MEMORANDUM

Distribution: XAP2 Release

Prepared By: Steve Allen

Subject: Macro Assembler User Manual

XAP2

MACRO ASSEMBLER
USER MANUAL

CONTENTS:

1 Introduction 6

2 Using the Macro Assembler 7
2.1 Getting Started 7
2.1.1 Assembly Process 8
2.1.2 Invoking the Assembler 9
2.1.3 Typical Program Structure 10
2.2 Memory Models 15
2.2.1 Selecting the Memory Model 15
2.2.2 Memory Model Differences 15
2.2.3 Memory Model Compliance Directives 16
2.2.4 Memory Model Variables 16
2.3 Address Modes 17
2.3.1 Address Mode Formats 17
2.3.2 Variable Length Instructions 18
2.4 Segments 20
2.4.1 Pre-Defined Segments 20
2.4.2 User Data Segments 21
2.4.3 User Code Segments 21
2.4.4 Location Counter 22
2.4.5 Segment Packing 22
2.4.6 ROM Start Address 22
2.5 Modules 26
2.5.1 Defining Modules 26
2.6 Symbols 28
2.6.1 Local Symbols 28
2.6.2 Global Symbols 28
2.6.3 Symbol Names 28
2.6.4 Symbol Classes 28
2.7 Labels and Assembler Variables 30
2.7.1 Labels 30
2.7.2 Assembler Variables 31
2.8 Numeric Data Types 33
2.8.1 Floating Point Arithmetic 33
2.8.2 Fractional Arithmetic 33
2.8.3 Long Integers and Long Fractionals 34
2.8.4 Allowed Number Ranges 35
2.9 Defining and Initializing Data 36
2.9.1 String Formats 36
2.9.2 Number Formats 37
2.9.3 Segment Usage 37
2.9.4 Forward References 38
2.10 Operands, Operators and Expressions 39
2.10.1 Operands 39
2.10.2 Operators 39
2.10.3 Arithmetic Operators 40
2.10.4 Bitwise Operators 41
2.10.5 Relation Operators 41

2.10.6 Logic Operators 42
2.10.7 Assignment Operators 43
2.10.8 Operator Precedence 46
2.10.9 Expressions 46
2.11 Structures 47
2.11.1 Defining Structure Types 47
2.11.2 Defining Structures 48
2.11.3 Embedded Equates 49
2.12 Macros 53
2.12.1 Using Macros 53
2.12.2 Defining Macros 53
2.12.3 Calling Macros 54
2.12.4 Using Local Macro Symbols 57
2.12.5 Exiting from a Macro 57
2.12.6 Macro Operators 60
2.12.7 Macro Recursion 62
2.12.8 Nesting Macro Definitions 62
2.12.9 Nesting Macro Calls 62
2.12.10 Redefining Macros 62
2.13 Conditional Assembly 63
2.13.1 Conditional Directives 63
2.13.2 Forcing Errors 64
2.14 Interrupts 65
2.14.1 Hardware Functionality 65
2.14.2 Sequence/Operation of a Simple Interrupt 65
2.14.3 Example Interrupt Code 66
2.15 Miscellaneous 68
2.15.1 Generating Output 68
2.15.2 INCLUDE files 69
2.15.3 Original Line Number Propagation 69
2.15.4 The Continuation Character 70
2.15.5 Maximum Line Length 70
2.15.6 Forcing a Page Break in a Listing File 70
2.15.7 Scratchpad RAM Access Relative to IY Register 70
2.15.8 Source File Parsing 70
2.16 Error Message List 72

3 Linker Support 76

4 Instruction Set 78

5 Key Words 126

6 Index 128

FIGURES:

Figure 2.1.1-1 The Assembly Process....8
Figure 2.1.3-1 Program “main.xap”....10
Figure 2.1.3-2 Code Table “main.xpv”....11
Figure 2.1.3-3 Constant Table - “main.xdv”....11
Figure 2.1.3-4 Code Table “main.xpv” with comments suppressed....12
Figure 2.1.3-5 Symbol Table - “main.sym”....12
Figure 2.1.3-6 Listing “main.lst”....13
Figure 2.1.3-7 MACRO expansion example....14
Figure 2.2.4-1 Use of Memory Model Variables....16
Figure 2.4.4-1 Using the location counter to size a string....22
Figure 2.4.6-1 Segment Example Code....23
Figure 2.4.6-2 Segment Example Listing....24
Figure 2.4.6-3 Segment Example Listing continued....25
Figure 2.5.1-1 Module Example Code....27
Figure 2.7.1-1 Label Example Code....30
Figure 2.7.2-1 Numeric Equates....31
Figure 2.7.2-2 String Equates....32
Figure 2.7.2-3 Redefinable Equates....32
Figure 2.8.4-1 Defining Data....36
Figure 2.9.4-1 Forward Reference Example....38
Figure 2.10.3-1 Arithmetic Operator Example....40
Figure 2.10.4-1 Bitwise Operator Example....41
Figure 2.10.6-1 Logic Operator Example....42
Figure 2.10.7-1 Assignment Example Code....44
Figure 2.10.7-2 Assignment Example Listing....45
Figure 2.10.8-1 Operator Precedence....46
Figure 2.11.2-1 Structure Operators Example....48
Figure 2.11.3-1 Embedded Equates Example....49
Figure 2.11.3-2 Struct Example Code....50
Figure 2.11.3-3 Struct Example Listing....51
Figure 2.11.3-4 Struct Example Listing (continued)....52
Figure 2.12.3-1 Macro Example Code....55
Figure 2.12.3-2 Macro Example Listing....56
Figure 2.12.5-1 Macro Example 2 Code....58
Figure 2.12.5-2 Macro Example 2 Listing....59
Figure 2.12.6-1 Macro Substitute Operator Example....60
Figure 2.12.6-2 Macro Expression Operator Example....61
Figure 2.15.1-1 %OUT Example....68
Figure 2.15.4-1 Continuation Character Example....70
Figure 3-1 Global Symbols File Example....76
Figure 3-2 Segment Size File Example....77

BOXES:

Box 2.2.3-1 Memory Model Compliance Directives....16
Box 2.3.1-1 General Instruction Format....17
Box 2.3.1-2 Branch Instruction Format....17
Box 2.3.2-1 Variable Length Instructions....18
Box 2.3.2-2 Prefix Pseudo Code....19
Box 2.4.1-1 Pre-Defined Segments....20
Box 2.9.3-1 Segment Usage....37
Box 2.10.3-1 Arithmetic Operators....40
Box 2.10.4-1 Bitwise Operators....41
Box 2.10.5-1 Relation Operators....41
Box 2.10.6-1 Logic Operators....42
Box 2.12.6-1 Macro Operators....60
Box 4-1 Data Instruction Format....78
Box 4-2 Branch Instruction Format....78

1 INTRODUCTION

The ‘xap2asm’ Macro Assembler forms an integral part of the XAP2 ASIC Processor development toolkit. It can be used as a stand-alone assembler or in the final processing stage of the ‘C’ programming environment – the ANSI ‘C’ compiler generates assembler as its output.

The philosophy behind the XAP2 has been to create a low-risk environment in which to develop low cost, low power, embedded solutions. For this reason a “keep-it-simple” strategy has been adopted through-out the assembler, allowing the programmer to remain close to the physical hardware model. The more powerful abstract features such as Macros, Conditional Assembly and Expression Evaluation can then be used to aid the readability and understanding of the code, as and when required.

The key features of the Macro Assembler can be summarized as:

- Concatenated assembly of all source files - no linker.
- Microsoft® MASM style macro multi-pass assembler.
- Powerful expression evaluator.
- Conditional assembly.
- INCLUDE files supported.
- STRUC type supported.
- Original line number propagation from ‘C’ compiler.
- Symbol Table and Code files generation for the Simulator / Emulator.

2 USING THE MACRO ASSEMBLER

The XAP2 Macro Assembler is a multi pass assembler and linker. This section deals with the users' view of the assembler, and covers the following areas:

- Getting Started
- Memory Models
- Addressing Modes
- Using Segments
- Creating Programs from Multiple Modules
- Defining Labels and Assembler variables
- Using Structures
- Using Operands and Expressions
- Macros
- Conditional Assembly
- Interrupts

General facilities which provide support for external linkers are discussed in Section 3.

2.1 Getting Started

The assembler is invoked from the command line with a list of source files and an optional output base name (by default the first source file is used as the base name for the output files). The source files are assembled and linked sequentially in a single operation to form a single executable program. The source file list can contain files written by the user and/or automatically generated by the XAP2 'C' compiler.

The assembler uses a multi-pass process to facilitate forward references and variable length instructions. On the first pass the symbol table is built (initial substitutions made); macros and conditional assembly directives are expanded; and the files are concatenated into a single working file. On subsequent passes the assembler fills in the forward references and recalculates labels and branches to account for changing instruction sizes (as instruction operands cross 8 bit boundaries – see Section 2.3.2).

Assembly continues until either: (a) the number of code size changes falls below a user definable threshold; (b) the maximum number of passes is reached (both values can be set by command line arguments); or (c) a fatal error makes further assembly impossible (i.e. forward reference on a macro expansion). The working file is then automatically deleted.

If no errors are detected three files are generated as output of the assembly process - a symbol table file (.sym), a code ROM file (.xpv) and a constant ROM file (.xdv). The symbol table contains all code and data symbols used by the source files, the code ROM file contains the assembled code, and the constant ROM file contains the values for the constant ROM.

A further two files may be generated during the assembly process – an error log and a listing file. The error log contains a detailed description of any errors detected, and is only generated if errors are found. The listing file contains details of the assembled code along with a summary of equates, labels, modules and errors, and is only generated if the '-list' flag is set.

2.1.1 Assembly Process

The assembly process can be viewed schematically as follows :

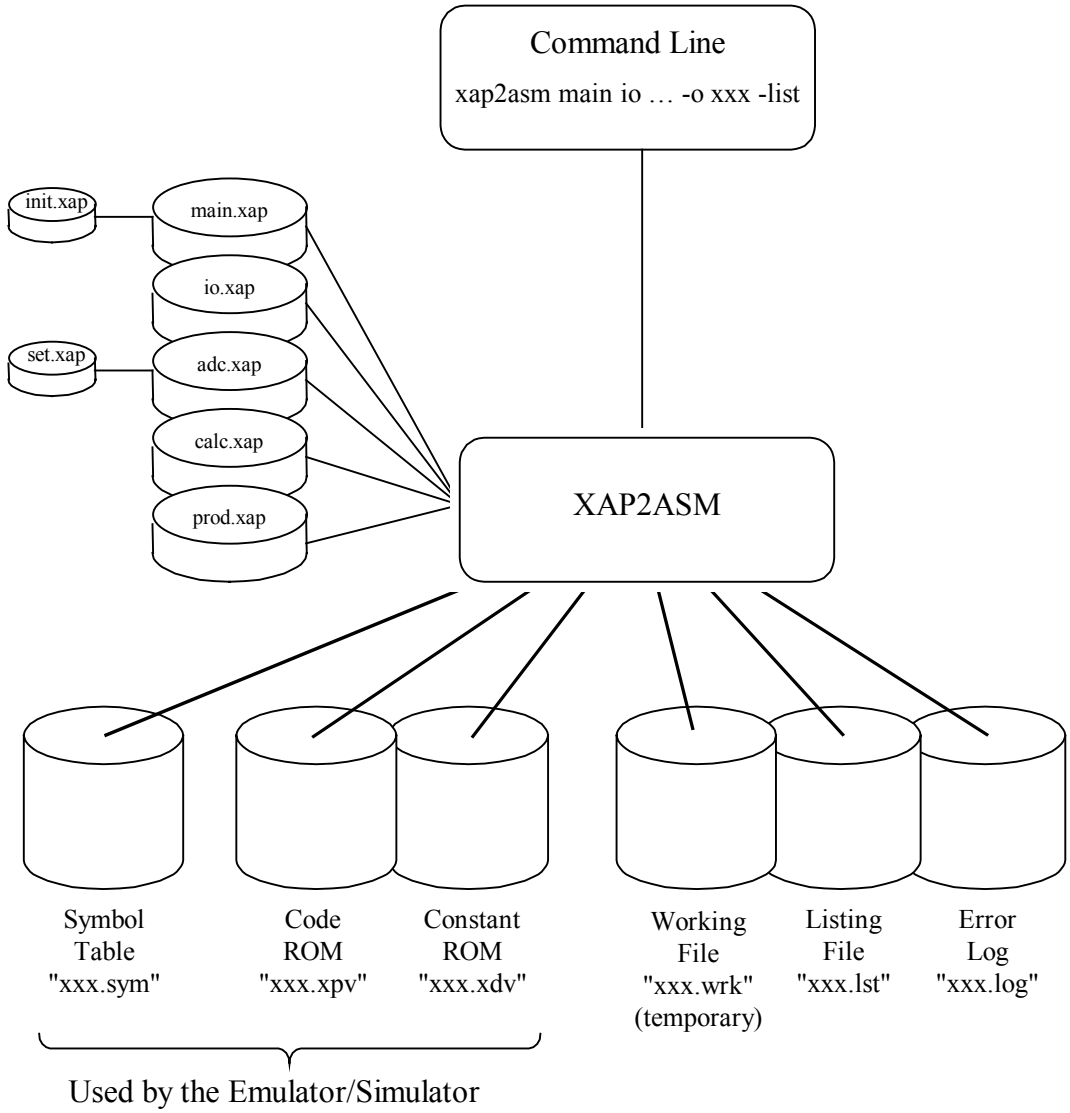


Figure 2.1.1-1 The Assembly Process

2.1.2 Invoking the Assembler

The assembler is invoked from the command line with a number of arguments.

xap2asm src_file1 [src_file2 ...] [-cmdfile file] [-list] [-supmac] [-supinc] [-supcmt] [-quiet] [-o out_file] [-c|s] [-small|large] [-pass n] [-opt n] [-code n]

Arguments:

src_file1	- first source file.
src_file2 ...	- additional source files.
-cmdfile ‘file’	- read arguments from 'file'.
-quiet	- suppress the screen banner.
-list	- generate list file.
-supmac	- suppress macro expansion in the list file.
-supinc	- suppress include file expansion in the list file.
-supcmt	- suppress comments in output files (.lst, .xpv, .xdv)
-o ‘out_file’	- set the output base name to 'out_file'.
-c	- generate global symbol list. (Linker support only – see Section 3)
-s	- generate segment sizing info. " "
-small	- small memory model.
-large	- large memory model.
-pass n	- set maximum number of passes (default 6)
-opt n	- set optimisation size (acceptable change in code size per pass)
-code n	- set the number of address bits for code.

