r,...}=[U]+
ROL d	79	--****	XXX X	7	Rotate Left	d={C,d}<-
ROLa	49	--****	X	2	Rotate Left acc.	a={C,a}<-
ROR d	76	--****	XXX X	7	Rotate Right	d=->{C,d}
RORa	46	--****	X	2	Rotate Right acc.	a=->{C,a}
RTI	3B	-*****	X	6	Return from Interrupt	{regs}=[S]+
RTS	39	------	X	5	Return from Subroutine	PC=[S]+
SBCa s	B2	--****	XXXXX	5	Subtract with Carry	a=a-s-C
SEX	1D	--**--	X	2	Sign Extend	D=B
STa d	B7	--**0-	XXX X	5	Store accumultor	d=a
STD d	FD	--**0-	XXX X	6	Store Double acc.	D=a
STS d	FF	--**0-	XXX X	7	Store Stack pointer	S=a (10H)
STU d	FF	--**0-	XXX X	6	Store User stack ptr	U=a
STi d	BF	--**0-	XXX X	6	Store index register	i=a (Y ~s=7)
SUBa s	B0	--****	XXXXX	5	Subtract	a=a-s
SUBD s	B3	--****	XXX*X	7	Subtract Double acc.	D=D-s
SWI	3F	1-----	X	J	Software Interrupt 1	-[S]={regs}
SWI2	3F	E-----	X	K	Software Interrupt 2	SWI (10H)
SWI3	3F	E-----	X	K	Software Interrupt 3	SWI (11H)
SYNC	13	------	X	2	Sync. to interrupt	(min ~s=2)
TFR r,r	1F	------	X	6	Transfer (r1 size<=r2)	r2=r1
TST s	7D	--**0-	XXX X	7	Test	s
TSTa	4D	--**0-	X	2	Test accumulator	a

6809 Opcode Summary (cont.)

```
-------------------------------------------------------------
| CCR      |-*01? |      | |Unaffect/affected/reset/set/unknown|
| E        |E     |      | |Entire flag (Bit 7, if set RTI~s=F)|
| F  I     |I     |      | |FIRQ/IRQ interrupt mask (Bit 6/4)  |
| H        | H    |      | |Half carry (Bit 5)                 |
| N        |  N   |      | |Negative (Bit 3)                   |
| Z        |   Z  |      | |Zero (Bit 2)                       |
| V        |    V |      | |Overflow (Bit 1)                   |
| C        |     C|      | |Carry/borrow (Bit 0)               |
|----------------+------+-----------------------------------|
| a              |I     | |Inherent (a=A,Op=4XH, a=B,Op=5XH)  |
| nn,E           | E    | |Extended (Op=E, ~s=e)              |
| [nn]           | x    | |Extended indirect                  |
| xx,p!          |  X   | |Indexed (Op=E-10H, ~s=e-1)         |
| [xx,p!]        |  X   | |Indexed indirect (p!=p++,--p only) |
| n,D            |   D  | |Direct (Op=E-20H, ~s=e-1)          |
| #n             |    # | |Immediate (8-bit, Op=E-30H, ~s=e-3)|
| #nn            |    * | |Immediate (16-bit)                 |
| m              |     x| |Relative (PC=PC+2+offset)          |
| [m]            |     R| |Relative indirect (ditto)          |
|------------------------+-----------------------------------|
|DIRECT                  |Direct addressing mode             |
|EXTEND                  |Extended addressing mode           |
|FCB      n              |Form Constant Byte                 |
|FCC      'string'       |Form Constant Characters           |
|FDB      nn             |Form Double Byte                   |
|RMB      nn             |Reserve Memory Bytes               |
|------------------------+-----------------------------------|
| A  B                   |Accumulators (8-bit)               |
| CC                     |Condition Code register (8-bit)    |
| D                      |A and B (16-bit, A high, B low)    |
| DP                     |Direct Page register (8-bit)       |
| PC                     |Program Counter (16-bit)           |
| S  U                   |System/User stack pointer(16-bit)  |
| X  Y                   |Index registers (16-bit)           |
|------------------------+-----------------------------------|
| a                      |Acc A or B (a=A,Op=BXH, a=B,Op=FXH)|
| d  s  EA               |Destination/source/effective addr. |
| i  p  r                |Regs X,Y/regs X,Y,S,U/any register |
| m                      |Relative address (-126 to +129)    |
| n  nn                  |8/16-bit expression(0 to 255/65535)|
| xx  p!                 |A,B,D,nn/p+,-p,p++,--p (indexed)   |
| +  -  *  /             |Add/subtract/multiply/divide       |
| &  ~  v  x             |AND/NOT/inclusive OR/exclusive OR  |
| <-  ->  <->            |Rotate left/rotate right/exchange  |
| [ ]  [ ]+  -[ ]        |Indirect address/increment/decr.   |
| { }                    |Combination of operands            |
| {regs}                 |If E {PC,U/S,Y,X,DP,B,A,CC}/{PC,CC}|
| (10H)  (11H)           |Hex opcode to precede main opcode  |
-------------------------------------------------------------
```

Hexidecimal, Binary, and Decimal Conversions

Use this chart to translate values between the different number types accepted by CCASM. You can use any number base system you prefer when writing software -- hexidecimal (base 16), binary (base 2), or decimal (base 10).

```
Hex   Bin          Dec   Neg   ASCII
-------------------------------------
$00 = %00000000 = 0
$01 = %00000001 = 1   = -255
$02 = %00000010 = 2   = -254
$03 = %00000011 = 3   = -253
$04 = %00000100 = 4   = -252
$05 = %00000101 = 5   = -251
$06 = %00000110 = 6   = -250
$07 = %00000111 = 7   = -249 = Bell
$08 = %00001000 = 8   = -248 = Backspace
$09 = %00001001 = 9   = -247 = TAB
$0A = %00001010 = 10  = -246 = Line Feed
$0B = %00001011 = 11  = -245
$0C = %00001100 = 12  = -244 = Form Feed/Clear
$0D = %00001101 = 13  = -243 = Carriage Return
$0E = %00001110 = 14  = -242
$0F = %00001111 = 15  = -241
$10 = %00010000 = 16  = -240
$11 = %00010001 = 17  = -239
$12 = %00010010 = 18  = -238
$13 = %00010011 = 19  = -237
$14 = %00010100 = 20  = -236
$15 = %00010101 = 21  = -235
$16 = %00010110 = 22  = -234
$17 = %00010111 = 23  = -233
$18 = %00011000 = 24  = -232
$19 = %00011001 = 25  = -231
$1A = %00011010 = 26  = -230
$1B = %00011011 = 27  = -229
$1C = %00011100 = 28  = -228
$1D = %00011101 = 29  = -227
$1E = %00011110 = 30  = -226
$1F = %00011111 = 31  = -225
$20 = %00100000 = 32  = -224 = '
$21 = %00100001 = 33  = -223 = '!
$22 = %00100010 = 34  = -222 = '"
$23 = %00100011 = 35  = -221 = '#
```

```
$24 = %00100100 = 36  = -220 = '$
$25 = %00100101 = 37  = -219 = '%
$26 = %00100110 = 38  = -218 = '&
$27 = %00100111 = 39  = -217 = ''
$28 = %00101000 = 40  = -216 = '(
$29 = %00101001 = 41  = -215 = ')
$2A = %00101010 = 42  = -214 = '*
$2B = %00101011 = 43  = -213 = '+
$2C = %00101100 = 44  = -212 = ',
$2D = %00101101 = 45  = -211 = '-
$2E = %00101110 = 46  = -210 = '.
$2F = %00101111 = 47  = -209 = '/
$30 = %00110000 = 48  = -208 = '0
$31 = %00110001 = 49  = -207 = '1
$32 = %00110010 = 50  = -206 = '2
$33 = %00110011 = 51  = -205 = '3
$34 = %00110100 = 52  = -204 = '4
$35 = %00110101 = 53  = -203 = '5
$36 = %00110110 = 54  = -202 = '6
$37 = %00110111 = 55  = -201 = '7
$38 = %00111000 = 56  = -200 = '8
$39 = %00111001 = 57  = -199 = '9
$3A = %00111010 = 58  = -198 = ':
$3B = %00111011 = 59  = -197 = ';
$3C = %00111100 = 60  = -196 = '<
$3D = %00111101 = 61  = -195 = '=
$3E = %00111110 = 62  = -194 = '>
$3F = %00111111 = 63  = -193 = '?
$40 = %01000000 = 64  = -192 = '@
$41 = %01000001 = 65  = -191 = 'A
$42 = %01000010 = 66  = -190 = 'B
$43 = %01000011 = 67  = -189 = 'C
$44 = %01000100 = 68  = -188 = 'D
$45 = %01000101 = 69  = -187 = 'E
$46 = %01000110 = 70  = -186 = 'F
$47 = %01000111 = 71  = -185 = 'G
$48 = %01001000 = 72  = -184 = 'H
$49 = %01001001 = 73  = -183 = 'I
$4A = %01001010 = 74  = -182 = 'J
$4B = %01001011 = 75  = -181 = 'K
$4C = %01001100 = 76  = -180 = 'L
$4D = %01001101 = 77  = -179 = 'M
$4E = %01001110 = 78  = -178 = 'N
$4F = %01001111 = 79  = -177 = 'O
```

```
$50 = %01010000 = 80  = -176 = 'P
$51 = %01010001 = 81  = -175 = 'Q
$52 = %01010010 = 82  = -174 = 'R
$53 = %01010011 = 83  = -173 = 'S
$54 = %01010100 = 84  = -172 = 'T
$55 = %01010101 = 85  = -171 = 'U
$56 = %01010110 = 86  = -170 = 'V
$57 = %01010111 = 87  = -169 = 'W
$58 = %01011000 = 88  = -168 = 'X
$59 = %01011001 = 89  = -167 = 'Y
$5A = %01011010 = 90  = -166 = 'Z
$5B = %01011011 = 91  = -165 = '[
$5C = %01011100 = 92  = -164 = '\
$5D = %01011101 = 93  = -163 = ']
$5E = %01011110 = 94  = -162 = '^
$5F = %01011111 = 95  = -161 = '_
$60 = %01100000 = 96  = -160 = '`
$61 = %01100001 = 97  = -159 = 'a
$62 = %01100010 = 98  = -158 = 'b
$63 = %01100011 = 99  = -157 = 'c
$64 = %01100100 = 100 = -156 = 'd
$65 = %01100101 = 101 = -155 = 'e
$66 = %01100110 = 102 = -154 = 'f
$67 = %01100111 = 103 = -153 = 'g
$68 = %01101000 = 104 = -152 = 'h
$69 = %01101001 = 105 = -151 = 'i
$6A = %01101010 = 106 = -150 = 'j
$6B = %01101011 = 107 = -149 = 'k
$6C = %01101100 = 108 = -148 = 'l
$6D = %01101101 = 109 = -147 = 'm
$6E = %01101110 = 110 = -146 = 'm
$6F = %01101111 = 111 = -145 = 'o
$70 = %01110000 = 112 = -144 = 'p
$71 = %01110001 = 113 = -143 = 'q
$72 = %01110010 = 114 = -142 = 'r
$73 = %01110011 = 115 = -141 = 's
$74 = %01110100 = 116 = -140 = 't
$75 = %01110101 = 117 = -139 = 'u
$76 = %01110110 = 118 = -138 = 'v
$77 = %01110111 = 119 = -137 = 'w
$78 = %01111000 = 120 = -136 = 'x
$79 = %01111001 = 121 = -135 = 'y
$7A = %01111010 = 122 = -134 = 'z
$7B = %01111011 = 123 = -133 = '{
```

```
$7C = %01111100 = 124 = -132 = '|
$7D = %01111101 = 125 = -131 = '}
$7E = %01111110 = 126 = -130 = '~
$7F = %01111111 = 127 = -129
$80 = %10000000 = 128 = -128
$81 = %10000001 = 129 = -127
$82 = %10000010 = 130 = -126
$83 = %10000011 = 131 = -125
$84 = %10000100 = 132 = -124
$85 = %10000101 = 133 = -123
$86 = %10000110 = 134 = -122
$87 = %10000111 = 135 = -121
$88 = %10001000 = 136 = -120
$89 = %10001001 = 137 = -119
$8A = %10001010 = 138 = -118
$8B = %10001011 = 139 = -117
$8C = %10001100 = 140 = -116
$8D = %10001101 = 141 = -115
$8E = %10001110 = 142 = -114
$8F = %10001111 = 143 = -113
$90 = %10010000 = 144 = -112
$91 = %10010001 = 145 = -111
$92 = %10010010 = 146 = -110
$93 = %10010011 = 147 = -109
$94 = %10010100 = 148 = -108
$95 = %10010101 = 149 = -107
$96 = %10010110 = 150 = -106
$97 = %10010111 = 151 = -105
$98 = %10011000 = 152 = -104
$99 = %10011001 = 153 = -103
$9A = %10011010 = 154 = -102
$9B = %10011011 = 155 = -101
$9C = %10011100 = 156 = -100
$9D = %10011101 = 157 = -99
$9E = %10011110 = 158 = -98
$9F = %10011111 = 159 = -97
$A0 = %10100000 = 160 = -96
$A1 = %10100001 = 161 = -95
$A2 = %10100010 = 162 = -94
$A3 = %10100011 = 163 = -93
$A4 = %10100100 = 164 = -92
$A5 = %10100101 = 165 = -91
$A6 = %10100110 = 166 = -90
$A7 = %10100111 = 167 = -89
```