- [] indicates an optional argument.
- If no arguments are given, a command format message is displayed.
- The output file name (‘out_file’) provides the base part of the output file names only. The extensions (.xpv, .xdv, .wrk, .err, .lst, .lif) cannot be changed.

The -small and -large flags specify the memory model to be used in the assembly process. The small memory model assumes a code address space of 16 bits, and the large model assumes a code address space of 24 bits. These values can be overridden with the –code n flag. If a code address space of greater than 16 bits is set then the assembler defaults to the large memory model, otherwise a small model is assumed.

If the memory model is defined (i.e. -small, -large, or -code flags are set) then every module within the source code must contain a memory model compliant directive (.SMALL, .LARGE, or .ALL – for small, large and all memory models respectively – see Section 2.2.3).

2.1.3 Typical Program Structure

A typical program will contain :

- MODULE definition.
- Memory Model Compliance directive.
- DATA definitions.
- CODE statements.
- ENDMOD statement.

```
                MODULE   test           ; define MODULE 'test'
                .ALL

                .DATA                   ; select DATA segment
var1            DS    1                 ; var1 at address H'0000
var2            DS    1                 ; var2 at address H'0001

                .CODE                   ; select CODE segment
start:          ld    al,#H'3F
                st    al,@var1          ; store H'3F in var1

                ld    al,#B'100101      ; load al with D'37
                add   al,@var1
                st    al,@var2          ; store result in var2

                ENDMOD   test           ; END module
```

Figure 2.1.3-1 Program "main.xap"

Invoking the assembler with 'xap2asm main' produces the following output:

```
XAP2 Assembler Vx.xx  (xx/xx/xx)
(c) Cambridge Consultants Ltd. 1999
Pass 1, 2(0), 3(0), 4(0), 5(0)
NO ERRORS FOUND
```

- The numbers in brackets along side each pass indicate the code size changes that have occurred in that particular pass. Assembly continues until the maximum number of passes have been reached (default 6), or two consecutive passes have generated no code size changes (a final pass is then needed to produce the output). A minimum of 5 passes are usually required to safely accommodate code size changes and packed code segments.

Code and Constant Table files are produced in a format compatible with Verilog ROM files :

```
// <Header>
// ROM START ADDRESS xx
// WORDSIZE xx
// BASE HEX : HEX
<Address>   <Code/Data> // <Source line>
```

(// = Verilog comment character)

```
// Version      : XAP2 Assembler - Vx.xx (xx/xx/xx)
// Title        : "Code Table"
// Date         : Fri Sep 17 11:49:29 1999
// Command Line : xap2asm main
//
//
// ROM START ADDRESS 000000
// WORDSIZE 16
// BASE HEX : HEX

// CODE - Start Address H'0; Size H'5

@000000   3F14 // start:               ld         al,#H'3F
@000001   0025 //                      st         al,@var1
@000002   2514 //                      ld         al,#B'100101
@000003   0035 //                      add        al,@var1
@000004   0125 //                      st         al,@var2
```

Figure 2.1.3-2 Code Table "main.xpv"

```
// Version      : XAP2 Assembler - Vx.xx (xx/xx/xx)
// Title        : "Constant Table"
// Date         : Fri Sep 17 11:49:29 1999
// Command Line : xap2asm main
//
//
// ROM START ADDRESS 0000
// WORDSIZE 16
// BASE HEX : HEX
```

Figure 2.1.3-3 Constant Table - "main.xdv"

The '// ROM START ADDRESS' line is a special comment used by the Simulator / Emulator to identify the start address of the ROM (see Section 2.4.5). The addresses in the body of the file are defined relative to the start address. The -supcmt flag can be used to suppress the comments and produce a smaller file (see Figure 2.1.3-4).

```
// Version      : XAP2 Assembler - Vx.xx (xx/xx/xx)
// Title        : "Code Table"
// Date         : Fri Sep 17 11:56:34 1999
// Command Line : xap2asm main -supcmt
//
//
// ROM START ADDRESS 000000
// WORDSIZE 16
// BASE HEX : HEX

// CODE - Start Address H'0; Size H'5

@000000   3F14
@000001   0025
@000002   2514
@000003   0035
@000004   0125
```

Figure 2.1.3-4 Code Table "main.xpv" with comments suppressed

A Symbol Table containing a list of all CODE and DATA symbols (along with their associated addresses) is produced for the symbolic debugger in the Simulator/Emulator (see Figure 2.1.3-5). CODE symbols are generated for each line of code, allowing the symbolic debugger to access the assembled program's original source code (both assembler and 'C'). It is also possible to manually define symbols using the CEQU and DEQU directives (see Section 2.7.2), thus allowing application specific hardware registers, or label aliases, to be accessed directly from the Simulator/Emulator.

```
// Version      : XAP2 Assembler - Vx.xx (xx/xx/xx)
// Title        : "Symbol Table"
// Date         : Fri Sep 17 11:49:29 1999
// Command Line : xap2asm main
//
//
.CODE
test.start, 0000
test.'main.xap'.?8.1, 0000
test.'main.xap'.?9.1, 0001
test.'main.xap'.?11.1, 0002
test.'main.xap'.?12.1, 0003
test.'main.xap'.?13.1, 0004
eof_label.'main.xap'.?999999.1, 0005
.DATA
test.var1, 0000
test.var2, 0001
.END
```

Figure 2.1.3-5 Symbol Table - "main.sym"

By setting the -list option on the command line (see Figure 2.1.3-6) a listing file is produced with the following format:

```
<Header>
<Line Num> <Address> <Segment> <Data> : <Source line>
<Summary>
<Error Summary>
```

where segments have the following codes:

: D - data segments (.DATA, .VAR)
R - ROM constant segment (.CONST)
U - user-defined data segments
C - code segment (.CODE & user-defined)

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx (xx/xx/xx)                Fri Sep 17 11:53:35 1999  #
#                                                                              #
# Command Line : xap2asm main -list                                            #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: main.xap

    1               :                       MODULE     test                          ; define MODULE 'test'
    2               :                       .ALL
    3               :
    4               :                       .DATA                                    ; select DATA segment
    5 000000 D ???? : var1                  DS         1                             ; var1 at address H'0000
    6 000001 D ???? : var2                  DS         1                             ; var2 at address H'0001
    7               :
    8               :                       .CODE                                    ; select CODE segment
    9 000000 C 3F14 : start:                ld         al,#H'3F
   10 000001 C 0025 :                       st         al,@var1                      ; store H'3F in var1
   11               :
   12 000002 C 2514 :                       ld         al,#B'100101                  ; load al with D'37
   13 000003 C 0035 :                       add        al,@var1
   14 000004 C 0125 :                       st         al,@var2                      ; store result in var2
   15               :
   16               :                       ENDMOD     test                          ; END module


SUMMARY
-------

MODULES : 1

test                            : Start Add = H'0000, Size = H'0005

CODE SYMBOLS : 1

test.start                      : Address = H'0000

DATA SYMBOLS : 2

test.var1                       : Address = H'0000
test.var2                       : Address = H'0001

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 3

$??XAP2                         : Value = H'00000001, D'1
$??SMALL                        : Value = H'00000001, D'1
$??LARGE                        : Value = H'00000000, D'0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

CODE                            : Segment Start Add = H'0000, Size = H'0005
DATA                            : Segment Start Add = H'0000, Size = H'0002
VAR                             : Segment Not Used
CONST                           : Segment Not Used


ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

Figure 2.1.3-6 Listing "main.lst"

Other Codes

When a source line produces no ROM output, the listing file has the following format:

```
<Line Num> <List Code> : <Source line>
```

where:

<List Code>:	X	- source line not processed due to conditional assembly.
	M	- macro definition.
	S	- struc definition
	–	- non output producing directive.

Macro Expansion

Macro expansions are indented, and high-lighted with a "*", in the listing file (see Figure 2.1.3-7).

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx (xx/xx/xx)                Fri Sep 17 12:07:45 1999  #
#                                                                              #
# Command Line : xap2asm macro -list                                           #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: macro.xap

    1               :                       MODULE      macro
    2               :
    3 M             : first                 MACRO
    4 M             :                       ld          x, #23
    5 M             :                       second                                  ; nested MACRO
    6 M             :                       ENDMAC
    7               :
    8 M             : second                MACRO
    9 M             :                       st          x, @H'02
   10 M             :                       ENDMAC
   11               :
   12               :                       .CODE
   13               :                       first                                   ; call MACRO first
    . 000000 C 1718 :                       * ld        x, #23
    .               :                       * second                                ; nested MACRO
    . 000001 C 0229 :                       * st        x, @H'02
    .               :                       * ENDMAC
    .               :                       * ENDMAC
   14               :                       ENDMOD
```

Figure 2.1.3-7 MACRO expansion example

2.2 Memory Models

The XAP2 has two memory models: large and small.

- The (default) small memory model has a 16 bit code and data space. This is identical to the memory model used on the XAP.
- The large memory model has a 24 bit code and 16 bit data space. This model has been introduced to cater for large applications which require more than the 16 bits of code space available in the small memory model.

2.2.1 Selecting the Memory Model

The assembler automatically defaults to the small memory model and assumes small memory model compliance for all modules. The large memory model is selected by setting the -large flag on the command line when invoking the assembler (see Section 2.1.2). Memory model compliance checking can be forced in the small model by setting the -small flag on the command line.

The memory model can also be defined by setting the physical code size with the -code n flag (this should only be done if you are sure that the hardware supports the specified physical code size). The assembler uses the large memory model for code sizes greater than 16 bits, and the small model for code sizes less than or equal to 16 bits. Setting the code size overrides the default values (16 for small and 24 for large memory models), allowing the assembler to check for physical code size violations and take advantage of code address wrap-around – i.e. when only 17 bits of code space are defined in hardware, a branch of +128,000 is the same as a branch of -3,072.

If the memory model is explicitly defined (i.e. -small, -large, or -code flags are set), then every module within the source code must contain a memory model compliant directive (.SMALL, .LARGE, or .ALL – for small, large and all memory models respectively).

2.2.2 Memory Model Differences

Although the two memory models are very similar, there are some subtle differences that need to be taken into account – i.e. the small memory model has no awareness of the xh register[1]. These differences could result in unexpected behaviour if unmodified code written for the small memory model is unwittingly used within the large memory model. For this reason memory model compliance directives have been introduced (see Section 2.2.3).

Extending the address space to 24 bits has two implications on the memory model: (1) the 24 bit return address for the *Branch to Sub-Routine* (BSR) instruction no longer fits in the 16 bit x register; and (2) in general we need 24 bits for branch addresses. For these reasons the large memory model introduces a new xh register to store the top 8 bits of the address in the BSR instruction and hold the top 8 bits in the different addressing mode. This register is not accessible in the small memory model, and is initialised to the value zero.

The programmer should remain vigilant to the fact that the xh register is only guaranteed to be zero in the small memory model, and that any code written for the large memory model must treat the xh register as part of the x register – i.e. store the register on entering a sub-

[1] The xh register is only accessible in large memory model compliant modules – i.e. within modules containing a .LARGE or .ALL directive.

routine and set the register in the *Relative X*, *Indexed Y*, and *Indirect* addressing modes. Although it is desirable for code to be compliant with both memory models, storing and restoring the xh register adds a non-insignificant code overhead. The small memory model is therefore still valid in applications that use on-chip ROM – where code size is of a premium – , and where it is unlikely that the code will be migrated to the large memory model.

2.2.3 Memory Model Compliance Directives

The memory model compliance directives are used to inform the assembler (and user) of the compliance of a particular module. The directives are:

Directive	Meaning
.SMALL	The module is small memory model compliant.
.LARGE	The module is large memory model compliant.
.ALL	The module is memory model independent, i.e. compliant with both small and large memory models.

Box 2.2.3-1 Memory Model Compliance Directives

If the large memory model is used then all assembled modules must use either the .LARGE or .ALL directives. The .SMALL (or .ALL) directive is only required if the code is assembled with the -small or -code n (where n ≤ 16) flags defined – i.e. by default the assembler assumes small memory model compliance and requires no explicit compliance directives.

2.2.4 Memory Model Variables

The assembler defines two global variables ($??SMALL and $??LARGE) to indicate the memory model used during the assembly process – the variables take the boolean values 1 = TRUE and 0 = FALSE. These variables can be used within conditional assembly expressions to conditionally assemble memory model specific code (see Figure 2.2.4-1). The variable $??XAP2 is set by the **xap2asm** assembler and can be used to conditionally assemble XAP2 only code.