$A8 = %10101000 = 168 = -88
$A9 = %10101001 = 169 = -87
$AA = %10101010 = 170 = -86
$AB = %10101011 = 171 = -85
$AC = %10101100 = 172 = -84
$AD = %10101101 = 173 = -83
$AE = %10101110 = 174 = -82
$AF = %10101111 = 175 = -81
$B0 = %10110000 = 176 = -80
$B1 = %10110001 = 177 = -79
$B2 = %10110010 = 178 = -78
$B3 = %10110011 = 179 = -77
$B4 = %10110100 = 180 = -76
$B5 = %10110101 = 181 = -75
$B6 = %10110110 = 182 = -74
$B7 = %10110111 = 183 = -73
$B8 = %10111000 = 184 = -72
$B9 = %10111001 = 185 = -71
$BA = %10111010 = 186 = -70
$BB = %10111011 = 187 = -69
$BC = %10111100 = 188 = -68
$BD = %10111101 = 189 = -67
$BE = %10111110 = 190 = -66
$BF = %10111111 = 191 = -65
$C0 = %11000000 = 192 = -64
$C1 = %11000001 = 193 = -63
$C2 = %11000010 = 194 = -62
$C3 = %11000011 = 195 = -61
$C4 = %11000100 = 196 = -60
$C5 = %11000101 = 197 = -59
$C6 = %11000110 = 198 = -58
$C7 = %11000111 = 199 = -57
$C8 = %11001000 = 200 = -56
$C9 = %11001001 = 201 = -55
$CA = %11001010 = 202 = -54
$CB = %11001011 = 203 = -53
$CC = %11001100 = 204 = -52
$CD = %11001101 = 205 = -51
$CE = %11001110 = 206 = -50
$CF = %11001111 = 207 = -49
$D0 = %11010000 = 208 = -48
$D1 = %11010001 = 209 = -47
$D2 = %11010010 = 210 = -46
$D3 = %11010011 = 211 = -45

$D4 = %11010100 = 212 = -44
$D5 = %11010101 = 213 = -43
$D6 = %11010110 = 214 = -42
$D7 = %11010111 = 215 = -41
$D8 = %11011000 = 216 = -40
$D9 = %11011001 = 217 = -39
$DA = %11011010 = 218 = -38
$DB = %11011011 = 219 = -37
$DC = %11011100 = 220 = -36
$DD = %11011101 = 221 = -35
$DE = %11011110 = 222 = -34
$DF = %11011111 = 223 = -33
$E0 = %11100000 = 224 = -32
$E1 = %11100001 = 225 = -31
$E2 = %11100010 = 226 = -30
$E3 = %11100011 = 227 = -29
$E4 = %11100100 = 228 = -28
$E5 = %11100101 = 229 = -27
$E6 = %11100110 = 230 = -26
$E7 = %11100111 = 231 = -25
$E8 = %11101000 = 232 = -24
$E9 = %11101001 = 233 = -23
$EA = %11101010 = 234 = -22
$EB = %11101011 = 235 = -21
$EC = %11101100 = 236 = -20
$ED = %11101101 = 237 = -19
$EE = %11101110 = 238 = -18
$EF = %11101111 = 239 = -17
$F0 = %11110000 = 240 = -16
$F1 = %11110001 = 241 = -15
$F2 = %11110010 = 242 = -14
$F3 = %11110011 = 243 = -13
$F4 = %11110100 = 244 = -12
$F5 = %11110101 = 245 = -11
$F6 = %11110110 = 246 = -10
$F7 = %11110111 = 247 = -9
$F8 = %11111000 = 248 = -8
$F9 = %11111001 = 249 = -7
$FA = %11111010 = 250 = -6
$FB = %11111011 = 251 = -5
$FC = %11111100 = 252 = -4
$FD = %11111101 = 253 = -3
$FE = %11111110 = 254 = -2
$FF = %11111111 = 255 = -1

HD63B09EP Technical Reference Guide
Revision 3

By Chet Simpson
Modifications and Corrections by Alan DeKok
Additional Notes by Roger Taylor

INDEX

Introduction

The HD63B09EP microprocessor by Hitachi, is a MC68B09E compatible chip containing additional registers and an additional instruction set. The 6309 was thought to be a flakey chip though, because it would sometimes crash or change the values of registers when it encountered an addressing mode or opcode invalid to the 6809. This was later found to be an extended instruction set and a feature that would trap some programming errors and jump to a specified location in memory.

Hitachi licensed the rights of the 6809 instruction set from Motorola to make a 6809 compatible chip. When they finished the design, they found there was a lot of unused space in the chip. With this in mind they added extra registers and expanded on the instruction set, but due to the licensing agreement with Motorola, they were unable to release the information about the extra features.

Not only does the chip have an expanded instruction set, but it also has a native mode that will run many of the instructions in fewer clock cycles and a mode select for the FIRQ (Fast Interrupt ReQuest) that will enable it to opperate the same as the IRQ.

In fact, all new instructions will execute in emulation mode, which was originally seen when 'illegal' 6809 instructions produced odd effects when run on a computer with a 6309 installed.

The additional instruction set was first written about in the April 1988 issue of "Oh!FM", a Japanese magazine, and was later translated by Hirotsugu Kakagawa. This opened a whole new door to those who wished to use the 6309 in place of the 6809.

In the beginning of 1992, Tandy Color Computer users in the US found out about these features. Although there has been limited and sometimes incorrect information about the new functions of the chip, I hope to bridge that gap with the information provided here.

Remember that this information is of technical nature and makes no attempt to teach assembly language programming. It is ONLY a technical reference guide for those who already know assembly and wish to use these features in their programs. Although all of the opcodes for the 6309/6809 chip are listed in the appendix, only the additional features supplied by the 6309 will be discussed.

Summary of Features

More registers:
- one 8/16 bit 'zero' register
- Two 8bit accumulators.
- One 16bit concatenated register
- One 16bit value register.
- One 8bit mode/error register.
- One 32bit concatenated register

Two modes: MC68B09E emulation mode and HD63B09EP native mode.

Reduced execution cycles when running in native mode.

Many additional instructions.

Error trapping of illegal instructions and zero divisions.

Description of Additional Registers

The 6309 has 7 additional registers. Only 4 of these are actual registers. 2 are combinations of registers, and the last is a constant-value register. These registers are:

```
ACCE  - 8 bit accumulator.
ACCF  - 8 bit accumulator.
W     - 16 bit concatenated register (ACCE and ACCF combined).
V     - 16 bit register (which can only be accessed with the
                         inter-register instructions).
0     - zero register
MD    - 8 bit mode/error register.
Q     - 32 bit concatenated register (ACCA, ACCB ,ACCE and ACCF
                                      combined).
```

ACCE and ACCF both work the same as the ACCA and ACCB accumulators. This makes for easier programming in math and data oriented routines.

The W register is like the D register in the 6809. It is a concatenated register containing the values of ACCE and ACCF as one 16 bit value. ACCE is contained in the high 8 bits and ACCF is contained in the low 8 bits.

The V register is a 16 bit register that can only be accessed with inter-register instructions such a TFR and EXG. The contents of this register will not change if the CPU is reset, allowing this register to be used as a constant value for the program.