```
                MODULE   memory          ; define MODULE 'memory'
                .ALL                     ; compliance

                .CODE                    ; select CODE segment
start:          st    x, @(0, y)

                IF    $??LARGE           ; conditional assembly
                  st  xh, @(-1, y)       ;  in Large memory model
                ENDIF

                .
                .
                .
                ENDMOD                   ; END module
```

Figure 2.2.4-1 Use of Memory Model Variables

2.3 Address Modes

The XAP supports the following address modes :

- Immediate - load/store immediate data
- Direct - load/store/branch from/to the contents of address (arg)
- Indexed X - load/store the contents of address (arg + X)
- Indexed Y - load/store the contents of address (arg + Y)
- PC Relative - branch to address (arg + PC)
- X Relative - branch to address (arg + X)

2.3.1 Address Mode Formats

The assembler encodes the address modes in the following formats:

Name	Assembler Syntax	'Data' (Source / Destination)
Immediate	#arg	arg^{16}
Direct	@arg	Contents of address (arg^{16})
Indexed X	@(arg, X)	Contents of address (arg^{16} + X reg)
Indexed Y	@(arg, Y)	Contents of address (arg^{16} + Y reg)

Box 2.3.1-1 General Instruction Format

Name	Assembler Syntax	'Branch Address' (new Program Counter value)
PC Relative	arg	PC + arg^{24}
Direct	@arg	{XH, Contents of address (arg^{16})}
X Relative	arg, X	{XH, X} + arg^{24}
Indexed Y	@(arg, Y)	{XH, Contents of address (arg^{16} + Y reg)}

Box 2.3.1-2 Branch Instruction Format

- @ indicates 'contents of'.
- # indicates 'immediate'.
- arg^{16} = 16 bit sign extended argument; arg^{24} = 24 bit sign extended argument
- {XH, X} indicates 24 bit address composed of {high 8 bits, low 16 bits}
- SP (Stack Pointer) can be used as an alias for Y.
- The program counter (PC) is not pre-incremented.

2.3.2 Variable Length Instructions

The XAP2 instructions have a nominal size of 16 bits (1 word), which is enough to directly encode an 8 bit operand. When operands greater than 8 bits are required, the assembler automatically inserts special *prefix* instructions to hold the extra bits of the operand. These *prefix* instructions are invisible to the user, and so it appears that the XAP2 instructions occupy a variable number of words - dependent on the size of the operand.

Instruction		Operand Size Requirement	Instruction Size
ld	x, #4	8 bits	1 word
st	y, #H'100	16 bits	2 words
bra	$+H'0004	8 bits	1 word
bra	$+H'FE00	16 bits	2 words
bra	$+H'120004	24 bits	3 words
bra2	H'0	don't care	2 words
bra3	H'100	don't care	3 words

Box 2.3.2-1 Variable Length Instructions

Two special branch instructions (`bra2` and `bra3`) have a fixed size of 2 and 3 words respectively. These instructions can be used for jump tables - where the index in to the table is reliant on the size of the table entries.

The *prefix* instruction is defined as a NOP with the data field at a non-zero value. The complete argument is formed by adding up each of the bytes which make up the complete 24 bit argument. All bytes are sign extended up to the MSB before adding. The three cases of 8, 16 and 24 bits are arranged as follows:

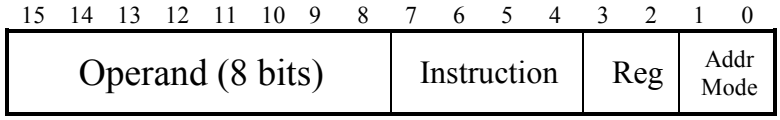
1. When only 8 bits of argument need to be defined (+127/-128) then it is included in the argument field of the instruction Opcode.
2. When 16 bits of argument need to be defined (+/-32K) then a single prefix is used which contains the most significant byte of the argument and the least significant byte is included in the argument field of the instruction Opcode.
3. When 24 bits of argument need to be defined (+/-8M) then two prefixes are used. The first prefix contains the most significant byte of the argument. This will always be non-zero defining it as a prefix as opposed to NOP. The second prefix will contain the middle significant byte and the least significant byte is included in the argument field of the instruction Opcode. In this case the XAP2 core will need to shift up the first prefix argument 8 bits before adding it to the second argument.

The operand and prefix arguments are calculated according to the pseudo code shown in Box 2.3.2-2.

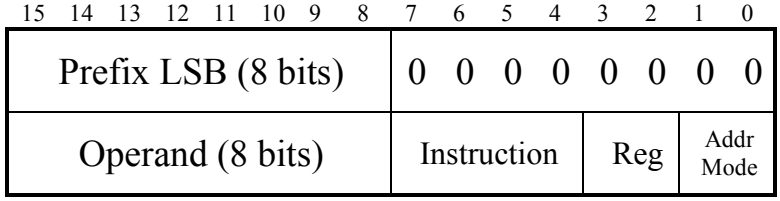
```
(assume char = 8 bits, short = 16-bits, true on PCs and Suns)

generate_prefix(long operand)
{
   char minimus = (char)operand;
   if (minimus != operand)
   {
      long remainder  = operand - minimus; // always a multiple of 256
      short minor = (short)remainder;
      if (minor != remainder)
      {
         long major = remainder - minor; // always a multiple of 65536
         printf("Prefix-MSB = %.6lx\n", major);
      }
      printf("Prefix-LSB=%.6lx\n", (long)minor);
   }
   printf("Operand=%.6lx\n",(long)minimus);
}
```

Box 2.3.2-2 Prefix Pseudo Code

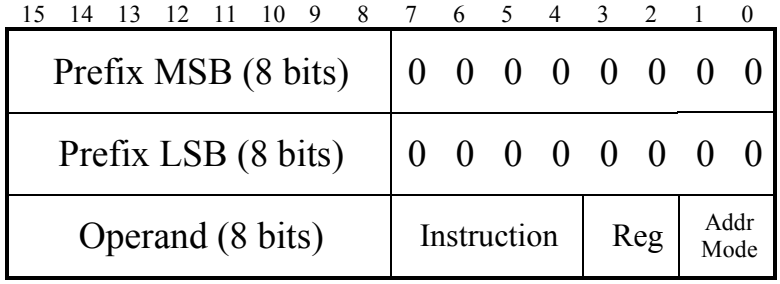
Instruction Formats:



8 bit Argument



16 bit Argument



24 bit Argument

2.4 Segments

Segments provide the programmer with a mechanism to group different data types together – simplifying the task of structuring a program's code and data space. Segments can best be thought of as blocks of memory reserved for specific uses – i.e. code, constants, look-up tables, structured arrays, IO space, or RAM variables. The programmer can freely switch between segments, and easily add items to the next available RAM/ROM location within a particular segment.

2.4.1 Pre-Defined Segments

The programmer has access to four pre-defined segments, allowing common data types to be grouped together. These pre-defined segments provide a basic structure for the address space, and can be selected with the following directives:

Directive	Segment	Use
.CONST	Constant	ROM Constants.
.DATA	Data	IO and large tables of constants / variables
.VAR	Variable	RAM Variables.
.CODE	Code	Program code.

Box 2.4.1-1 Pre-Defined Segments

The XAP2 uses a Harvard architecture with separate code and data address spaces. The DATA, VAR and CONST segments map on to the same 16 bit data space, whereas the CODE segment has its own code space (16 bits in the small memory model and 24 bits in the large memory model). The assembler checks for data space violations, and returns an error if the same address is used in different data segments.

Although the assembler supports multiple segments, only one segment can be selected at a time – the active segment. Items are always added to (and space reserved from) the active segment at a point defined by a special assembler variable called the location counter. Items can be added with the DC (Define Constant) directive, and space can be reserved with the DS (Define Storage) directive. XAP2 code instructions – i.e. bsr, ld, st, add – can only be added to the code segments.

Each segment maintains its own location counter. The pre-defined segment location counters are initialised to zero, and stored in the global variables $??CODE, $??DATA, $??VAR and $??CONST. The location counter for the active segment is also stored in the location counter variable – $. Reading a location counter effectively returns the next insertion address of the segment. The location counter variables can therefore be used within expressions to calculate sizes of arrays / strings etc.
It is possible to change the location counter of the active segment with the ORG directive. This address can be changed at any point after the segment has been selected, and as many times as required. By reassigning the location counter it is possible to accommodate gaps in the address space or place tables at specific locations.

```
ORG   address
```

The ORG directive can not be used in *Packed Segments* (see 2.4.5) – as addresses are then automatically assigned by the assembler – and restrictions are applied to the code segments to ensure that addresses can only ever increment. No restrictions apply to the DATA, VAR and CONST segments, which can be freely navigated.

As it is possible to freely navigate a data segment, it is also possible for segments to overlap in the assembler's data space without actually using the same addresses. An error is only generated if an actual address conflicts occurs between data segments, or code segments overlap. The boundaries of a segment can be seen in the listing file and with the assembler sizing command line option -s.

2.4.2 User Data Segments

In addition to the pre-defined data segments, any number of user-defined data segments can also be created. They share the same assembler address space (data space) as the DATA, VAR, and CONST segments, but maintain their own location counters and can be used to hold large arrays, pre-defined variables, tables etc. The segments are defined, and selected, using the .SEG directive; the first time the segment is encountered it is defined, subsequently it is simply selected.

```
.SEG    LOOKUP_TABLE
```

The current value of the segment's location counter is stored in the global variable $??*segment_name* and initialised to zero. To avoid a name space clash with pre-defined assembler variables (i.e. the Memory Model Variables described in Section 2.2.4, and $??XAP2), the segment names LARGE, SMALL, and XAP2 are reserved. An error will also be generated if the segment's location counter name clashes with a user defined variable of the format $??*xxx*.

The .SEG directive can also be used to select the pre-defined DATA, VAR, CONST segments. User defined data segments can be freely navigated with the ORG directive.

2.4.3 User Code Segments

In addition to the pre-defined CODE segment, any number of user-defined code segments can also be created. They share the same physical address space (code space) as the CODE segment, but maintain their own location counters. Code segments can be used to group related sub-routines together and position them at known points of the address space.

Code segments are defined, and selected, using the .CSEG directive; the first time the segment is encountered it is defined, subsequently it is simply selected.

```
.CSEG   IO_ROUTINES
```

The current value of the segment's location counter is stored in the global variable $??*segment_name* and initialised to zero. To avoid a name space clash with pre-defined assembler variables (i.e. the Memory Model Variables described in Section 2.2.4, and $??XAP2), the segment names LARGE, SMALL, and XAP2 are reserved. An error will also be generated if the segment's location counter name clashes with a user defined variable that matches the template $??*xxx*.

The .CSEG directive can also be used to select the pre-defined CODE segment. The location counter of user defined code segments can only be incremented with the ORG directive.

2.4.4 Location Counter

The location counter ($) is a special symbol that, during assembly, represents the address of the statement currently being assembled.

```
            .CONST
string      DC    P'Who wants to count every word in a string'
            DC    P'especially if it might be changed later'
lstring     DC    $ - string
```

Figure 2.4.4-1 Using the location counter to size a string

2.4.5 Segment Packing

Segments can either be manually positioned in the address space using the ORG directive, or automatically assigned an address by the assembler using the .PACK (for data space) and .CPACK (for code space) directives. Packing segments allows a program to make the best use of the available address space. Each segment packing directive specifies a list of segments to pack, a start address and an optional size:

```
.PACK     {Seg1, Seg2, Seg3, ...},   Start_Address
.PACK     {Seg1, Seg2, Seg3, ...},   {Start_Address, Size}

.CPACK    {Seg1, Seg2, Seg3, ...},   Start_Address
.CPACK    {Seg1, Seg2, Seg3, ...},   {Start_Address, Size}
```

Errors are generated if: the total size of a packed segment list exceeds the optional *Size* parameter; a segment is included in two lists; or an attempt is made to manually position a packed segment with an ORG directive.

2.4.6 ROM Start Address

ROM start addresses are stored in the header section of the code and constant table files. They are read by the Simulator / Emulator and used as an offset for the addresses in the rest of the file.

The code segment ROM start address is taken as the first address in a code segment to contain code. Code normally starts at H'0000, however this can be changed using either the ORG directive, packing the code with a new start address, or by reserving an area of ROM (from H'0000) with the DS directive. A non-zero start address will generate a warning.

The constant segment ROM start address is taken as the first address for which a constant is defined (DC) in the data space. A non-zero start address will generate a warning.