The 0 register is always zero, independant of reads/writes to it. It enables a zero value to be used in inter-register operations without accessing memory, or changing the value of another register. If a 0 byte is stored at address $0000, it may also be used to clear large amounts of memory quickly via 'TFM 0,r+'

The MD register is a mode and error register and works much in the same way as the CC register. The bit definitions are as follows:

```
Write bits
  Bit 0   -  Execution mode of the 6309.
             If clear ( 0 ), the cpu is in 6809 emulation mode.
             If set ( 1 ), the cpu is in 6309 native mode.
  Bit 1   -  FIRQ mode
             If clear ( 0 ), the FIRQ will occur normally.
             If set ( 1 ) , the FIRQ will operate the same as the
                            IRQ

  Bits 2 to 5 are unused

Read bits - One of these bits is set when the 6309 traps an error
  Bit 6   -  This bit is set ( 1 ) if an illegal instruction is
             encountered
  Bit 7   -  This bit is set ( 1 ) if a zero division occurs.
```

The Q register is a 32 bit concatenated register. This register is the same as the D and W register except for one respect. It contains the values of ACCA, ACCB, ACCE and ACCF respectively. This register is used mostly with the additional math instructions supplied with the 6309 which will be discussed later.

Modes of Operation

The 6309 has two modes of operation; 6809 Emulation mode in which the chip acts and executes instructions the same as the 6809, and 6309 Native mode which stores an extra two bytes on the stack when an interrupt (IRQ) occurs, and executes instructions in fewer clock cycles.

When in native mode, the W register (2 additional bytes) is stored (PSHS) on the system stack when an interrupt occurs, it is stored on the stack right after the D (general data) register. Since ALL register values are stored on the system stack when an IRQ (NOT FIRQ - See FIRQ modes for more information) occurs, great care should be taken when writing or patching those routines to run in native mode.

Pull <- CC,A,B,E*,F*,DP,Xhi,Xlo,Yhi,Ylo,Uhi,Ulo,PChi,PClo <- Push

* indicates the additional registers stored on the system stack

When in native mode those interrupt routines which modify the return address by modifying the 10th and 11th byte offsets from the stack (STX 10,S or STY 10,S etc.) will have to be changed to modify the 12th and 13th byte offsets from the stack (STX 12,S or STY 12,S etc.). If those routines are not patched to run in native mode they will either get stuck in a continuous loop or will crash the system due to the fact that they are not returning to the correct address. This poses a MAJOR problem for OS-9 Level II since its main interrupt handling routine relies highly on the changing of the return (PC) address on the stack. Disk read/write and formatting routines also rely heavily on changing the return address during an NMI (Non-Maskable Interrupt).

To patch those routines which do modify the return address, the program or routine must be disassembled or modified with a disk sector editing program. Look for instructions such as STX 10,S or STY 10,S that has an RTI (Return from Interrupt) instruction within the next few lines of the routine. The line containing STX 10,S or STY 10,S should be changed to STX 12,S or STY 12,S respectively.

Remember, after those routines are patched, those programs using them will NOT work in emulation mode and will require native mode to be enabled upon startup.

Native Mode and Timing Loops

There is at least one more problem that needs to be addressed. Those are routines which are dependant on timing loops for accuarate operation. Since the 6309 executes instructions faster when in native mode, those routines that use timing loops would be effected. Since this can pose a problem and can create erratic operation, the delay value or routine will need to be changed for the routine to operate correctly.

Those routines are usually serial-printer routines, cassette read/write timimg routines, software clocks and some disk read/write routines.

Modes of the Fast Interrupt Request (FIRQ)

The designers of the 6309 decided that with the additional instructions and native mode of operation, the FIRQ may be used more than it usually is. With this in mind they decided to allow you to make the FIRQ run the same as the IRQ and store (PSHS) all the current values of the registers on the system stack. Normally, the FIRQ only stores the CC (condition code) and the PC (Program Counter/return address) on the stack, so to keep compatability with the 6809, they included it as a selectable feature in the MD (Mode/status) register.

Inter-Register Instructions

The new Inter-Register instructions (ADCR, ADDR, CMPR, EORR, ORR, SBCR, and SUBR) all work the same as their register/memory (ADCA, ADDA, etc.) counterparts except that they operate between registers. All of the new instructions use the same post-byte information as the normal TFR instruction and use the format of R0,R1 (register 0 and Register 1 respectively) with the result going into R1. See Block Transfers for information on the TFR block move instructions.

Mixed-size inter-register operations default to using identical sized register. So TFR A,X actually executes as TFR D,X. You could also do 'lea(d) d,pc' type things by doing 'addr pc,d'. As the new inter-register instructions can now perform math using the PC register, REALLY odd possibilities exist. Try looking at code like 'eorr d,pc', and figuring out where it ends up.

Inter-register instructions with 16-bit r1 and CC or DP (8-bit r2) are legal, but the results are unknown.

Bit Manipulation of Memory Locations

The AIM, EIM, OIM and TIM instructions all do logical bit manipulations to locations in memory, with the result stored into the location, and the respective bits for each instruction set in the CC register. They can be used in the DIRECT, INDEXED or EXTENDED adressing modes.

Instruction descriptions:

AIM - AND IN MEMORY
EIM - EOR IN MEMORY
OIM - OR IN MEMORY
TIM - TEST bits IN MEMORY

Instruction format: X, post byte, operand

Where X is the instruction op-code, post-byte contains the bits to AND, OR, EOR or TEST against the memory location, and the operand is the memory location or indexing post-byte depending on the mode of operation.

Mnemonic format:
Instruction logical operation value, memory location or index operation

Mnemonic example:
AIM #$0F,$E00

The example takes the contents of memory location $E00, does a LOGICAL and with the Value #$0F and then stores the result back into $E00.

Bit Transfers Between Memory Locations and Registers

The BAND, BIAND, BOR, BIOR, BEOR, BIEOR, LDBT, and STBT all do logical operations to bits for the n-th bit in a memory location and the m-th bit of a register. The LDBT and STBT instructions allow you to transfer certain bits between registers and memory locations. All instructions allow you to specify which register to use, which bit location to use in the register, which bit location to use in the memory location, and the memory location to use. This allows you to transfer/or do a logical operation with the 7th bit of a register and the 3rd bit of a memory location. All bits are accessible on either the register or memory locations. The only limitations are that the instructions can only be used with the A and B accumulators and the CC (condition Code) registers. It should also be noted that these instructions can only be used in the DIRECT addressing mode.

Instruction description:

BAND - AND a bit in a register with bit from memory location
BIAND - AND a bit in a register with the complement of the bit in memory
BOR - OR a bit in a register with a bit from a memory location
BIOR - OR a bit in a register with the complement of the bit in memory
BEOR - EOR a bit in a register with a bit from a memory location
BIEOR - EOR a bit in a register with the complement of the bit in memory
LDBT - Load a bit from a memory location into a bit in a register
STBT - Store a bit from a register into a memory location.

Instruction format:

x, post-byte, memory location

Where X is the instruction op-code, the post-byte contains the register, source and destination bit information and the memory location is the 8 bit value of the memory location to be used (Remember only DIRECT mode is allowed with these instructions).

Mnemonic format:

instruction, register, source bit, destination bit, memory location

Mnemonic example:

```
BOR A,1,7,$00
```

The example would take the first (1) bit of register A (A) and OR it into the 7th (7) bit of memory location $00 ($00) of the direct page (DP register value)

The post-byte of these instructions are not the same as the post-byte used in any other operation (indexed or inter-register) as all of the information (register, source and destination bit) is contained in one post-byte value.

Block Transfers

Block transfers are used to move a certain number of bytes from one place in memory to another with the use of one instruction. Two 16 bit registers (X, Y, U or S) are used to specify the source and destination addresses, and the size of the block to be transferred is specified with the W register. It should be noted that even though the IRQ and FIRQ only occur after the current instruction is finished, block moves can be interrupted. After the interrupt returns, the last byte read is read once more. i.e. It is read _twice_ by the CPU This can cause problems with memory mapped I/O devices, so caution is advised when using the block transfers. There isn't much control over these 4 instructions so the only thing applicable for them would be large block moves such as scrolling the screen or clearing an area in memory with a certain value.

TFM r0+,r1 and TFM r0,r1+ can be considered a poor mans DMA channel. Since all the data is either copied into or read from one memory location.

Four types of block transfers have been provided.

Mnemonic examples:

(R0 - source address register, R1 - destination address register.)

TFM r0+,r1+
- Transfer from R0 to R1 in incrementing order.

TFM r0-,r1-
- Transfer from R0 to R1 in decrementing order.

TFM r0+,r1
- Pour from R0 into R1, only incrementing R0 (R1 stays the same).

TFM r0,r1+
- Read from R0 into R1, only incrementing R0 (R1 stays the same).

Mnemonic example:

```
LDW #$100
LDX #$600
LDY #$700
TFM X+,Y+
```

The example would move 256 (LDW $100) bytes from #$600 (LDX #$&00) in memory to #$700 (LDY #$700) in memory, incrementing the value of each register (X and Y), and decrementing the value of the W register each time a byte if moved.

When moves like this are done, the pointer registers (X and Y in the example) will not be the same value they were before the transfer was initiated, but will but will be their original values PLUS the value of the W register (#$100 in the example). So in the example once the move is complete, the value of X will be returned as #$700 and the value of Y will be returned as #$800. The value of W register will be 0.

The 0 register may be used as a source or destination address register, and the data will be read from, or written to, address $0000.

It is illegal to use any of the CC, DP, W, V, or PC registers as either a source or destination register.

New math commands

The 6309 has 3 additional math instructions. A 16 bit by 16 bit signed multiply (MULD), a 16 bit by 8 bit signed divide (DIVD) and a 32 bit by 16 bit signed divide (DIVQ). These instructions can all be used in Immediate, direct, indexed and extended addressing modes.

The MULD (16 bit by 16 bit) instruction does a signed multiply of the contents of the D register and a value from memory (or in direct mode). The signed result is stored in the Q register.

The DIVD (16 bit by 8 bit) instruction does a signed divide of the contents of the D register with a value from memory (or in direct mode). The signed result is stored with the quotient in W and the modulo (remainder) in D.

The DIVQ (32 bit by 16 bit) instruction does a signed divide of the contents of the Q register with a value from memory (or in direct mode). The signed result is stored with the quotient in W and the modulo (remainder) in D.

Error Trapping

The 6309 has an internal error trapping handler that will jump to a specific location in memory when either an error is encountered in the DIVision instructions (only divide by zero) or an illegal instruction is encountered. When an error is encountered, the 6309 will jump to the memory location contained in $FFF0 (and $FFF1) which was originally reserved by the 6809.

The trap may cause problems with machines that have $FF00 hardcoded with the values $0000. A new EPROM should be burned to correct for the new behaviour of the 6309.

As many people know, an illegal instruction trap is extremely useful for debugging programs, as it prevents the entire machine from crashing when a bug is encountered.

Note that many pseudo-legal instructions on the 6809 are now illegal on the 6309, e.g. $1020xxxx executes as an LBRA on a 6809, but results in a trap on a 6309.

Additional Instructions

The 6309 has MANY new instructions. Most are variations of old instructions of the 6809 for use with the new registers. The new instruction set can be used in both native and emulation mode. Here is a list of the new instructions of the 6309:

ADCD
- Adds immediate or memory operand to the D register plus the current status of the carry with the result going to D.

ADCR
- Adds two registers together plus the current status of the carry.

ADDE , ADDF, ADDW
- Add of immediate or memory operand to E, F or W with results going to E, F or W

ADDR
- Adds two registers together

ANDD
- Logical AND of immediate or memory operand to D register with result going to D.

ANDR
- Logical AND of a register with the contents of another register

ASLD (Same as LSLD)
- Arithmetic shift left. Shifts D one bit left, clearing LSB.

ASRD
- Arithmetic shift right of the D register with sign extending.

BITD
- Test any bit or bits of the D register.

BITMD
- Test any bit or bits of the MD (mode) register.

CLRD, CLRE, CLRF, CLRW
- Clear register D, E, F or W to zero.

CMPE, CMPF, CMPW
- Compares the contents of E, F or W with the immediate or memory operand. Sets all CC except H on result.

CMPR
- Compares one register to another and sets all CC bits except H on result.

COMD, COME, COMF, COMW
- One's complement D ,E, F, or W. Changes all zero's to one's and all one's to zero's.

DECD, DECE, DECF, DECW
- Decrement D, E, F, or W by 1.

DIVD, DIVQ
- Does a 16 bit by 8 bit (DIVD) or a 32 bit by 16 bit (DIVQ) signed divide with immediate or memory operand with quotient in W and modulo (remainder) in D.

EORD
- Logical exclusive OR of D and immediate or memory operand.

EORR
- Logical exclusive OR of one register with the value of another register.

INCD, INCE, INCF, INCW
- Increment D, E, F or W by 1.

LDE, LDF, LDQ, LDW, LDMD
- Standard loading of E, F, Q, W or MD with immediate data value or operand from memory. (LDMD only valid with IMMEDIATE mode)

LSLD (Same as ASLD)
- Logical shift left. Shifts D one bit left, clearing LSB.

LSRD, LSRW
- Logical shift right. Shifts D or W one bit right, clearing MSB.

MULD
- Performs as 16bit by 16bit signed multiply with immediate or operand from memory. Result stored in Q.

NEGD
- Two's complement D register.

ORD
- Logical OR of register D and immediate or memory operand.

ORR
- Logical OR of one register with another.

PSHSW, PSHUW
- Stores contents of the W register on the (system or user) stack.

PULSW, PULUW
- Pull value from (system or user) stack into register W.

ROLD, ROLW
- Rotate D or W one bit left through the Carry Condition code.

RORD, RORW
- Rotate D or W one bit right through the Carry Condition code.

SBCD
- Subtract an immediate or memory operand plus any borrow in Carry from contents of D. Result stored in D.

SBCR
- Subtract the value of one register from another plus any borrow in the CC carry.

SEXW
- sign exdend the W register into the D register.

STE, STF, STQ, STW
- Store register E, F, Q or W to memory location (E,F), two memory locations(W), or four memory locations (Q).

SUBE, SUBF, SUBW
- Subtract immediate or memory operand from E, F or W. Result stored back in same register.

SUBR
- Subtract the value of one register from another.

TFM (Block transfer)
- Transfer W number of bytes from one location to another. Returns pointer registers offset of the starting value in the W register and returns the W register as 0. Indexed operation only

TSTD, TSTE, TSTF, TSTW
- Test contents of D, E, F or W by setting N and X condition codes based on data in register.

The Opcode and Mnemonics opcode reference tables are both complete listings that contain both the Opcode instruction and the HEX equivalant in all available addressing modes. The first table is arranged sequentially by the binary opcodes, while the second table is arranged alphabetically by the Mnemonic instructions.

At the end of the second table there are data tables containing information on Bit transfer/manipulation, branch instructions, inter-register instructions, and general register and stack information. These are all helpful to the serious assembly language programmer, who should always have one.

Opcode table

Opcode (* 6309)	Mnemonic	Mode	Cycles 6809	Cycles (6309)	Length
00	NEG	Direct	6	(5)	2
* 01	OIM	Direct	6		3
* 02	AIM	Direct	6		3
03	COM	Direct	6	(5)	2
04	LSR	Direct	6	(5)	2
* 05	EIM	Direct	6		3
06	ROR	Direct	6	(5)	2
07	ASR	Direct	6	(5)	2
08	ASL/LSL	Direct	6	(5)	2
09	ROL	Direct	6	(5)	2
0A	DEC	Direct	6	(5)	2
* 0B	TIM	Direct	6		
0C	INC	Direct	6	(5)	2
0D	TST	Direct	6	(4)	2
0E	JMP	Direct	3	(2)	2
0F	CLR	Direct	6	(5)	2
10	(PREBYTE)				
11	(PREBYTE)				
12	NOP	Inherent	2	(1)	1
13	SYNC	Inherent	2	(1)	1
* 14	SEXW	Inherent	4		1
16	LBRA	Relative	5	(4)	3
17	LBSR	Relative	9	(7)	3
19	DAA	Inherent	2	(1)	1
1A	ORCC	Immediate	3	(2)	2
1C	ANDCC	Immediate	3		2
1D	SEX	Inherent	2	(1)	1
1E	EXG	Immediate	8	(5)	2
1F	TFR	Immediate	6	(4)	2
20	BRA	Relative	3		2
21	BRN	Relative	3		2
22	BHI	Relative	3		2
23	BLS	Relative	3		2
24	BHS/BCC	Relative	3		2
25	BLO/BCS	Relative	3		2
26	BNE	Relative	3		2
27	BEQ	Relative	3		2

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
28	BVC	Relative	3	2
29	BVS	Relative	3	2
2A	BPL	Relative	3	2
2B	BMI	Relative	3	2
2C	BGE	Relative	3	2
2D	BLT	Relative	3	2
2E	BGT	Relative	3	2
2F	BLE	Relative	3	2
30	LEAX	Indexed	4+	2
31	LEAY	Indexed	4+	2
32	LEAS	Indexed	4+	2
33	LEAU	Indexed	4+	2
34	PSHS	Immediate	5+ (4+)	2
35	PULS	Immediate	5+ (4+)	2
36	PSHU	Immediate	5+ (4+)	2
37	PULU	Immediate	5+ (4+)	2
39	RTS	Inherent	5 (1)	1
3A	ABX	Inherent	3 (1)	1
3B	RTI	Inherent	6/15 (17)	1
3C	CWAI	Immediate	22 (20)	2
3D	MUL	Inherent	11 (10)	1
3F	SWI	Inherent	19 (21)	1
40	NEGA	Inherent	2 (1)	1
43	COMA	Inherent	2 (1)	1
44	LSRA	Inherent	2 (1)	1
46	RORA	Inherent	2 (1)	1
47	ASRA	Inherent	2 (1)	1
48	ASLA/LSLA	Inherent	2 (1)	1
49	ROLA	Inherent	2 (1)	1
4A	DECA	Inherent	2 (1)	1
4C	INCA	Inherent	2 (1)	1
4D	TSTA	Inherent	2 (1)	1
4F	CLRA	Inherent	2 (1)	1
50	NEGB	Inherent	2 (1)	1
53	COMB	Inherent	2 (1)	1
54	LSRB	Inherent	2 (1)	1
56	RORB	Inherent	2 (1)	1
57	ASRB	Inherent	2 (1)	1
58	ASLB/LSLB	Inherent	2 (1)	1
59	ROLB	Inherent	2 (1)	1
5A	DECB	Inherent	2 (1)	1
5C	INCB	Inherent	2 (1)	1
5D	TSTB	Inherent	2 (1)	1
5F	CLRB	Inherent	2 (1)	1
60	NEG	Indexed	6+	2+
* 61	OIM	Indexed	6+	3+
* 62	AIM	Indexed	7	3+
63	COM	Indexed	6+	2+
64	LSR	Indexed	6+	2+
* 65	EIM	Indexed	7+	3+
66	ROR	Indexed	6+	2+
67	ASR	Indexed	6+	2+
68	ASL/LSL	Indexed	6+	2+
69	ROL	Indexed	6+	2+

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
6A	DEC	Indexed	6+	2+
* 6B	TIM	Indexed	7+	3+
6C	INC	Indexed	6+	2+
6D	TST	Indexed	6+ (5+)	2+
6E	JMP	Indexed	3+	2+
6F	CLR	Indexed	6+	2+
70	NEG	Extended	7 (6)	3
* 71	OIM	Extended	7	4
* 72	AIM	Extended	7	4
73	COM	Extended	7 (6)	3
74	LSR	Extended	7 (6)	3
76	ROR	Extended	7 (6)	3
* 75	EIM	Extended	7	4
77	ASR	Extended	7 (6)	3
78	ASL/LSL	Extended	7 (6)	3
79	ROL	Extended	7 (6)	3
7A	DEC	Extended	7 (6)	3
* 7B	TIM	Extended	7	4
7C	INC	Extended	7 (6)	3
7D	TST	Extended	7 (5)	3
7E	JMP	Extended	4 (3)	3
7F	CLR	Extended	7 (6)	3
80	SUBA	Immediate	2	2
81	CMPA	Immediate	2	2
82	SBCA	Immediate	2	2
83	SUBD	Immediate	4 (3)	3
84	ANDA	Immediate	2	2
85	BITA	Immediate	2	2
86	LDA	Immediate	2	2
88	EORA	Immediate	2	2
89	ADCA	Immediate	2	2
8A	ORA	Immediate	2	2
8B	ADDA	Immediate	2	2
8C	CMPX	Immediate	4 (3)	3
8D	BSR	Relative	7 (6)	2
8E	LDX	Immediate	3	3
90	SUBA	Direct	4 (3)	2
91	CMPA	Direct	4 (3)	2
92	SBCA	Direct	4 (3)	2
93	SUBD	Direct	6 (4)	2
94	ANDA	Direct	4 (3)	2
95	BITA	Direct	4 (3)	2
96	LDA	Direct	4 (3)	2
97	STA	Direct	4 (3)	2
98	EORA	Direct	4 (3)	2
99	ADCA	Direct	4 (3)	2
9A	ORA	Direct	4 (3)	2
9B	ADDA	Direct	4 (3)	2
9C	CMPX	Direct	6 (4)	2
9D	JSR	Direct	7 (6)	2
9E	LDX	Direct	5 (4)	2
9F	STX	Direct	5 (4)	2
A0	SUBA	Indexed	4+	2+
A1	CMPA	Indexed	4+	2+