```
                .PACK   {DATA, INIT_STR}, H'FF00
                .CPACK  {IO_ROUTINES, CODE}, {H'0000, H'100}

                MODULE   first
                .ALL

                .DATA                   ; selects DATA segment
fred            DS    1                 ; reserve 1 word and label address fred

                .SEG IO
                ORG   H'FE80            ; set IO location counter to H'FE80
$io_space       DS    H'30              ; reserve space for IO

$timer_ctl      DEQU  H'FE80            ; define some io addresses
$timer_reg1     DEQU  H'FE81


; Note : the first use of the '.SEG' directive creates the segment
; subsequent uses select the same segment

                .SEG  INIT_STR          ; create the INIT_STR user-defined segment
$INIT_addr      =     $??INIT_STR       ; $INIT_addr = start of INIT_STR segment
time            DS    1
cost            DS    1
$INIT_len       =     $ - $INIT_addr    ; $INIT_len = the length of INIT_STR

; Note : $ and $??INIT_STR have the same value within the segment and can
; be interchanged

                .CONST                  ; select CONST segment
                ORG   H'0020            ; set CONST location counter to H'0020
v1              DC    H'3002            ; define a constant v1 with value H'3002

                .SEG  INIT_STR          ; select the INIT_STR user-defined segment
result          DS    1
$INIT_len       =     $ - $INIT_addr    ; $INIT_len = the new length of INIT_STR

                .CODE                   ; select CODE segment
                ld    x,#H'1234
                st    x,@fred           ; store x at H'FF00

                .CONST                  ; re-select CONST segment, = H'0021
v2              DC    'Hello'           ; store the string 'Hello'


                .CODE                   ; re-select CODE segment, = H'0002
                add   x,@($timer_reg1)  ; add the contents of H'FE81 to x

                ENDMOD   first

                MODULE   second
                .ALL

                .CSEG IO_ROUTINES
$read:          st    al, @$timer_ctl
                ld    ah, @$timer_reg1

                ENDMOD   second
```

Figure 2.4.6-1 Segment Example Code

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx (xx/xx/xx)                Thu Sep 23 16:47:35 1999  #
#                                                                              #
# Command Line : xap2asm segment -list                                         #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: segment.xap

    1                :                       .PACK      {DATA, INIT_STR}, H'FF00
    2                :                       .CPACK     {IO_ROUTINES, CODE}, {H'0000, H'100}
    3                :
    4                :                       MODULE     first
    5                :                       .ALL
    6                :
    7                :                       .DATA                                  ; selects DATA segment
    8 00FF00 D ????  : fred                  DS         1                           ; reserve 1 word and label address fred
    9                :
   10                :                       .SEG       IO
   11                :                       ORG        H'FE80                      ; set IO location counter to H'FE80
   12 00FE80 U ????  : $io_space            DS         H'30                        ; reserve space for IO
   13                :
   14                : $timer_ctl           DEQU       H'FE80                      ; define some io addresses
   15                : $timer_reg1          DEQU       H'FE81
   16                :
   17                :
   18                : ; Note : the first use of the '.SEG' directive creates the segment
   19                : ; subsequent uses select the same segment
   20                :
   21                :                       .SEG       INIT_STR                    ; create the INIT_STR user-defined segment
   22                : $INIT_addr           =          $??INIT_STR                 ; $INIT_addr = start of INIT_STR segment
   23 00FF01 U ????  : time                 DS         1
   24 00FF02 U ????  : cost                 DS         1
   25                : $INIT_len            =          $ - $INIT_addr              ; $INIT_len = the length of INIT_STR
   26                :
   27                : ; Note : $ and $??INIT_STR have the same value within the segment and can
   28                : ; be interchanged
   29                :
   30                :                       .CONST                                 ; select CONST segment
   31                :                       ORG        H'0020                      ; set CONST location counter to H*0020
   32 000020 R 3002  : v1                   DC         H'3002                      ; define a constant v1 with value H*3002
   33                :
   34                :                       .SEG       INIT_STR                    ; select the INIT_STR user-defined segment
   35 00FF03 U ????  : result               DS         1
   36                : $INIT_len            =          $ - $INIT_addr              ; $INIT_len = the new length of INIT_STR
   37                :
   38                :                       .CODE                                  ; select CODE segment
   39 000004 C 1200  :                      ld         x,#H'1234
    . 000005 C 3418  :
   40 000006 C FF00  :                      st         x,@fred                     ; store x at H'FF00
    . 000007 C 0029  :
   41                :
   42                :                       .CONST                                 ; re-select CONST segment, = H'0021
   43 000021 R 0048  : v2                   DC         'Hello'                     ; store the string 'Hello'
    . 000022 R 0065  :
    . 000023 R 006C  :
    . 000024 R 006C  :
    . 000025 R 006F  :
   44                :
   45                :
   46                :                       .CODE                                  ; re-select CODE segment, = H'0002
   47 000008 C FF00  :                      add        x,@($timer_reg1)            ; add the contents of H'FE81 to x
    . 000009 C 8139  :
   48                :
   49                :                       ENDMOD     first
   50                :
   51                :                       MODULE     second
   52                :                       .ALL
   53                :
   54                :                       .CSEG      IO_ROUTINES
   55 000000 C FF00  : $read:               st         al, @$timer_ctl
    . 000001 C 8025  :
   56 000002 C FF00  :                      ld         ah, @$timer_reg1
    . 000003 C 8111  :
   57                :
   58                :                       ENDMOD     second
```

Figure 2.4.6-2 Segment Example Listing

```
SUMMARY
-------

MODULES : 2

first                           : Start Add = H'0004, Size = H'0006
second                          : Start Add = H'0000, Size = H'0004

CODE SYMBOLS : 1

$read                           : Address = H'0000

DATA SYMBOLS : 7

first.fred                      : Address = H'FF00
$io_space                       : Address = H'FE80
first.time                      : Address = H'FF01
first.cost                      : Address = H'FF02
first.v1                        : Address = H'0020
first.result                    : Address = H'FF03
first.v2                        : Address = H'0021

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 5

$??XAP2                         : Value = H'00000001, D'1
$??SMALL                        : Value = H'00000001, D'1
$??LARGE                        : Value = H'00000000, D'0
$timer_ctl                      : Value = H'0000FE80, D'65152
$timer_reg1                     : Value = H'0000FE81, D'65153

REDEFINABLE EQUATE SYMBOLS : 2

$INIT_addr                      : Value = H'0000FF01, D'65281
$INIT_len                       : Value = H'00000003, D'3

STRING EQUATE SYMBOLS : 0

SEGMENTS : 7

CODE                            : Segment Start Add = H'0004, Size = H'0006
DATA                            : Segment Start Add = H'FF00, Size = H'0001
VAR                             : Segment Not Used
CONST                           : Segment Start Add = H'0020, Size = H'0006
INIT_STR                        : Segment Start Add = H'FF01, Size = H'0003
IO_ROUTINES                     : Segment Start Add = H'0000, Size = H'0004
IO                              : Segment Start Add = H'FE80, Size = H'0030



ASSEMBLY
--------

NO ERRORS FOUND
WARNINGS : 1

Warning     : Non-zero ROM Start Address
Description : Constant ROM Start Address = H'0020
```

Figure 2.4.6-3 Segment Example Listing continued

2.5 Modules

Modules provide the programmer with a fine-grain mechanism to group related pieces of code into self-contained units. These self-contained units form logical entities which are independent of the physical source files – a source file can contain a number of different modules, and a single module can span a number of different source files.

Modules are also independent of code and data segments - i.e. a single module can have code in a number of code segments (although it is more likely that modules restrict themselves to a single code segment for readability).

Every module provides the programmer with a local symbol name space for program labels and variables (see Section 2.6) – thus avoiding the problem of inter-module name conflicts. Communication between modules can be accomplished using a stack based call mechanism, or through shared global variables (it is also possible to access the local symbols of a module directly by prefixing the symbol name with '^*module*.', i.e. '^main.temp').

2.5.1 Defining Modules

Modules are surrounded by the MODULE and ENDMOD directives. The source file name is used as the module name if no other name is specified with the MODULE directive. Obviously if a file contains several modules, only one can be named in this way. The ENDMOD directive can have an optional module name, matching the MODULE directive.

```
MODULE  [name]
ENDMOD  [name]
```

Symbols declared in INCLUDE files (see Section 2.15.2) within the scope of a module remain local to that module – modules are the only divider. A module may contain any number of INCLUDE files.

```
                MODULE   main
                .ALL

                .CPACK   {CODE, CODE2}, H'0000

                .CONST
                ORG   H'0000
mess            DC    'Hello',0         ; local to module main
temp            DC    1                 ; local to module main

                .VAR
                ORG   H'FE00
$x              DS    1                 ; global variable

$ptr            =     2                 ; global redefinable equate

                .CODE
                bra   $calc
$return:        sleep                   ; global label
                ENDMOD   main

                MODULE   sort
                .ALL

                .CONST
temp            DC    H'0F01            ; local to module sort

                .CSEG CODE2
$calc:          ld    ah,@temp          ; ah = H'0F01
                st    ah,@$x            ; store AH in $x (declared in main)
                ld    al,#$ptr          ; al = 2
                ld    ah,@^main.temp    ; access temp in main with '^' prefix
                bra   $return
                ENDMOD   sort
```

Figure 2.5.1-1 Module Example Code

2.6 Symbols

2.6.1 Local Symbols

All labels, assembler variables, structures and structure type definitions are declared local to a module, unless prefixed by the special global identifier ($).

Symbols local to another module can be accessed by prefixing the symbol name with '^*module*.', i.e. '^main.temp'

2.6.2 Global Symbols

Global symbols are symbols prefixed by a '$'. They are accessible from any module in any file.

Macros are always treated as global.

2.6.3 Symbol Names

Symbol names can be of any length and :

- Start with 'A..Z', 'a..z', '_' , '?' or '$'.
- Contain only the following characters: 'A..Z', 'a..z', '0..9', '_' or '?'
- Symbol names conform to the following templates :-

$??name	- segment location counters and general assembler variables for global symbols which must not interfere with the 'C' name space.	
$??hex	- MACRO label.	(*reserved*)
$?name	- 'C' automatically generated global symbol.	(*reserved*)
?name	- 'C' automatically generated local symbol.	(*reserved*)
$name	- global variable / label.	
name	- local variable / label.	

- Are case sensitive.
- Accesses to local symbols in different modules use the following format:

  ^*module.name* - local symbol access

- Original line number labels have the following format:

  ^*module*.'*file_name*'.?*num.num* - original line number label

- Symbols must not match an OPERATOR or REGISTER (i.e. ah, al, x, y, sp, flags, ux, uy, xh) name. The comparison is made in a case insensitive way.

2.6.4 Symbol Classes

The assembler distinguishes between three different symbol classes:

- Labels, Assembler variables and Structures
- Structure Types and MACROs
- Structure Fields

Each symbol class has its own set of “Operators”, “Reserved Words” and “Name Space” (set of symbol names). Structure fields have a separate “Name Space” for each structure type.

It is possible for a structure, structure type and structure field all to have the same name, as they belong to different “Name Spaces”. Structure fields belonging to different structure types can also have the same name.

2.7 Labels and Assembler Variables

2.7.1 Labels

Labels give symbolic names to the addresses of instructions or data. These labels can be used as the operands to branch instructions, or as addresses in expressions. Labels can be created in both the code and data segments.

```
name:
```

```
                .CONST
table:                                  ; label (in data segment)
v1              DC    H'1000            ; label
v2              DC    H'10F3            ; label

                .CODE
                ld    x,#table          ; first element of table

                cmp   al,#5
                bne   not_equal         ; branch if al not equal to 5
                bra   equal             ; branch always

not_equal:      ...                     ; label (in code segment)
                ...
                bra   done

equal:          ...                     ; label
                ...
                bra   done
```

Figure 2.7.1-1 Label Example Code

Labels can also be created using the LABEL directive, or by preceding the DC (Define Constant) or DS (Define Storage) directive with a name.

```
name    LABEL
name    DC …
name    DS …
```

2.7.2 Assembler Variables

Assembler variables are used to store text strings or numeric values. There are three basic variable types:

- Numeric equates
- String equates
- Redefinable equates

Global variables are very useful for passing information between modules / files and to external linkers (see -c assembler option; Section 3). All variables are assigned at assembly time.

Numeric Equates

Numeric equates assign a numeric constant to a variable. Variables automatically record the type (integer or fractional) of the numeric constant they hold. The assembler replaces each occurrence of name with the value of expression, wherever it occurs within scope, whether before or after the definition.

```
name     EQU    expression
name     DEQU   expression
name     CEQU   expression
```

Once a numeric equate has been defined with the EQU directive, it cannot be redefined within its scope (module or global). No storage is allocated for the symbol.

The assembler attempts to work out the context of the numeric equate by its usage (i.e. if it refers to a code or a data symbol) which can then be used by the sim/emulator for determining whether a code or data address is referenced. The programmer can explicitly define the context of a numeric equate with the CEQU (for code equates) and DEQU (for data equates) directives.

```
column      EQU   80                  ; numeric constant 80
row         EQU   25                  ; numeric constant 25
screenful   EQU   column * row        ; numeric constant 2000
line        EQU   row                 ; same as "row", numeric constant 25
$screensize EQU   screenful           ; screen size in bytes (global variable)

forward     DC    YES                 ; use before definition
YES         EQU   1

$timer_reg  DEQU  H'0F01              ; data space equate
$io_start   CEQU  H'0010              ; code space equate
```

Figure 2.7.2-1 Numeric Equates

String Equates

String equates are used to assign a text string constant to an assembler variable. String equates can be used in a variety of contexts, including defining aliases and string constants.

```
name     EQU     < string >
```

The assembler replaces each occurrence of name with string, wherever it occurs within scope, whether before or after the definition. If the string matches the name of a variable, it creates an alias for the variable.

Like numeric equates, the value of the variable cannot be redefined within its scope (module or global).

```
pi          EQU   <3.1415>              ; string constant for "3.1415"
prompt      EQU   <'Type Name : '>      ; string const "'Type Name : '"
arg1        EQU   <(0, SP)>             ; string constant for "(0, SP)"
arg2        EQU   <pi>                  ; alias of "pi", arg2 replaced with 3.1415

            .CONST
message     DC    prompt                ; expands to DC 'Type Name: '

            .CODE
            ld    a1,@arg1              ; expands to ld a1,@(0, SP)
```

Figure 2.7.2-2 String Equates

Redefinable Equates

Redefinable equates assign a value to an assembler variable, that may be redefined at a later time. Variables automatically record the type (integer or fractional) of the value they hold. The variable can be redefined at any point during assembly.

```
name     = expression
```

Unlike numeric and string equates, variables defined using redefinable equates can only be used in statements following their definition.

```
errors      =     errors + io_errors   ; keep track of the number of errors
                                       ; in error table.

            IF errors GT error_table_size
               .ERR Too many errors    ; stop assembly if too many errors
            ENDIF

$NUM_FILES  =    $NUM_FILES + 1        ; global number of files variable
```

Figure 2.7.2-3 Redefinable Equates

2.8 Numeric Data Types

The Macro Assembler supports Integer, Fractional, and Floating Point arithmetic.

2.8.1 Floating Point Arithmetic

Floating point values can be freely used in expressions and assembler variables. There is, however, no XAP instruction set support for floating point arithmetic.

Floating point arithmetic can be handled efficiently by restricting the number range to $-1.0 \leq x < 1.0$, and using the fixed point fractional arithmetic support of the XAP. Floating point numbers in the range $-1.0 \leq x < 1.0$ are automatically converted to fractionals when assigned to memory (see Section 2.8.2).

Numbers outside the range $-1.0 \leq x < 1.0$ are converted to integers when assigned to memory, with a range violation warning. They can be explicitly converted to the nearest whole integer using the INT operator (see Section 2.10.7).

Full floating point arithmetic can be implemented by writing specialist ‘Math Libraries’. This will lead to a considerable increase in code size and design complexity.

2.8.2 Fractional Arithmetic

A fractional is a fixed point (two’s complement number) expressing values in the range $-1.0 \leq x < 1.0$. Floating point numbers in the range $-1.0 \leq x < 1.0$ are automatically converted to fractionals when they are assigned to memory. Numbers outside this range are converted to integers and a range violation warning issued (unless explicitly assigned with the INT operator).

Fractionals have the following bit representation:

Most Significant Bit				Least Significant Bit
-1	2^{-1}	2^{-2}	2^{-3}	…

0.5	=	H'40000000	(32-bit)	H'4000 (16-bit)
-1.0	=	H'80000000	(32-bit)	H'8000 (16-bit)
-0.5	=	H'C0000000	(32-bit)	H'C000 (16-bit)
-0.183	=	H'E872B021	(32-bit)	H'E873 (16-bit)
0.206	=	H'1A5E353F	(32-bit)	H'1A5E (16-bit)

By comparison, integers have the following bit representation:

Most Significant Bit				Least Significant Bit
-2^{31} or -2^{15}	…	2^2	2^1	2^0

The XAP2's instruction set has been designed to support both fractional and integer arithmetic. Addition and Subtraction operations work on signed or unsigned, integer or fractional values. Multiplication and Division operations differ slightly between fractional and integer (see the MULT and DIV instructions in Section 4).

2.8.3 Long Integers and Long Fractionals

Both integer and fractional arithmetic can be extended to 32-bits using the carry / borrow flag. The Macro Assembler provides the operators HWRD and LWRD, to select the high and low words of 32-bit expressions, allowing easy assignment of 32-bit constants.

2.8.4 Allowed Number Ranges

Assembler variables and expressions are evaluated and stored as 32-bits numbers. Numbers assigned to memory locations are stored in 16-bits, and numbers used as instruction operands are stored in multiple chunks of 8-bits (see Section 2.3.2).

Error checking is performed to ensure that the number remains within range.

- *Expressions and Assembler Variables (signed 32-bit)*

Integer range:	$-2147483648 \leq x \leq 2147483647$
Floating Point range:	$\pm 10^{-38}$ to 10^{+37}

- *Addresses/Constants assigned to Memory (16-bit)*

Integer range (signed/unsigned):	$-32768 \leq x \leq 65536$
Fractional range (signed):	$-1.0 \leq x < 1.0$

- *Operands (8, 16, 24-bit) - see Section 2.3.2*

Data Addresses:	H’0 .. H’FFFF
Code Addresses (Small memory model):	H’0 .. H’00FFFF
Code Addresses (Large memory model):	H’0 .. H’FFFFFF

2.9 Defining and Initializing Data

The DC (Define Constant) directive is used to declare ROM space. The DS (Define Storage) directive is used to reserve RAM space. RAM space cannot be initialised.

[*label_name*]DC *expression*	(literal values)
[*label_name*]DS *expression*	(number of words to reserve)

```
                .CONST
v1              DC    'HELLO',0         ; null terminated text string - 6 words
v2              DC    "HELLO"           ; null terminated text string - 6 words
v3              DC    P'Hello\0'        ; packed null terminated string - 3 words
v4              DC    'A','B'           ; ASCII constants
v5              DC    H'10A0            ; hex
v6              DC    D'35              ; decimal - default
v7              DC    B'01100101        ; binary
v8              DC    0.317             ; floating point

                .VAR
v9              DS    (3+5) MOD 3       ; reserve 2 words
```

Figure 2.8.4-1 Defining Data

2.9.1 String Formats

The assembler accepts the following string formats:

Single Quotes	'Hello'	; standard, single character per word.
Double Quotes	"Hello"	; null terminated.
Packed	P'Hello'	; two characters per word.

Within a string special characters can be inserted using the Back-Slash escape character.

\n	Newline
\r	Carriage Return
\0	Null
\\	Back-Slash
\'	Single Quote
\"	Double Quote
\x*XX*	Hex Code *XX*

2.9.2 Number Formats

Numbers can be entered as Hex, Decimal, Binary or Floating Point as follows:

D’23	Decimal
23	Decimal (default)
H’12F3	Hex
B’1001	Binary
.423	Floating Point / Fractional
‘a’	ASCII

Numbers can also be entered as the result of an expression.

23 E -2	Floating point
6 MOD 2	Decimal

2.9.3 Segment Usage

Although the DS and DC directives can be used in all segments their use should be restricted to the following:

Segment	DC	DS
CODE	Insert an 16 bit instruction	Skip a range of addresses
DATA	Define a table (fixed array)	Define IO space or array
VAR	Not used	Define a variable
CONST	Define a constant / string	Not used
User Defined	No Restriction	No Restriction

Box 2.9.3-1 Segment Usage

2.9.4 Forward References

Although the XAP2 Macro Assembler is a multi-pass assembler, structures, macros and conditional assembly directives are expanded during pass 1 and must therefore not use any forward reference variables. Additionally, any expressions that can not be resolved at then end of the assembly process will generate an error message.

```
            .CONST
length      DC    end - hello          ; double forward referencing
table0      DC    start_pt             ; ok as start_pt has a size of 1
table1      DC    fred

hello       EQU   start_pt
fred        EQU   <'HELLO'>            ; gives error as fred assumed to be size 1
                                       ; as used in a DC forward reference

            .CODE
start_pt:   ld    x,#4
            IF (start_pt EQ hello)     ; illegal as hello not defined on pass 1
               ld    y,#6
            ENDIF
            IF (start_pt LT $)         ; ok as both start_pt and $ are known
               ld    al,#7
            ENDIF
end:        nop
```

Figure 2.9.4-1 Forward Reference Example

Symbols forward referenced with the DC directive are assumed to be of size one. If the symbol is subsequently defined as a type with a size greater than one, an error message will be produced.

Equates embedded within a structure (see Section 2.11.3) cannot contain forward references as structures are processed during pass 1 of the assembly process.

2.10 Operands, Operators and Expressions

2.10.1 Operands

Operands are the arguments of instructions, directives or expressions. An operand can take the form of a constant, a label, an assembler variable, a structure, a structure type, a register or an expressions.

The number, type and position of the operand is determined by the type of directive, instruction or operator.

2.10.2 Operators

The assembler provides a variety of operators for combining, comparing, or changing the values of operands in an expression.

- Arithmetic Operators (integer and floating point)
- Bitwise Operators (integer)
- Relation Operators (integer and floating point)
- Logic Operators (integer)
- Assignment Operators

Mixed Expressions

If an Arithmetic or Relation (integer and floating point) operator is presented with one integer and one floating point operand, the result will be a floating point number.

```
pattern = 3 * 1.2                    ; pattern = 3.6
```

If a Bitwise or Logic (integer only) operator is presented with a floating point operand in the range $-1.0 \leq x < 1.0$, the operator is applied to the fractional bit representation of the floating point number (see Section 2.8.2)

```
pattern = 0.206 AND H'FF00FFF0       ; pattern = H'1A003530
```

If a Bitwise or Logic (integer only) operator is presented with a floating point operand outside the range $-1.0 \leq x < 1.0$, the floating point number is converted to an integer and a range violation warning generated.

```
pattern = 500.2 AND H'FF00FFF0       ; pattern = H'000001F0
```

2.10.3 Arithmetic Operators

These operators perform arithmetic operations on the operands are shown in Box 2.10.3-1, with some example code in Figure 2.10.3-1.

Operator	Syntax	Meaning
+	+ *exp*	Positive (unary)
-	- *exp*	Negative (unary)
*	*exp1* * *exp2*	Multiplication
/	*exp1* / *exp2*	Division
MOD	*exp1* MOD *exp2*	Remainder (integer)
ABS	ABS *exp*	Absolute value
+	*exp1* + *exp2*	Addition
-	*exp1* - *exp2*	Subtraction
FEXP	FEXP *exp*	Extract Exponent (dec)
FMAN	FMAN *exp*	Extract Mantissa (dec)
E	*exp1* E *exp2*	*exp1* x 10^*exp2*
FREXP	FREXP *exp*	Extract Exponent (bin)
FRMAN	FRMAN *exp*	Extract Mantissa (bin)
LDEXP	*exp1* LDEXP *exp2*	*exp1* x 2^*exp2*

Box 2.10.3-1 Arithmetic Operators

```
intgr           =     14 * 3            ; 14 * 3 = 42
intgr           =     intgr / 4         ; 42 / 4 = 10
intgr           =     intgr MOD 4       ; 10 mod 4 = 2
intgr           =     intgr + 4         ; 2 + 4 = 6
intgr           =     intgr - 3         ; 6 - 3 = 3
intgr           =     -intgr - 8        ; -3 - 8 = -11
intgr           =     -intgr - intgr    ; 11 - -11 = 22

fm              =     FMAN 23.4         ; fm = .234
fe              =     FEXP 23.4         ; fe = 2
f               =     fm E fe           ; f = 23.4

frm             =     FRMAN 23.4        ; frm = 0.73125
fre             =     FREXP 23.4        ; fre = 5
fr              =     frm LDEXP fre     ; fr = 23.4


                ORG   H'100
a               DS    1
b               DS    1
mem1            EQU   a + 5             ; mem1 = H'105
mem2            EQU   a - 5             ; mem2 = H'FB
```

Figure 2.10.3-1 Arithmetic Operator Example

2.10.4 Bitwise Operators

These operators perform bit manipulation operations on the operands.