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
A2	SBCA	Indexed	4+	2+
A3	SUBD	Indexed	6+ (5+)	2+
A4	ANDA	Indexed	4+	2+
A5	BITA	Indexed	4+	2+
A6	LDA	Indexed	4+	2+
A7	STA	Indexed	4+	2+
A8	EORA	Indexed	4+	2+
A9	ADCA	Indexed	4+	2+
AA	ORA	Indexed	4+	2+
AB	ADDA	Indexed	4+	2+
AC	CMPX	Indexed	6+ (5+)	2+
AD	JSR	Indexed	7+ (6+)	2+
AE	LDX	Indexed	5+	2+
AF	STX	Indexed	5+	2+
B0	SUBA	Extended	5 (4)	3
B1	CMPA	Extended	5 (4)	3
B2	SBCA	Extended	5 (4)	3
B3	SUBD	Extended	7 (5)	3
B4	ANDA	Extended	5 (4)	3
B5	BITA	Extended	5 (4)	3
B6	LDA	Extended	5 (4)	3
B7	STA	Extended	5 (4)	3
B8	EORA	Extended	5 (4)	3
B9	ADCA	Extended	5 (4)	3
BA	ORA	Extended	5 (4)	3
BB	ADDA	Extended	5 (4)	3
BC	CMPX	Extended	7 (5)	3
BD	JSR	Extended	8 (7)	3
BE	LDX	Extended	6 (5)	3
BF	STX	Extended	6 (5)	3
C0	SUBB	Immediate	2	2
C1	CMPB	Immediate	2	2
C2	SBCB	Immediate	2	2
C3	ADDD	Immediate	4 (3)	3
C4	ANDB	Immediate	2	2
C5	BITB	Immediate	2	2
C6	LDB	Immediate	2	2
C8	EORB	Immediate	2	2
C9	ADCB	Immediate	2	2
CA	ORB	Immediate	2	2
CB	ADDB	Immediate	2	2
CC	LDD	Immediate	3	3
* CD	LDQ	Immediate	5	5
CE	LDU	Immediate	3	3
D0	SUBB	Direct	4 (3)	2
D1	CMPB	Direct	4 (3)	2
D2	SBCB	Direct	4 (3)	2
D3	ADDD	Direct	6 (4)	2
D4	ANDB	Direct	4 (3)	2
D5	BITB	Direct	4 (3)	2
D6	LDB	Direct	4 (3)	2
D7	STB	Direct	4 (3)	2
D8	EORB	Direct	4 (3)	2
D9	ADCB	Direct	4 (3)	2

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
DA	ORB	Direct	4 (3)	2
DB	ADDB	Direct	4 (3)	2
DC	LDD	Direct	5 (4)	2
DD	STD	Direct	5 (4)	2
DE	LDU	Direct	5 (4)	2
DF	STU	Direct	5 (4)	2
E0	SUBB	Indexed	4+	2+
E1	CMPB	Indexed	4+	2+
E2	SBCB	Indexed	4+	2+
E3	ADDD	Indexed	6+ (5+)	2+
E4	ANDB	Indexed	4+	2+
E5	BITB	Indexed	4+	2+
E6	LDB	Indexed	4+	2+
E7	STB	Indexed	4+	2+
E8	EORB	Indexed	4+	2+
E9	ADCB	Indexed	4+	2+
EA	ORB	Indexed	4+	2+
EB	ADDB	Indexed	4+	2+
EC	LDD	Indexed	5+	2+
ED	STD	Indexed	5+	2+
EE	LDU	Indexed	5+	2+
EF	STU	Indexed	5+	2+
F0	SUBB	Extended	5 (4)	3
F1	CMPB	Extended	5 (4)	3
F2	SBCB	Extended	5 (4)	3
F3	ADDD	Extended	7 (5)	3
F4	ANDB	Extended	5 (4)	3
F5	BITB	Extended	5 (4)	3
F6	LDB	Extended	5 (4)	3
F7	STB	Extended	5 (4)	3
F8	EORB	Extended	5 (4)	3
F9	ADCB	Extended	5 (4)	3
FA	ORB	Extended	5 (4)	3
FB	ADDB	Extended	5 (4)	3
FC	LDD	Extended	6 (5)	3
FD	STD	Extended	6 (5)	3
FE	LDU	Extended	6 (5)	3
FF	STU	Extended	6 (5)	3
1021	LBRN	Reletive	5/6 ( )	4
1022	LBHI	Reletive	5/6 ( )	4
1023	LBLS	Reletive	5/6 ( )	4
1024	LBHS/LBCC	Reletive	5/6 ( )	4
1025	LBCS/LBLO	Reletive	5/6 ( )	4
1026	LBNE	Reletive	5/6 ( )	4
1027	LBEQ	Reletive	5/6 ( )	4
1028	LBVC	Reletive	5/6 ( )	4
1029	LBVS	Reletive	5/6 ( )	4
102A	LBPL	Reletive	5/6 ( )	4
102B	LBMI	Reletive	5/6 ( )	4
102C	LBGE	Reletive	5/6 ( )	4
102D	LBLT	Reletive	5/6 ( )	4
102E	LBGT	Reletive	5/6 ( )	4
102F	LBLE	Reletive	5/6 ( )	4
* 1030	ADDR	Register	4	3

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
* 1031	ADCR	Register	4	3
* 1032	SUBR	Register	4	3
* 1033	SBCR	Register	4	3
* 1034	ANDR	Register	4	3
* 1035	ORR	Register	4	3
* 1036	EORR	Register	4	3
* 1037	CMPR	Register	4	3
* 1038	PSHSW	Register	6	2
* 1039	PULSW	Register	6	2
* 103A	PSHUW	Register	6	2
* 103B	PULUW	Register	6	2
103F	SWI2	Inherent	20 (22)	2
* 1040	NEGD	Inherent	3 (2)	2
* 1043	COMD	Inherent	3 (2)	2
* 1044	LSRD	Inherent	3 (2)	2
* 1046	RORD	Inherent	3 (2)	2
* 1047	ASRD	Inherent	3 (2)	2
* 1048	ASLD/LSLD	Inherent	3 (2)	2
* 1049	ROLD	Inherent	3 (2)	2
* 104A	DECD	Inherent	3 (2)	2
* 104C	INCD	Inherent	3 (2)	2
* 104D	TSTD	Inherent	3 (2)	2
* 104F	CLRD	Inherent	3 (2)	2
* 1053	COMW	Inherent	3 (2)	2
* 1054	LSRW	Inherent	3 (2)	2
* 1056	RORW	Inherent	3 (2)	2
* 1059	ROLW	Inherent	3 (2)	2
* 105A	DECW	Inherent	3 (2)	2
* 105C	INCW	Inherent	3 (2)	2
* 105D	TSTW	Inherent	3 (2)	2
* 105F	CLRW	Inherent	3 (2)	2
* 1080	SUBW	Immediate	5 (4)	4
* 1081	CMPW	Immediate	5 (4)	4
* 1082	SBCD	Immediate	5 (4)	4
1083	CMPD	Immediate	5 (4)	4
* 1084	ANDD	Immediate	5 (4)	4
* 1085	BITD	Immediate	5 (4)	4
* 1086	LDW	Immediate	5 (4)	4
* 1088	EORD	Immediate	5 (4)	4
* 1089	ADCD	Immediate	5 (4)	4
* 108A	ORD	Immediate	5 (4)	4
* 108B	ADDW	Immediate	5 (4)	4
108C	CMPY	Immediate	5 (4)	4
108E	LDY	Immediate	5 (4)	4
* 1090	SUBW	Direct	7 (5)	3
* 1091	CMPW	Direct	7 (5)	3
* 1092	SBCD	Direct	7 (5)	3
1093	CMPD	Direct	7 (5)	3
* 1094	ANDD	Direct	7 (5)	3
* 1095	BITD	Direct	7 (5)	3
* 1096	LDW	Direct	6 (5)	3
* 1097	STW	Direct	7 (5)	3
* 1098	EORD	Direct	7 (5)	3
* 1099	ADCD	Direct	7 (5)	3