Operator	Syntax	Meaning	
SHL	exp SHL count	Arithmetic Shift Left	(exp1 x 2^count)
SHR	exp SHR count	Arithmetic Shift Right	(exp1 / 2^count)
NOT	NOT exp	Bitwise complement	
AND	exp1 AND exp2	Bitwise AND	
OR	exp1 OR exp2	Bitwise inclusive OR	
XOR	exp1 XOR exp2	Bitwise exclusive OR	

Box 2.10.4-1 Bitwise Operators

```
         ORG   H'F100
val      EQU   H'201
a        DC    val AND 255             ; a = H'F100, @a = H'01
b        DS    1                       ; b = H'F101, @b = H'01

mem2     EQU   a - 5                   ; mem2 = H'0000F0FB
c        DC    (mem2 SHR 4) OR H'FF00  ; c = H'F102, @c = H'FF0F
```

Figure 2.10.4-1 Bitwise Operator Example

2.10.5 Relation Operators

These operators perform relation operations on the operands. They would normally be used with conditional assembly directives.

Operator	Syntax	Meaning
LT	exp1 LT exp2	Less Than
GT	exp1 GT exp2	Greater Than
LE	exp1 LE exp2	Less Than or Equal To
GE	exp1 GE exp2	Greater Than or Equal To
EQ	exp1 EQ exp2	Equal To
NE	exp1 NE exp2	Not Equal To

Box 2.10.5-1 Relation Operators

All operators return -1 for TRUE and 0 for FALSE.

2.10.6 Logic Operators

These operators perform logic operations in the operands.

Operator	Syntax	Meaning
LOGIC	LOGIC exp	Convert non-zero to -1
NOT	NOT exp	Logical complement
AND	exp1 AND exp2	Logical AND
OR	exp1 OR exp2	Logical inclusive OR

Box 2.10.6-1 Logic Operators

TRUE is equivalent to -1 and FALSE is equivalent to 0. All operators are effectively bitwise. Non-zero values need to be converted to -1 with the LOGIC operator, before use.

```
         ORG    H'F100
a        DS     1                       ; a = H'F100
b        DS     1                       ; b = H'F101
A        EQU    H'201                   ; A = H'201
B        EQU    (a SHR 4) OR H'FF10     ; B = H'FF10

mem2     EQU    ((A GT B) AND  a) OR ((A LT B) AND b)        ; mem2 = H'F101
```

Figure 2.10.6-1 Logic Operator Example

2.10.7 Assignment Operators

The assembler supports the following assignment operators:

- Word (HWRD, LWRD)
- Integer (INT)
- Fractional (FRBIT)
- String (PASC)

Word Assignment

Syntax: HWRD *exp* Assign High Word of *exp*
LWRD *exp* Assign Low Word of *exp*

Usage: Fractionals and Integers are automatically rounded to 16-bits when they are assigned to a memory location. The HWRD and LWRD operators assign the high and low order words of the 32-bit internal representation, without rounding. These operators act on the bit representations of the number. It is important to take into consideration how the two different number types (fractional and integer) are represented internally, as these representations are very different. See Section 2.8.3

The HWRD operator can also be used to extract the top 8 bits of an address in the large memory model for assignment to the xh register.

Integer Assignment

Syntax: INT exp Assign the Integer part of the floating point number exp.

Usage: The INT operator views the operand as a floating point number. It then rounds it up (or down) to the nearest whole number, and assigns it as an integer.

Fractional Assignment

Syntax: FRBIT exp Cast the fractional bit pattern of *exp* to an Integer type.

Usage: The FRBIT operator views the operand as a fraction. It then casts the fraction to an integer, maintaining the same bit pattern. Floating point numbers outside the fractional range are rounded to the nearest whole value (as in the INT operator), with a warning of the range violation.

String Assignment

Syntax: PASC string Convert '\0' to space in string.

Usage: The PASC operator is often used on unpacked strings, in combination with the %OUT directive, to generate output during the assembly process (see Section 2.15.1).

```
        MODULE   assignment_example
        .ALL
        .CONST
s1      DC       'Hello'           ; s1 = '\0H\0e\0l\0l\0o'
s2      DC       PASC('Hello')     ; s2 = ' H e l l o'
s3      DC       P"Hello"          ; s3 = 'Hello\0'

v1      EQU      H'12345678        ; v1 = H'12345678
v2      EQU      HWRD(v1)          ; v2 = H'1234
v3      EQU      LWRD(v1)          ; v3 = H'5678
v4      EQU      INT(34.23)        ; v4 = D'34
v5      EQU      INT(34.67)        ; v5 = D'35
v6      EQU      FRBIT(0.5)        ; v6 = H'40000000
        ENDMOD   assignment_example
```

Figure 2.10.7-1 Assignment Example Code

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx  (xx/xx/xx)               Fri Oct 08 14:10:20 1999  #
#                                                                              #
# Command Line : xap2asm assign -list                                          #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: assign.xap

    1               :                       MODULE     assignment_example
    2               :                       .ALL
    3               :                       .CONST
    4 000000 R 0048 : s1                    DC         'Hello'                        ; s1 = '\0H\0e\0l\0l\0o'
    . 000001 R 0065 :
    . 000002 R 006C :
    . 000003 R 006C :
    . 000004 R 006F :
    5 000005 R 2048 : s2                    DC         PASC('Hello')                  ; s2 = ' H e l l o'
    . 000006 R 2065 :
    . 000007 R 206C :
    . 000008 R 206C :
    . 000009 R 206F :
    6 00000A R 4865 : s3                    DC         P"Hello"                       ; s3 = 'Hello\0'
    . 00000B R 6C6C :
    . 00000C R 6F00 :
    7               :
    8               : v1                    EQU        H'12345678                     ; v1 = H'12345678
    9               : v2                    EQU        HWRD(v1)                       ; v2 = H'1234
   10               : v3                    EQU        LWRD(v1)                       ; v3 = H'5678
   11               : v4                    EQU        INT(34.23)                     ; v4 = D'34
   12               : v5                    EQU        INT(34.67)                     ; v5 = D'35
   13               : v6                    EQU        FRBIT(0.5)                     ; v6 = H'40000000
   14               :                       ENDMOD     assignment_example


SUMMARY
-------

MODULES : 1

assignment_example              : Module Not Used

CODE SYMBOLS : 0

DATA SYMBOLS : 3

assignment_example.s1           : Address = H'0000
assignment_example.s2           : Address = H'0005
assignment_example.s3           : Address = H'000A

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 9

$??XAP2                         : Value = H'00000001, D'1
$??SMALL                        : Value = H'00000001, D'1
$??LARGE                        : Value = H'00000000, D'0
assignment_example.v1           : Value = H'12345678, D'305419896
assignment_example.v2           : Value = H'00001234, D'4660
assignment_example.v3           : Value = H'00005678, D'22136
assignment_example.v4           : Value = H'00000022, D'34
assignment_example.v5           : Value = H'00000023, D'35
assignment_example.v6           : Value = H'40000000, D'1073741824

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

CODE                            : Segment Not Used
DATA                            : Segment Not Used
VAR                             : Segment Not Used
CONST                           : Segment Start Add = H'0000, Size = H'000D


ASSEMBLY
--------

NO ERRORS FOUND
WARNINGS : 1

Warning     : No Code generated
```

Figure 2.10.7-2 Assignment Example Listing

2.10.8 Operator Precedence

Expressions are evaluated according to the following rules:

- Operations of highest precedence are performed first.
- Operations of equal precedence are performed from left to right
- The order of evaluation can be over-ridden by using parentheses.
- Operations in parentheses are always performed before any adjacent operations.

The order of precedence is shown below, operators on the same line have equal precedence and are evaluated left to right.

Precedence	Operators
(Highest)	
10	LENGTH, (, {, < (text-lit), ! (char-lit), & (substitute)
9	INDEX
8	LWRD, HWRD,FREXP, FRMAN, LDEXP, FEXP, FMAN, E
7	+, - (unary)
6	*, /, MOD, SHL, SHR
5	+, - (binary), ABS
4	EQ, NE, LT, LE, GT, GE
3	LOGIC, NOT (bitwise / logical), PASC, INT, FRBIT
2	AND (bitwise / logical)
1	OR, XOR (bitwise / logical)
0	), }, > (text-lit)
(Lowest)	

Figure 2.10.8-1 Operator Precedence

Symbols, Constants and Strings have precedence 0 when evaluated as part of an expression.

2.10.9 Expressions

Expressions consist of several operands (constants, labels, assembler variables, or expressions) combined (using operators) to describe a value or memory location. They are often used to describe values that would be difficult or inconvenient to specify directly in the source code. Expressions are evaluated at assembly time.

2.11 Structures

2.11.1 Defining Structure Types

The STRUC and ENDS directives mark the beginning and end of a type definition for a structure. The names given to fields must be unique within the structure in which they are defined.

Structure types act as templates, allowing structures to be defined. They also allow offsets within the structure to be referred to.

```
name      STRUC
field_definitions
ENDS
```

Structure type names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ , ‘?’ or ‘$’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Are case insensitive.
- Conform to the following templates :-

  $?name - ‘C’ automatically generated global structure type. (reserved)
  ?name - ‘C’ automatically generated local structure type. (reserved)
  $name - global structure type.
  name - local structure type.

- Must not match a DIRECTIVE, INSTRUCTION or MACRO name (case insensitive comparison). Structure types share the same “Name Space” as MACROs.

Field definitions have the following format :

```
name      DS      expression
```

Field names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ or ‘?’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Conform to the following templates :-

  ?name - ‘C’ automatically generated local field name. (reserved)
  name - user generated field name.

- Are case sensitive.

Each structure definition has its own “Name Space”. Field declaration names must be unique within this “Name Space”.

A structure, structure type and structure field can all have the same name.

2.11.2 Defining Structures

Structures are defined as follows:

```
name    structure_type [< expression_list >]
```

The expressions contained in the optional expression list (enclosed by the < > operators) are assigned to the fields within the structure when the structure is defined. The list contains an entry for each field, with entries separated by commas.

Each field entry can be evaluated as a series of expressions by nesting it in another set of < > operators. Only one level of nesting is allowed.

Structure names have the same restrictions as label and assembler variable names (see Section 2.6.3) and share the same name space.

The continuation character ('\') can be used to split the expression_list up over a number of lines to aid readability.

Operators

LENGTH	- gives the length of a structure or structure field in words.
INDEX	- gives the offset of a field in words.

The LENGTH and INDEX operators are only valid on structure types.

```
fred            STRUC
jim             DS    2
fred            DS    1
                ENDS

                .CONST
fred            fred <<10,19>, 23>      ; creates structure fred of structure type
                                        ; fred

v1              =     fred.fred         ; address of field fred in structure fred.
v2              =     fred              ; address of structure fred
v3              =     INDEX(fred.fred)  ; offset of field fred in structure type
                                        ; fred
v4              =     LENGTH(fred.fred) ; size of field fred in structure type
                                        ; fred
v5              =     LENGTH(fred)      ; size of structure type fred
```

Figure 2.11.2-1 Structure Operators Example

As well as defining tables, structures provide an easy way of representing stacks and accessing variables stored on a stack.

2.11.3 Embedded Equates

In addition to field definitions, structures can also contain comments, and numeric/string equates (EQU and EQU <…>). Equates defined inside a structure have the same scope and visibility as equates defined outside a structure. However, there is one significant difference. As structures are processed during pass 1 of the assembly process, an equate defined within a structure cannot contain a forward reference, whereas equates defined outside structures can contain forward references.