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
* 109A	ORD	Direct	7 (5)	3
* 109B	ADDW	Direct	7 (5)	3
109C	CMPY	Direct	7 (5)	3
109E	LDY	Direct	6 (5)	3
109F	STY	Direct	6 (5)	3
* 10A0	SUBW	Indexed	7+ (6+)	3+
* 10A1	CMPW	Indexed	7+ (6+)	3+
* 10A2	SBCD	Indexed	7+ (6+)	3+
10A3	CMPD	Indexed	7+ (6+)	3+
* 10A4	ANDD	Indexed	7+ (6+)	3+
* 10A5	BITD	Indexed	7+ (6+)	3+
* 10A6	LDW	Indexed	6+	3+
* 10A7	STW	Indexed	6+	3+
* 10A8	EORD	Indexed	7+ (6+)	3+
* 10A9	ADCD	Indexed	7+ (6+)	3+
* 10AA	ORD	Indexed	7+ (6+)	3+
* 10AB	ADDW	Indexed	7+ (6+)	3+
10AC	CMPY	Indexed	7+ (6+)	3+
10AE	LDY	Indexed	6	3+
10AF	STY	Indexed	6	3+
* 10B0	SUBW	Extended	8 (6)	4
* 10B1	CMPW	Extended	8 (6)	4
* 10B2	SBCD	Extended	8 (6)	4
10B3	CMPD	Extended	8 (6)	4
* 10B4	ANDD	Extended	8 (6)	4
* 10B5	BITD	Extended	8 (6)	4
* 10B6	LDW	Extended	7 (6)	4
* 10B7	STW	Extended	7 (6)	4
* 10B8	EORD	Extended	8 (6)	4
* 10B9	ADCD	Extended	8 (6)	4
* 10BA	ORD	Extended	8 (6)	4
* 10BB	ADDW	Extended	8 (6)	4
10BC	CMPY	Extended	8 (6)	4
10BE	LDY	Extended	7 (6)	4
10BF	STY	Extended	7 (6)	4
10CE	LDS	Immediate	4	4
* 10DC	LDQ	Direct	8 (7)	3
* 10DD	STQ	Direct	8 (7)	3
10DE	LDS	Direct	6 (5)	3
10DF	STS	Direct	6 (5)	3
* 10DC	LDQ	Indexed	8+	3+
* 10DD	STQ	Indexed	8+	3+
10EE	LDS	Indexed	6+	3+
10EF	STS	Indexed	6+	3+
* 10DC	LDQ	Extended	9 (8)	4
* 10DD	STQ	Extended	9 (8)	4
10FE	LDS	Extended	7 (6)	4
10FF	STS	Extended	7 (6)	4
* 1130	BAND	Memory	7 (6)	4
* 1131	BIAND	Memory	7 (6)	4
* 1132	BOR	Memory	7 (6)	4
* 1133	BIOR	Memory	7 (6)	4
* 1134	BEOR	Memory	7 (6)	4
* 1135	BIEOR	Memory	7 (6)	4

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
* 1136	LDBT	Memory	7 (6)	4
* 1137	STBT	Memory	8 (7)	4
* 1138	TFM R+,R+	Register	6+3n	3
* 1139	TFM R-,R-	Register	6+3n	3
* 113A	TFM R+,R	Register	6+3n	3
* 113B	TFM R,R+	Register	6+3n	3
* 113C	BITMD	Immediate	4	3
* 113D	LDMD	Immediate	5	5
113F	SWI2	Inherent	20 ( )	2
* 1143	COME	Inherent	3 (2)	2
* 114A	DECE	Inherent	3 (2)	2
* 114C	INCE	Inherent	3 (2)	2
* 114D	TSTE	Inherent	3 (2)	2
* 114F	CLRE	Inherent	3 (2)	2
* 1153	COMF	Inherent	3 (2)	2
* 115A	DECF	Inherent	3 (2)	2
* 115C	INCF	Inherent	3 (2)	2
* 115D	TSTF	Inherent	3 (2)	2
* 115F	CLRF	Inherent	3 (2)	2
11AC	CMPS	Indexed	7 ( )	3
* 1180	SUBE	Immediate	3	3
* 1181	CMPE	Immediate	3	3
1183	CMPU	Immediate	5 (4)	4
* 1186	LDE	Immediate	3	3
* 118B	ADDE	Immediate	3	3
118C	CMPS	Immediate	5 (4)	4
* 118D	DIVD	Immediate	25	4
* 118E	DIVQ	Immediate	36	4
* 118F	MULD	Immediate	28	4
* 1190	SUBE	Direct	5 (4)	3
* 1191	CMPE	Direct	5 (4)	3
1193	CMPU	Direct	7 (5)	3
* 1196	LDE	Direct	5 (4)	3
* 1197	STE	Direct	5 (4)	3
* 119B	ADDE	Direct	5 (4)	3
119C	CMPS	Direct	7 (5)	3
* 119D	DIVD	Direct	27 (26)	3
* 119E	DIVQ	Direct	36 (35)	3
* 119F	MULD	Direct	30 (29)	3
* 11A0	SUBE	Indexed	5+	3+
* 11A1	CMPE	Indexed	5+	3+
11A3	CMPU	Indexed	7+ (6+)	3+
* 11A6	LDE	Indexed	5+	3+
* 11A7	STE	Indexed	5+	3+
* 11AB	ADDE	Indexed	5+	3+
11AC	CMPS	Indexed	7+ (6+)	3+
* 11AD	DIVD	Indexed	27+	3+
* 11AE	DIVQ	Indexed	36+	3+
* 11AF	MULD	Indexed	30+	3+
* 11B0	SUBE	Extended	6 (5)	4

Opcode (* 6309)	Mnemonic	Mode	Cycles	Length
* 11B1	CMPE	Extended	6 (5)	4
11B3	CMPU	Extended	8 (6)	4
* 11B6	LDE	Extended	6 (5)	4
* 11B7	STE	Extended	6 (5)	4
* 11BB	ADDE	Extended	6 (5)	4
11BC	CMPS	Extended	8 (6)	4
* 11BD	DIVD	Extended	28 (27)	4
* 11BE	DIVQ	Extended	37 (36)	4
* 11BF	MULD	Extended	31 (30)	4
* 11C0	SUBF	Immediate	3	3
* 11C1	CMPF	Immediate	3	3
* 11C6	LDF	Immediate	3	3
* 11CB	ADDF	Immediate	3	3
* 11D0	SUBF	Direct	5 (4)	3
* 11D1	CMPF	Direct	5 (4)	3
* 11D6	LDF	Direct	5 (4)	3
* 11D7	STF	Direct	5 (4)	3
* 11DB	ADDF	Direct	5 (4)	3
* 11E0	SUBF	Indexed	5+	3+
* 11E1	CMPF	Indexed	5+	3+
* 11E6	LDF	Indexed	5+	3+
* 11E7	STF	Indexed	5+	3+
* 11EB	ADDF	Indexed	5+	3+
* 11F0	SUBF	Extended	6 (5)	4
* 11F1	CMPF	Extended	6 (5)	4
* 11F6	LDF	Extended	6 (5)	4
* 11F7	STF	Extended	6 (5)	4
* 11FB	ADDF	Extended	6 (5)	4

Mnemonics Table

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	+	OP	~/~	#	OP	~/~	#	OP	~/~	#
ABX													3A	3/1	1
ADCA	89	2	2	99	4/3	2	A9	4+	2+	B9	5/4	3			
ADCB	C9	2	2	D9	4/3	2	E9	4+	2+	F9	5/3	3			
*ADCD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	89			99			A9			B9					
ADDA	8B	2	2	9B	4/3	2	AB	4+	2+	BB	5/4	3			
ADDB	CB	2	2	DB	4/3	2	EB	4+	2+	FB	5/4	3			
ADDD	C3	4/3	3	D3	6/4	2	E3	6+/5+	2+	F3	7/5	3			
*ADDE	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	8B			9B			AB			BB					
*ADDF	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	CB			DB			EB			FB					
*ADDW	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	8B			9B			AB			BB					
*AIM				02	6	3	62	7+	3+	72	7	4			
ANDA	84	2	2	94	4/3	2	A4	4+	2	B4	5/4	3			
ANDB	C4	2	2	D4	4/3	2	E4	4+	2	F4	5/4	3			
ANDCC	1C	3	2												
*ANDD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	84			94			A4			B4					
ASLA													48	2/1	1
ASLB													58	2/1	1
*ASLD													10	3/2	2
													48		
ASL				08	6/5	2	68	6+	2+	78	7/6	3			
ASRA													47	2/1	1
ASRB													57	2/1	1
*ASRD													10	3/2	1
													47		
ASR				07	6/6	2	67	6+	2+	77	7/6	3			
BITA	85	2	2	95	4/3	2	A5	4+	2+	B5	5/4	3			
BITB	C5	2	2	D5	4/3	2	E5	4+	2+	F5	5/4	3			
BITD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	85			95			A5			B5					
BITMD	11	4	3												
	3C														
CLRA													4F	2/1	1
CLRB													5F	2/1	1
*CLRD													10	3/2	2
													4F		
*CLRE													11	3/2	2
													4F		
*CLRF													11	3/2	2
													5F		
*CLRW													10	3/2	2
													5F		
CLR				0F	6/5	2	6F	6+	2+	7F	7/6	3			

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	+	OP	~/~	#	OP	~/~	#	OP	~/~	#
CMPA	81	2	2	91	4/3	2	A1	4+	2+	B1	5/4	3			
CMPB	C1	2	2	D1	4/3	2	E1	4+	2+	F1	5/4	3			
CMPD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	83			93			A3			B3					
*CMPE	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	81			91			A1			B1					
*CMPF	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	C1			D1			E1			F1					
CMPS	11	5/4	4	11	7/5	3	11	7+/6+	3+	11	8/6	4			
	8C			9C			AC			BC					
CMPU	11	5/4	4	11	7/5	3	11	7+/6+	3+	11	8/6	4			
	83			93			A3			B3					
*CMPW	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	81			91			A1			B1					
CMPX	8C	4/3	3	9C	6/4	2	AC	6+/5+	2+	BC	7/5	3			
CMPY	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	8C			9C			AC			BC					
COMA													43	2/1	1
COMB													53	2/1	1
*COMD													10	3/2	2
													43		
*COME													11	3/2	2
													43		
*COMF													11	3/2	2
													53		
*COMW													10	3/2	2
													53		
COM				03	6/5	2	63	6+	2+	73	7/6	3			
CWAI	3C	22/20	2												
DAA													19	2/1	1
DECA													4A	2/1	1
DECB													5A	2/1	1
*DECD													10	3/2	2
													4A		
*DECE													11	3/2	2
													4A		
*DECF													11	3/2	2
													5A		
*DECW													10	3/2	2
													5A		
DEC				0A	6/5	2	6A	6+	2+	7A	7/6	3			
*DIVD	11	25	3	11	27/26	3	11	27+	3+	11	28/27	4			
	8D			9D			AD			BD					
*DIVQ	11	34	4	11	36/35	3	11	36+	3+	11	37/36	4			
	8E			9E			AE			BE					
*EIM				05	6	3	65	7+	3+	75	7	4			
EORA	88	2	2	98	4/3	2	A8	4+	2+	B8	5/4	3			
EORB	C8	2	#	D8	4/3	2	E8	4+	2+	F8	5/4	3			
*EORD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	88			98			A8			B8					

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	+	OP	~/~	#	OP	~/~	#	OP	~/~	#
EXG	1E	8/5	2												
INCA													4C	2/1	1
INCB													5C	2/1	1
*INCD													10	3/2	2
													4C		
*INCE													11	3/2	2
													4C		
*INCF													11	3/2	2
													5C		
*INCW													10	3/2	2
													5C		
INC				0C	6/5	2	6C	6+	2+	7C	7/6	3			
JMP				0E	3/2	2	6E	3+	2+	7E	4/3	3			
JSR				9D	7/6	2	AD	7+/6+	2+	BD	8/7	3			
LDA	86	2	2	96	4/3	2	A6	4+	2+	B6	5/4	3			
LDB	C6	2	2	D6	4/3	2	E6	4+	2+	F6	5/4	3			
LDD	CC	3	3	DC	5/4	2	EC	5+	2+	FC	6/5	3			
*LDE	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	86			96			A6			B6					
*LDF	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	C6			D6			E6			F6					
*LDQ	CD	5	5	10	8/7	3	10	8+	3+	10	9/8	4			
				DC			EC			FC					
LDS	10	4	4	10	6/5	3	10	6+	3+	10	7/6	4			
	CE			DE			EE			FE					
LDU	CE	3	3	DE	5/4	2	EE	5+	2+	FE	6/5	3			
*LDW	10	4	4	10	6/5	3	10	6+	3+	10	7/6	4			
	86			96			A6			B6					
LDX	8E	3	3	9E	5/4	2	AE	5+	2+	BE	6/5	3			
LDY	10	4	4	10	6/5	3	10	6+	3+	10	7/6	4			
	8E			9E			AE			BE					
*LDMD	11	5	3												
	3D														
LEAS							32	4+	2+						
LEAU							33	4+	2+						
LEAX							30	4+	2+						
LEAY							31	4+	2+						
LSLA/LSLB/LSLD/LSL - Same as ASL															
LSRA													44	2/1	1
LSRB													54	2/1	1
*LSRD													10	3/2	2
													44		
*LSRW													10	3/2	2
													54		
LSR				04	6/5	2	64	6+	2+	74	7/6	3			
MUL													3D	11/10	1
*MULD	11	28	4	11	30/29	3	11	30+	3+	11	31/30	4			
	8F			9F			AF			BF					

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	+	OP	~/~	#	OP	~/~	#	OP	~/~	#
NEGA													40	2/1	1
NEGB													50	2/1	1
*NEGD													10	3/2	2
													40		
NEG				00	6/5	2	60	6+	2+	70	7/6	3			
NOP													12	2/1	1
*OIM				01	6	3	61	7+	3+	71	7	4			
ORA	8A	2	2	9A	4/3	2	AA	4+	2	BA	5/4	3			
ORB	CA	2	2	DA	4/3	2	EA	4+	2	FA	5/4	3			
ORCC	1A	3/2	2												
*ORD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	8A			9A			AA			BA					
PSHS	34	5+/4+	2												
PSHU	36	5+/4+	2												
*PSHSW	10	6	2												
	38	6	2												
*PSHUW	10	6	2												
	3A	6	2												
PULS	35	5+/4+	2												
PULU	37	5+/4+	2												
*PULSW	10	6	2												
	39														
*PULUW	10	6	2												
	3B														
ROLA													49	2/1	1
ROLB													59	2/1	1
*ROLD													10	3/2	2
													49		
*ROLW													10	3/2	2
													59		
ROL				09	6/5	2	69	6+	2+	79	7/6	3			
RORA													46	2/1	1
RORB													56	2/1	1
*RORD													10	3/2	2
													46		
*RORW													10	3/2	2
													56		
ROR				06	6/5	2	66	6+	2+	76	7/6	3			
RTI													3B	6/17	1
														15/17	
RTS													39	5/4	1
SBCA	82	2	2	92	4/3	2	A2	4+	2+	B2	5/4	3			
SBCB	C2	2	2	D2	4/3	2	E2	4+	2+	F2	5/2	3			
*SBCD	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	82			92			A2			B2					
SEX													1D	2/1	1
*SEXW													14	4	1

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	+	OP	~/~	#	OP	~/~	#	OP	~/~	#
STA				97	4/3	2	A7	4+	2+	B7	5/4	3			
STB				D7	4/3	2	E7	4+	2+	F7	5/4	3			
STD				DD	5/4	2	ED	5+	2+	FD	6/5	3			
*STE				11	5/4	3	11	5+	3+	11	6/5	4			
				97			A7			B7					
*STF				11	5/4	3	11	5+	3+	11	6/5	4			
				D7			E7			F7					
*STQ				10	8/7	3	10	8+	3+	10	9/8	4			
				DD			ED			FD					
*STS				10	6/5	3	10	6+	3+	10	7/6	4			
				DF			EF			FF					
STU				DF	5/4	2	EF	5+	2+	FF	6/5	3			
*STW				10	6/5	3	10	6+	3+	10	7/6	4			
				97			A7			B7					
STX				9F	5/4	2	AF	5+	2+	BF	6/5	3			
STY				10	6/5	3	10	6+	3+	10	7/6	4			
				9F			AF			BF					
SUBA	80	2	2	90	4/3	2	A0	4+	2+	B0	5/4	3			
SUBB	C0	2	2	D0	4/3	2	E0	4+	2+	F0	5/4	3			
SUBD	83	4/3	3	93	6/4	3	A3	6+/5+	2+	B3	7/5	3			
*SUBE	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	80			90			A0			B0					
*SUBF	11	3	3	11	5/4	3	11	5+	3+	11	6/5	4			
	C0			D0			E0			F0					
*SUBW	10	5/4	4	10	7/5	3	10	7+/6+	3+	10	8/6	4			
	80			90			A0			B0					
SWI													3F	19/21	1
SWI2													10	20/22	2
													3F		
SWI3													11	20/22	2
													3F		
SYNC													13	2+/1+	1
TFR 1	1F	6/4	2												
*TIM				0B	6	3	6B	7+	3+	7B	5	4			
TSTA													4D	2/1	1
TSTB													5D	2/1	1
*TSTD													10	3/2	2
													4D		
*TSTE													11	3/2	2
													4D		
*TSTF													11	3/2	2
													5D		
*TSTW													10	3/2	2
													5D		
TST				0D	6/4	2	6D	6+/5+	2+	7D	7/5	3			

Branch Instructions

Mnem	Immed.		
	OP	~/~	#
BCC	24	3	2
LBCC	10 24	5/6	4
BCS	25	3	2
LBCS	10 25	5/6	4
BEQ	27	3	2
LBEQ	10 27	5/6	4
BGE	2C	3	2
LBGE	10 2C	5/6	4
BGT	2E	3	2
LBGT	10 2E	5/6	4
BHI	22	3	2
LBHI	10 22	5/6	4
BHS	2F	3	2
LBHS	10 2F	5/6	4

Mnem	Immed.		
	OP	~/~	#
BLE	2F	3	2
LBLE	10 2F	5/6	4
BLO	25	3	2
LBLO	10 25	5/6	4
BLS	23	3	2
LBLS	10 23	5/6	4
BLT	2D	3	2
LBLT	10 2D	5/6	4
BMI	28	3	2
LBMI	10 28	5/6	4
BNE	26	3	2
LBNE	10 26	5/6	4

Mnem	Immed.		
	OP	~/~	#
BPL	2A	3	2
LBPL	10 2A	5/6	4
BRA	20	3	2
LBRA	16	5/4	3
BRN	21	3	2
LBRN	10 21	5/6	4
BSR	8D	7/6	2
LBSR	17	9/7	3
BVC	28	3	2
LBVC	10 28	5/6	4
BVS	29	3	2
LBVS	10 29	5/6	4

Bit Transfer/Manipulation

Mnem	Direct		
	OP	~/~	#
*BAND	11 30	7/6	4
*BIAND	11 31	7/6	4
*BOR	11 32	7/6	4
*BIOR	11 33	7/6	4
*BEOR	11 34	7/6	4
*BIEOR	11 35	7/6	4
*LDBT	11 36	7/6	4
*STBT	11 37	8/7	4

Post-Byte

7	6	5	4	3	2	1	0

Bits 7 and 6: Register

00 - CC 10 - B
01 - A 11 - Unused

Bits 5, 4 and 3: Source Bit

Bits 2, 1 and 0: Destination bit

Source/Destination Bit in binary form:

0 - 000 2 - 010 5 - 100 6 - 110
1 - 001 3 - 011 5 - 101 7 - 111

Both the source and destination bit portions of the post-byte are looked at by the 6309 as the actual bit NUMBER to transfer/store. Use the binary equivilant of the numbers (0 thru 7) and position them into the bit area of the post byte.

Logical Memory Operations

Mnem	Immed.			Direct			Indexed			Extended			Inherent		
	OP	~/~	#	OP	~/~	#	OP	~/~	#	OP	~/~	#	OP	~/~	#
*AIM				02	6	3	62	7+	3+	72	7	4			
*EIM				05	6	3	65	7+	3+	75	7	4			
*OIM				01	6	3	61	7+	3+	71	7	4			
*TIM				0B	6	3	6B	7+	3+	7B	5	4			

Inter-Register Instructions

Mnem	Forms	Register		
		OP	~/~	+
*ADCR	R0,R1	10	4	3
		31		
*ADDR	R0,R1	10	4	3
		30		
*ANDR	R0,R1	10	4	3
		34		
*CMPR	R0,R1	10	4	3
		37		
*EORR	R0,R1	10	4	3
		36		
EXG	R0,R1	1E	8/5	2
*ORR	R0,R1	10	4	3
		35		
*SBCR	R0,R1	10	4	3
		33		
*SUBR	R0,R1	10	4	3
		32		
TFR	R0,R1	1F	6/4	2
*TFM	R0+,R1+	11	6+3n	3
		38		
*TFM	R0-,R1-	11	6+3n	3
		39		
*TFM	R0+,R1	11	6+3n	3
		3A		
*TFM	R0,R1+	11	6+3n	3
		3B		

Transfer/Exchange and Inter-Register Post Byte

SOURCE				DESTINATION			
HI NIBBLE				LOW NIBBLE			

Register Field
(source or destination)

0000 - D (A:B)	1000 - A
0001 - X	1001 - B
0010 - Y	1010 - CCR
0011 - U	1011 - DPR
0100 - S	1100 - 0
0101 - PC	1101 - 0
0110 - W	1110 - E
0111 - V	1111 - F

The results of all Inter-Register operations are passsed into R1 with the exception of EXG which exchanges the values of registers and the TFR block transfers.