```
T_Field         STRUC

;
; crop is either C_Maize, C_Corn or C_Barley
;

C_Maize         EQU   H'0001
C_Corn          EQU   H'0002
C_Barley        EQU   H'0003

crop            DS    1

;
; size can be either C_Small or C_Large
;

C_Small         EQU   H'0000
C_Large         EQU   H'0001

size            DS    1

                ENDS
```

Figure 2.11.3-1 Embedded Equates Example

```
                MODULE    time
                .ALL

time            STRUC
string          DS        20
day             DS        1
month           DS        1
year            DS        1
time            DS        2
                ENDS

date            STRUC
day             DS        1
month           DS        1
year            DS        1
                ENDS

fred_s          STRUC
var1            DS        1
var2            DS        1
                ENDS

                .CONST
time            time      <<P'Monday 18th June 1995', 0>,     \
                          18,                                 \
                          6,                                  \
                          1995,                               \
                          <2312, 54>>

yday            date      <9,4,1987>                    ; assign constant

                .CODE
                ld        al,@yday.day
                ld        x,#yday
                ld        ah,@(INDEX(date.year), x)

                ld        y,#time.time
                ld        x,#(time.time + 1)

                sub       sp,#(LENGTH fred_s)           ; allocate space on stack

                ld        al,@(INDEX(fred_s.var1), sp)  ; access var1
                add       sp,#(LENGTH fred_s)           ; return space

                ENDMOD    time
```

Figure 2.11.3-2 Struct Example Code

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx  (xx/xx/xx)               Fri Oct 08 14:35:06 1999  #
#                                                                              #
# Command Line : xap2asm struct -list                                          #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: struct.xap

    1                :                       MODULE     time
    2                :                       .ALL
    3                :
    4 S              : time                  STRUC
    5 S              : string                DS         20
    6 S              : day                   DS         1
    7 S              : month                 DS         1
    8 S              : year                  DS         1
    9 S              : time                  DS         2
   10 S              :                       ENDS
   11                :
   12 S              : date                  STRUC
   13 S              : day                   DS         1
   14 S              : month                 DS         1
   15 S              : year                  DS         1
   16 S              :                       ENDS
   17                :
   18 S              : fred_s                STRUC
   19 S              : var1                  DS         1
   20 S              : var2                  DS         1
   21 S              :                       ENDS
   22                :
   23                :                       .CONST
   24 000000 R 4D6F : time                   time       <<P'Monday 18th June 1995', 0>,     \
    .                                                    18,                                 \
    .                                                    6,                                  \
    .                                                    1995,                               \
    .                                                    <2312, 54>>
    . 000001 R 6E64 :
    . 000002 R 6179 :
    . 000003 R 2031 :
    . 000004 R 3874 :
    . 000005 R 6820 :
    . 000006 R 4A75 :
    . 000007 R 6E65 :
    . 000008 R 2031 :
    . 000009 R 3939 :
    . 00000A R 3500 :
    . 00000B R 0000 :
    . 000014 R 0012 :
    . 000015 R 0006 :
    . 000016 R 07CB :
    . 000017 R 0908 :
    . 000018 R 0036 :
   29                :
   30 000019 R 0009 : yday                   date       <9,4,1987>                      ; assign constant
    . 00001A R 0004 :
    . 00001B R 07C3 :
   31                :
   32                :                       .CODE
   33 000000 C 1915 :                        ld         al,@yday.day
   34 000001 C 1918 :                        ld         x,#yday
   35 000002 C 0212 :                        ld         ah,@(INDEX(date.year), x)
   36                :
   37 000003 C 171C :                        ld         y,#time.time
   38 000004 C 1818 :                        ld         x,#(time.time + 1)
   39                :
   40 000005 C 025C :                        sub        sp,#(LENGTH fred_s)             ; allocate space on stack
   41                :
   42 000006 C 0017 :                        ld         al,@(INDEX(fred_s.var1), sp)    ; access var1
   43 000007 C 023C :                        add        sp,#(LENGTH fred_s)             ; return space
   44                :
   45                :                       ENDMOD     time


SUMMARY
-------

MODULES : 1

time                            : Start Add = H'0000, Size = H'0008

CODE SYMBOLS : 0

DATA SYMBOLS : 0
```

Figure 2.11.3-3 Struct Example Listing

```
DATA STRUCTURE SYMBOLS : 10

time.time                       : Address = H'0000, Size = H'0019, D'25
time.time.string                : Address = H'0000, Size = H'0014, D'20
time.time.day                   : Address = H'0014, Size = H'0001, D'1
time.time.month                 : Address = H'0015, Size = H'0001, D'1
time.time.year                  : Address = H'0016, Size = H'0001, D'1
time.time.time                  : Address = H'0017, Size = H'0002, D'2
time.yday                       : Address = H'0019, Size = H'0003, D'3
time.yday.day                   : Address = H'0019, Size = H'0001, D'1
time.yday.month                 : Address = H'001A, Size = H'0001, D'1
time.yday.year                  : Address = H'001B, Size = H'0001, D'1

EQUATE SYMBOLS : 3

$??XAP2                         : Value = H'00000001, D'1
$??SMALL                        : Value = H'00000001, D'1
$??LARGE                        : Value = H'00000000, D'0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

CODE                            : Segment Start Add = H'0000, Size = H'0008
DATA                            : Segment Not Used
VAR                             : Segment Not Used
CONST                           : Segment Start Add = H'0000, Size = H'001C



ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

Figure 2.11.3-4 Struct Example Listing (continued)

2.12 Macros

Macros are a series of statements that are assigned to a symbol name (and optionally parameters) so that the symbol can be used in place of the statements.

Macros are processed at assembly time. They can simplify writing source code by allowing the user to substitute mnemonic names for simple inline functions and repetitive code. By changing a macro, a programmer can easily change the effect of common blocks of statements throughout the source code.

2.12.1 Using Macros

Macros enable you to assign a symbolic name to a block of source statements, and then to use that name in your source file to represent the statements. Parameters can also be defined to represent arguments passed to the macro.

Macro expansion is a text-processing function that occurs at assembly time. Each time the assembler encounters the text associated with a macro name, it replaces that text with the text of the statements in the macro definition. Similarly, the text parameter names are replaced with the text of the corresponding actual arguments. When used the text parameter name must be prefixed with the substitute character '&'.

Macros are global. A macro can be defined anywhere in the source code as long as the definition precedes the first source line that calls the macro. Macros and equates are often kept in a separate file and made available to the program through an INCLUDE directive at the start of the source code.

All macros are expanded on the first pass of the assembler. Macros are expanded on every occurrence of the macro name, so they can increase the length of the executable file if called repeatedly.

2.12.2 Defining Macros

The MACRO and ENDMAC directives are used to define macros. MACRO designates the beginning of the macro block and ENDMAC designates the end.

```
name        MACRO [parameter [,parameter] ...]
            statements
            ENDMAC
```

Macro names can be of any length and :

- Start with ‘A..Z’, ‘a..z’, ‘_’ or ‘?’.
- Contain only the following characters: ‘A..Z’, ‘a..z’, ‘0..9’, ‘_’ or ‘?’
- Can be used to replace INSTRUCTIONs (case insensitive).
- Are case insensitive.
- Must not match a DIRECTIVE, or structure type name (shares the same “Name Space” with structure types).

The name must be unique and can be used later in the source code to invoke the macro.

The parameters (sometimes called dummy parameters) are names that act as placeholders for values to be passed as arguments to the macro when it is called. Any number of parameters can be specified, but they must all fit on one line. If you give more than one parameter, you must separate them with commas, spaces, or tabs.

Any valid assembler statement may be placed within a macro, including statements that call or define other macros. Any number of statements can be used. The parameters can be used any number of times in the statement. Macros can be nested, redefined, or used recursively.

The assembler assembles the statements in a macro only if the macro is called, and only at the point in the source file from which it is called. The macro definition itself is never assembled.

A macro definition can include the LOCAL directive, which lets you define labels used only within a macro, or the EXITM directive, which allows you to exit from a macro before all the statements in the block are expanded.

2.12.3 Calling Macros

A macro call directs the assembler to copy the statements of the macro to the point of the call and to replace any parameters in the macro statements with the corresponding actual arguments.

```
name [argument [,argument] ... ]
```

The name must be the name of a macro defined earlier in the source file. The arguments can be any text. For example, symbols, constants, and registers are often given as arguments. Any number of arguments can be given, but they must all fit on one line. Multiple arguments must be separated by commas, spaces, or tabs. Commas can always be used as separators; spaces and tabs may cause ambiguity if the arguments are expressions.

If a macro call has more arguments than the macro has parameters, the extra arguments are ignored. If a call has fewer arguments than the macro has parameters, any remaining parameters are replaced with a null (empty) string.

You can use conditional statements to enable macros to check for null strings or other types of arguments. The macro can then take appropriate action to adjust to different kinds of arguments.

```
addup       MACRO ad1, ad2, ad3, result
            ld    ah,#0
            ld    al,&ad1
            add   al,&ad2
            addc  ah,#0
            add   al,&ad3
            addc  ah,#0                ; result of addition in ah and al
            st    ah,@&result          ; store the high word
            st    al,@(&result+1)      ; store the low word
            ENDMAC


            MODULE   macro_example
            .ALL

            .VAR
v1          DS    2
v2          DS    2

            .CODE
            addup    #1, #4, #H'231, v1
            addup    #B'101, #124, #-H'123, v2

            ENDMOD   macro_example
```

Figure 2.12.3-1 Macro Example Code

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx  (xx/xx/xx)               Fri Oct 08 15:39:01 1999  #
#                                                                              #
# Command Line : xap2asm macro -list                                           #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: macro.xap

    1 M             : addup               MACRO     ad1, ad2, ad3, result
    2 M             :                     ld        ah,#0
    3 M             :                     ld        al,&ad1
    4 M             :                     add       al,&ad2
    5 M             :                     addc      ah,#0
    6 M             :                     add       al,&ad3
    7 M             :                     addc      ah,#0                       ; result of addition in ah and al
    8 M             :                     st        ah,@&result                 ; store the high word
    9 M             :                     st        al,@(&result+1)             ; store the low word
   10 M             :                     ENDMAC
   11               :
   12               :
   13               :                     MODULE    macro_example
   14               :                     .ALL
   15               :
   16               :                     .VAR
   17 000000 D ???? : v1                  DS        2
   18 000002 D ???? : v2                  DS        2
   19               :
   20               :                     .CODE
   21               :                     addup     #1, #4, #H'231, v1
    . 000000 C 0010 :                     * ld      ah,#0
    . 000001 C 0114 :                     * ld      al,#1
    . 000002 C 0434 :                     * add     al,#4
    . 000003 C 0040 :                     * addc    ah,#0
    . 000004 C 0200 :                     * add     al,#H'231
    . 000005 C 3134 :
    . 000006 C 0040 :                     * addc    ah,#0                       ; result of addition in ah and al
    . 000007 C 0021 :                     * st      ah,@v1                      ; store the high word
    . 000008 C 0125 :                     * st      al,@(v1+1)                  ; store the low word
    .               :                     * ENDMAC
   22               :                     addup     #B'101, #124, #-H'123, v2
    . 000009 C 0010 :                     * ld      ah,#0
    . 00000A C 0514 :                     * ld      al,#B'101
    . 00000B C 7C34 :                     * add     al,#124
    . 00000C C 0040 :                     * addc    ah,#0
    . 00000D C FF00 :                     * add     al,#-H'123
    . 00000E C DD34 :
    . 00000F C 0040 :                     * addc    ah,#0                       ; result of addition in ah and al
    . 000010 C 0221 :                     * st      ah,@v2                      ; store the high word
    . 000011 C 0325 :                     * st      al,@(v2+1)                  ; store the low word
    .               :                     * ENDMAC
   23               :
   24               :                     ENDMOD    macro_example


SUMMARY
-------

MODULES : 1

macro_example                   : Start Add = H'0000, Size = H'0012

CODE SYMBOLS : 0

DATA SYMBOLS : 2

macro_example.v1                : Address = H'0000
macro_example.v2                : Address = H'0002

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 3

$??XAP2                         : Value = H'00000001, D'1
$??SMALL                        : Value = H'00000001, D'1
$??LARGE                        : Value = H'00000000, D'0

REDEFINABLE EQUATE SYMBOLS : 0

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

CODE                            : Segment Start Add = H'0000, Size = H'0012
DATA                            : Segment Not Used
VAR                             : Segment Start Add = H'0000, Size = H'0004
CONST                           : Segment Not Used



ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

Figure 2.12.3-2 Macro Example Listing

2.12.4 Using Local Macro Symbols

The LOCAL directive can be used within a macro to define symbols that are available only within the defined macro.

```
LOCAL localname [,localname] ...
```

The localname is a temporary symbol name that is to be replaced by a unique symbol name when the macro is expanded. At least one localname is required for each LOCAL directive. Once declared, localname can be used in any statement within the macro definition. The name must be prefixed with the substitute character '&'.

The assembler creates a new actual name for localname each time the macro is expanded. The actual name has the following form:

$??*number*

The number is a hexadecimal number in the range (hex) 0000 to (hex) FFFF. You should not give other symbols names in this format, since doing so may produce a symbol with multiple definitions. Non local labels may be used in a macro, but if the macro is used more than once, the same label will appear in both expansions, and the assembler will display an error message. To avoid this problem, use only local labels (or redefinable equates) in macros.

2.12.5 Exiting from a Macro

Normally the assembler processes all the statements in a macro definition and then continues with the next statement after the macro call. However, you can use the EXITM directive to tell the assembler to terminate macro expansion before all the statements in the macro have been assembled.

When the EXITM directive is encountered, the assembler exits the macro immediately. Any remaining statements in the macro are not processed. If EXITM is encountered in a nested macro, the assembler returns to expanding the outer block.