The register field codes %1100 and %1101 are both zero registers. They can be used as source or destination.

Indexed Address Modes and Post byte Information

Type	Forms	Assembler form	PostByte OP code	+/+ ~/~	+ #
Non-Indirect Modes					
Constant offset from R	No offset	,R	1rr00100	0	0
	5 bit offset	n,R	0rrnnnnn	1	0
	8 bit offset	n,R	1rr01000	1	1
	16 bit offset	n,R	1rr01001	4/3	2
Accumulator offset from R (Twos complement *offset)	A - Register	A,R	1rr00110	1	0
	B - Register	B,R	1rr00101	1	0
	E - Register	E,R	1rr00111	1	0
*	F - Register	F,R	1rr01010	1	0
	D - Register	D,R	1rr01011	4/2	0
*	W - Register	W,R	1rr01110	4/1	0
Auto increment and decrement of R	Increment 1	,R+	1rr00000	2/1	0
	Increment 2	,R++	1rr00001	3/2	0
	Decrement 1	,-R	1rr00010	2/1	0
	Decrement 2	,--R	1rr00011	3/2	0
Constant offset from PC (Twos complement offset)	8 bit offset	n,PC	1xx01100	1	1
	16 bit offset	n,PC	1xx01101	5/3	2
*Relative to W	No Offset	,W	10001111	0	0
*(Twos complement offset)	16 bit offset	n,W	10101111	5/2	2
* AutoIncrement W	Increment 2	,W++	11001111	3/1	0
* AutoDecrement W	Decrement 2	,--W	11101111	3/1	0
Indirect Modes					
Constant offset from R	No offset	[,R]	1rr10100	3	0
	5 bit offset	[n,R]	Defaults to 8 bit		
	8 bit offset	[n,R]	1rr11000	4	1
	16 bit offset	[n,R]	1rr11001	7	2
Accumulator offset from R (Twos complement *offset)	A - Register	[A,R]	1rr10110	4	0
	B - Register	[B,R]	1rr10101	4	0
	E - Register	[E,R]	1rr10111	1	0
*	F - Register	[F,R]	1rr11010	1	0
	D - Register	[D,R]	1rr11011	4	0
*	W - Register	[W,R]	1rr11110	4	0
Auto Increment and decrement of R	Increment 2	[,R++]	1rr10001	6	0
	Decrement 2	[,--R]	1rr10011	6	0
Constant offset from PC (Twos complement offset)	8 bit offset	[n,PC]	1xx11100	4	1
	16 bit offset	[n,PC]	1xx11101	8	2
Extended indirect	16 bit address	[n]	10011111	5	2
*Relative to W	No Offset	[,W]	10010000	0	0
*(Twos complement offset)	16 bit offset	[n,W]	10110000	5	2
* AutoIncrement W	Increment 2	[,W++]	11010000	3	0
* AutoDecrement W	Decrement 2	[,--W]	11110000	3	0

rr = X, Y, U or S X = 00 Y = 01
xx = Doesn't care U = 10 S = 11
+ and + indicates the additional number of cycles and bytes for the
~ # particular variation

Register Descriptions

```
 ____________________________________________________________________________
| X  - 16 bit index register                                                 |
| Y  - 16 bit index register                                                 |
| U  - 16 bit user-stack pointer                                             |
| S  - 16 bit system-stack pointer                                           |
| PC - 16 bit program counter register                                       |
|*V  - 16 bit variable register (inter-register instructions only)           |
|*0  - 8/16 bit zero register   (inter-register instructions only)           |
|----------------------------------------------------------------------------|
| A  - 8 bit accumulator             |                                       |
| B  - 8 bit accumulator             |    Accumulator structure map:         |
|*E  - 8 bit accumulator             |      ----- ----- ----- -----          |
|*F  - 8 bit accumulator             |     |  A  |  B  |  E  |  F  |         |
| D  - 16 bit concatenated reg.(A B) |      -----------+-----------          |
|*W  - 16 bit concatenated reg.(E F) |     |     D     |     W     |         |
|*Q  - 32 bit concatenated reg.(D W) |      -----------------------          |
|------------------------------------|     |           Q           |         |
|*MD - 8 bit mode/error register     |      -----------------------          |
| CC - 8 bit condition code register | bit  31   24    15    8     0         |
| DP - 8 bit direct page register    |                                       |
 ----------------------------------------------------------------------------
```

* Indicates new registers in 6309 CPU.

Push/Pull Order of Stack

```
Pull order                  Push/Pull Post byte
    |                      --------------------------------
   \|/                    | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
    '                      --------------------------------
    CC                     |   |   |   |   |   |   |   |____CCr
    A\                     |   |   |   |   |   |   |________A
    B/ D\ Q                |   |   |   |   |   |_____________B
    E\ W/                  |   |   |   |   |_________________DPr
    F/                     |   |   |   |_____________________X
    DP                     |   |   |_________________________Y
   X-hi                    |   |_____________________________S/U
   X-low                   |_____________________________PC
   Y-hi
   Y-low
   U/S-hi
   U/S-low
   PC-hi
   PC-low
     .
    /|\
     |
 Push order
```

Condition Code Register

```
                          --------------------------------
                         | E | F | H | I | N | Z | V | C |
                          --------------------------------
              Entire flag____|   |   |   |   |   |   |   |____Carry flag
                FIRQ mask____|   |   |   |   |   |   |________Overflow
               Half carry____|   |   |   |   |_____________Zero
                 IRQ mask____|   |   |_____________________Negative
```

The PSH(s,u) and PUL(s,u) instructions require one additional cycle for each byte pushed or pulled.

Alan DeKok's addition to the above...

The new features of the 6309 are closely related to the changes in design from the 6809. The 6309 is micro-coded, which allowed the designers to easily add new instrctions and registers. It also has a one byte pre-fetch 'cache', and an internal pipeline. The 'cache' enables the 6309 to execute instructions like 'lsld' (2-bytes) in one clock cycle. The design of the 6809 series allows them to read one byte per clock cycle MAXIMUM, but there is a catch. Most instructions take more clock cycles to execute than bytes they contain. While the 6309 is performing internal calculations, the 'cache' hardware goes and reads the next instruction byte, leaving only one additional byte to be read to execute the 'lsld'. Reading this byte requires one clock cycle, and then the instruction is executed while the CPU fetches the next instruction.

The 6309 has a true 16-bit internal design.
e.g. the EXG instruction operates as

```
6809: read op-code
      read inter-register byte (r0,r1)
        r0_high -> temp_high
        r0_low  -> temp_low
        r1_high -> r0_high
        r1_low  -> r0_low
        r0_high -> r1_high
        r0_low  -> r1_low
```

8 actions, 8 clock cycles.

```
6809: read op-code
      read inter-register byte (r0,r1)
        r0 -> temp
        r1 -> r0
        r0 -> r1
```

5 actions, 5 clock cycles.

The 6309 native mode instruction execution clock lengths can be mostly accounted for by accounting for the pre-fetch cache and the internal 16-bit ALU.

TFM has some caveats. TFM r1-,r2- should NOT be used to setup the stack, as it's a POST-decrement instruction, not PRE-decrement.

Watch out for TFM r1,r2+ if you're reading from a peripherial. Why? The TFM uses the 1-byte 'cache' as an internal buffer for the byte that it's currently moving. The TFM instruction is interruptible (the only instruction that is), and code execution during the interrupt will destroy the byte in the cache.

On returning from the interrupt, the TFM will read the FROM address again to get the lost byte, which may be the wrong one. The visible effect of this is that block moves sometimes have a byte missing from the middle, and everything after that byte shifted down one address.

The W,E, and F registers do not have the full immediate addressing mode capabilities that D,A, and B do. SBC, AND, BIT, EOR, ADC, OR with E,F,W are available only in register-register mode. LSR, ROR, ROL are available for W but not for E,F. ASR, ASL, LSL, NEG do not exist at all for W,E,F.

ASL can sort of be implemented by doing a ADDR R1,R1. (see later)

You can also do things like 'leax u,x' by doing a ADDR u,x.

Sadly, many of the new 6309 instructions are useless in everyday life. The bit manipulation instructions are interesting, but slow and mostly of limited value. Same with much of the DIV/MUL higher math. The AIM, etc. are very useful, though.

Programmer recommendations

Try to stay away from using the W register. It's got another pre-byte (like instructions using 'Y' or 'S'), and is correspondingly larger and slower. E and F are best used mainly instead of pushing loop counters onto the stack when you're running out of registers.

The V register is mostly pointless. If you're doing context switches, it isn't saved across interrupts unless you do so manually. Shuffling data back and forth between other registers and V is a lot of trouble. Any math, etc. involving V is generally done much faster using a real register. After going through 1meg+ of 6309 assembly code which is everything from an OS kernel to serial drivers to graphics drivers, I've never seen a use for the V register.

Of course, you could put '$FFFF' into V, and have registers for reg-reg addressing modes with bits all zero (0), and another with bits all 1 (V).

Pseudo-nops: tfr 0,0; exg 0,0

Extremely small software timing loops with large delays may be generated by performing a 'LDW',and then 'TFM 0+,0+'.

Many programs can be executed in 6309 native mode by patching only the IRQ code, if it accesses the stack. A 'LDMD #$01' may be performed as soon as your program starts executing, and will see an immediate 10-15% speed increase. Software timing loops must be checked!

Opcodes Hitachi left out of the 6309: and some round-about equivalents

```
E/F/W
-----
ADCr:  ADCR 0,r
ANDr:  ; ANDR V,r
ASLr/LSLr:  ADDR r,r
ASRr
BITr
EORr
NEGr:  COMr INCr
ORr
SBCr:  SBCR Z,r


E/F
---
LSRr
ROLr: ADCR r,r
RORr

Q (Long word =W1:W0)
--------------------
ADDQ: ADDW W0; ADCD W1
SUBQ: SUBW W0; SBCD W1
ASLQ: ASLW   ; ROLD
ROLQ: ROLW   ; ROLD
LSRQ: LSRD   ; RORW
RORQ: RORD   ; RORW
ASRQ: ASRD   ; RORW
COMQ: COMD   ; COMW
NEGQ: COMD   ; COMW ; SBCR 0,D
```