```
three_times    MACRO    first, second
               LOCAL    loop

               IF &second GT 20
                 nop
                 EXITM                ; exit if second GT 20
               ENDIF

               IF $count LT 3
                 ld     al,#0
                 ld     x,#&first     ; clear x, al
&loop:           add    al,#&second
                 add    x,#-1
                 bne    &loop         ; loop &first times
$count           =      $count + 1    ; increment $count
               ENDIF
               ENDMAC


               MODULE   macro_example_2
               .ALL

$count         =     0

start:         three_times 4, 2
               three_times 5, 4
               three_times 5, 24          ; no action, 24 GT 20
               three_times 6, 6
               three_times 7, 8           ; no action, $count EQ 3
               sleep

               ENDMOD   macro_example_2
```

Figure 2.12.5-1 Macro Example 2 Code

```
################################################################################
#                                                                              #
# XAP2 Assembler - Vx.xx  (xx/xx/xx)               Fri Oct 08 15:57:19 1999  #
#                                                                              #
# Command Line : xap2asm macro2 -list                                          #
#                                                                              #
#                                                                              #
#                                                                              #
################################################################################


File: macro2.xap

    1 M             : three_times          MACRO      first, second
    2 M             :                      LOCAL      loop
    3 M             :
    4 M             :                      IF         &second GT 20
    5 M             :                        nop
    6 M             :                        EXITM                             ; exit if second GT 20
    7 M             :                      ENDIF
    8 M             :
    9 M             :                      IF         $count LT 3
   10 M             :                        ld       al,#0
   11 M             :                        ld       x,#&first                 ; clear x, al
   12 M             : &loop:                 add      al,#&second
   13 M             :                        add      x,#-1
   14 M             :                        bne      &loop                     ; loop &first times
   15 M             : $count                 =        $count + 1                ; increment $count
   16 M             :                      ENDIF
   17 M             :                      ENDMAC
   18               :
   19               :
   20               :                      MODULE     macro_example_2
   21               :                      .ALL
   22               :
   23               : $count               =          0
   24               :
   25               : start:               three_times 4, 2
    . 000000 C 0014 :                      * ld       al,#0
    . 000001 C 0418 :                      * ld       x,#4                      ; clear x, al
    . 000002 C 0234 : $??0000:             * add      al,#2
    . 000003 C FF38 :                      * add      x,#-1
    . 000004 C FEF0 :                      * bne      $??0000                   ; loop &first times
    .               : $count               * =        $count + 1                ; increment $count
    .               :                      * ENDMAC
   26               :                      three_times 5, 4
    . 000005 C 0014 :                      * ld       al,#0
    . 000006 C 0518 :                      * ld       x,#5                      ; clear x, al
    . 000007 C 0434 : $??0001:             * add      al,#4
    . 000008 C FF38 :                      * add      x,#-1
    . 000009 C FEF0 :                      * bne      $??0001                   ; loop &first times
    .               : $count               * =        $count + 1                ; increment $count
    .               :                      * ENDMAC
   27               :                      three_times 5, 24                     ; no action, 24 GT 20
    . 00000A C 0000 :                      * nop
    .               :                      * EXITM                           ; exit if second GT 20
   28               :                      three_times 6, 6
    . 00000B C 0014 :                      * ld       al,#0
    . 00000C C 0618 :                      * ld       x,#6                      ; clear x, al
    . 00000D C 0634 : $??0003:             * add      al,#6
    . 00000E C FF38 :                      * add      x,#-1
    . 00000F C FEF0 :                      * bne      $??0003                   ; loop &first times
    .               : $count               * =        $count + 1                ; increment $count
    .               :                      * ENDMAC
   29               :                      three_times 7, 8                      ; no action, $count EQ 3
    .               :                      * ENDMAC
   30 000010 C 0008 :                      sleep
   31               :
   32               :                      ENDMOD     macro_example_2


SUMMARY
-------

MODULES : 1

macro_example_2                  : Start Add = H'0000, Size = H'0011

CODE SYMBOLS : 4

macro_example_2.start            : Address = H'0000
$??0000                          : Address = H'0002
$??0001                          : Address = H'0007
$??0003                          : Address = H'000D

DATA SYMBOLS : 0

DATA STRUCTURE SYMBOLS : 0

EQUATE SYMBOLS : 3

$??XAP2                          : Value = H'00000001, D'1
$??SMALL                         : Value = H'00000001, D'1
$??LARGE                         : Value = H'00000000, D'0

REDEFINABLE EQUATE SYMBOLS : 1

$count                           : Value = H'00000003, D'3

STRING EQUATE SYMBOLS : 0

SEGMENTS : 4

CODE                             : Segment Start Add = H'0000, Size = H'0011
DATA                             : Segment Not Used
VAR                              : Segment Not Used
CONST                            : Segment Not Used



ASSEMBLY
--------

NO ERRORS FOUND
NO WARNINGS
```

Figure 2.12.5-2 Macro Example 2 Listing

2.12.6 Macro Operators

Macros use the following special set of macro operators:

Operator	Definition	Use
&	Substitute operator	Macro definition
!	Literal-character operator	Macro definition
<>	Literal-text operator	Macro call
%	Expression operator	Macro call

Box 2.12.6-1 Macro Operators

When used in a macro definition or a macro call, these operators carry out special control operations.

Substitute Operator

Within a macro definition, the substitute operator (&) forces the assembler to replace a parameter with its corresponding actual argument value.

&parameter

The substitute operator can be used when a parameter immediately precedes or follows other characters, or even if the parameter appears in a quoted string

```
errgen          MACRO    num, msg
                .CONST
$err&num!_msg   DC       'Error &num: &msg'
                ENDMAC

                .CODE
                errgen   5,<Unreadable disk>
```

Figure 2.12.6-1 Macro Substitute Operator Example

the macro is expanded to

```
$err5_msg      DC      'Error 5: Unreadable disk'
```

For complex nested macros you can use extra ampersands to delay the replacement of a parameter. In general you need to supply as many ampersands as there are nested levels.

Literal-Character Operator

Within a macro definition, the literal-character operator (!) forces the assembler to treat a specific character literally rather than as a symbol

`!`*character*

The literal-character operator is used with special characters such as the semicolon or ampersand when the meaning of the special character must be suppressed. It can also be used to explicitly mark the end of a parameter name in a (&) substitution.

Literal-Text Operator

When calling a macro, the literal-text operator (<>) directs the assembler to treat a list as a single string rather that as separate arguments.

```
<text>
```

The text is considered a single literal element even if it contains commas, spaces, or tabs. The literal-text operator is most often used in macro calls to ensure that values in a parameter list are treated as single elements.

The literal-text operator can also be used to force the assembler to treat special characters, such as the semicolon or the ampersand, literally.

The assembler resolves one set of angle brackets each time the parameter is used in a macro. When using nested macros, you will need to supply as many sets of angled brackets as there are levels of nesting.

```
work  1,2,3,4,5          ; pass five parameters to the macro work
work  <1,2,3,4,5>        ; pass a single parameter to the macro work
```

Expression Operator

When calling a macro, the expression operator (%) causes the assembler to treat the argument following the operator as an expression.

```
%text
```

The assembler computes the expression's value and replaces text with the result. The expression can either be a numeric expression or a text equate. If there are additional arguments after an argument that uses the expression operator, the additional arguments must be preceded by a comma, not a space or tab.

```
printe   MACRO exp, val
         %OUT  &exp = &val
         ENDMAC

         MODULE   expression_operator_example

sym1     EQU      100
sym2     EQU      200
msg      EQU      <'Hello World'>

         .CODE
         %OUT                                    ; dummy newline
         printe   <sym1 + sym2>, %(sym1 + sym2)
         printe   msg, %(PASC(msg))
         ENDMOD   expression_operator_example
```

Figure 2.12.6-2 Macro Expression Operator Example

will produce the following result on the display

```
sym1 + sym2 = 300
msg = H e l l o   W o r l d
```

2.12.7 Macro Recursion

Macro definitions can be recursive; that is, they can call themselves. Using recursive macros is one way of doing repeated operations. The macro does a task, and then calls itself to do the task again. The recursion is repeated until a specified condition is met.

2.12.8 Nesting Macro Definitions

One macro can define another. The assembler does not process nested definitions until the outer macro has been called. Therefore, nested macros cannot be called until the outer macro has been called at least once. Macro definitions can be nested to any depth. Nesting is limited only by the amount of memory available when the source files are assembled.

2.12.9 Nesting Macro Calls

Macro definitions can contain calls to other macros. Nested macros are expanded like any other macro call, but only when the outer macro is called.

2.12.10 Redefining Macros

Macros can be redefined. The new definition automatically replaces the old definition. If you redefine a macro from within the macro itself, make sure there are no statements or comments between the ENDMAC directive of the nested redefinition and the ENDMAC directive of the original macro.

2.13 Conditional Assembly

2.13.1 Conditional Directives

Conditional assembly directives test for a specified condition and assemble a block of statements if the condition is true. The directives are:

IF	IFDIF	IFNB	ELSE
IFB	IFE	IFNDEF	.ERR
IFDEF	IFIDN	ENDIF	

The statements following the IF directive can be any valid statements, including other conditional blocks. The ELSE directive and its statements are optional. ENDIF ends the block.

```
IF condition
   statements
[ELSE
   statements]
ENDIF
```

The statements in the conditional block are assembled only if the condition specified by the corresponding IF statement is satisfied. If the conditional block contains an ELSE directive, only the statements up to the ELSE directive are assembled. The statements that follow the ELSE directive are assembled only if the condition is not met. An ENDIF directive must mark the end of any conditional-assembly block. No more than one ELSE directive is allowed for each IF statement.

IF statements can be nested up to 255 levels. A nested ELSE directive always belongs to the nearest preceding IF statement that does not have its own ELSE.

The IF and IFE directives test the value of an expression and grant assembly based on the result.

```
IF expression
IFE expression
```

The IF directive grants assembly if the value of expression is true (non-zero). The IFE directive grants assembly if the value of expression is false (0). The expression must resolve to a constant value and must not contain forward references.

The IFDEF and IFNDEF directives test whether or not a symbol has been defined and grant assembly based on the result.

```
IFDEF name
IFNDEF name
```

The IFDEF directive grants assembly only if name is a defined label, variable or symbol. The IFNDEF directive grants assembly if name has not yet been defined. The name can be any valid name. If the name is a forward reference, it is considered undefined on pass 1.

The IFB and IFNB directives test to see if a specified argument was passed to a macro and grant assembly based on the result.

```
IFB <argument>
IFNB <argument>
```

These directives are always used inside macros, and they always test whether a real argument was passed for a specified dummy argument. The IFB directive grants assembly if argument is blank. The IFNB directive grants assembly if argument is not blank. The arguments can be any name, number or expression. Angle brackets (<>) are required.

The IFIDN and IFDIF directives compare two macro arguments and grants assembly based on the result.

```
IFIDN[I] <argument1>, <argument2>
IFDIF[I] <argument1>, <argument2>
```

These directives are always used inside macros, and they test whether real arguments passed for two specified arguments are the same. The IFIDN directive grants assembly if argument1 and argument2 are identical. The IFDIF directive grants assembly if argument1 and argument2 are different. The arguments can be names, numbers, or expressions. They must be enclosed in angle brackets and separated by a comma.

The optional I at the end of the directive name specifies that the comparison should be case insensitive.

2.13.2 Forcing Errors

The .ERR directive forces an error. The directive can be placed within a conditional-assembly block to limit the errors to certain situations.

```
.ERR <text>
```

The text is displayed on the standard output device.

2.14 Interrupts

The XAP2 includes relatively simple hardware support for interrupts. From this simple hardware support in the XAP2 core the user is able to design the appropriate complexity for the application by the inclusion of hardware (Interrupt Controller) and software.

2.14.1 Hardware Functionality

- A single interrupt input signal to the processor core IRQ_IN. This signal is sampled (registered) in the core. When the registered version of the signal goes high and the XAP2 is in User mode, the next program address (return address) is loaded automatically into the IX register and the PC is set to h'000004.
- A User mode flag which indicates the mode (User or Interrupt) of the processor. Interrupts are only enabled in User mode. The flag can be written to at any time.
- Duplicates of the X, XH any Y registers called IX, IHX and IY. In Interrupt mode these registers are automatically accessed in place of the original X and Y registers.
- Instructions to store and load flags and the User X and Y registers (UX, UXH and UY). These instructions are all indexed to the Y register
- An RTI instruction to restore the PC and flags following an interrupt.

2.14.2 Sequence/Operation of a Simple Interrupt

1. The XAP2 is in User mode and IRQ_IN goes high.
2. PC+1 is stored in IXH and IX, the User mode flag is cleared and the PC jumps to address h'0000004.
3. Code then determines that an interrupt has occurred and its type. (Depending on the implementation there may be several sources of interrupt). All registers which will be affected by the interrupt initiated code must be saved on the stack pointed to by IY.
4. The code must clear down the external interrupt signal via suitable hardware. The IRQ_IN signal is not latched, this must be done in hardware and the interrupt acknowledged in the interrupt processing code.
5. At this stage the code can continue in Interrupt mode or return to User mode by setting the flag register bit. Thus the interrupt processing can be re-entrant.
6. When all the interrupt processing is complete the code should return the processor to Interrupt mode, restore the modified registers from the stack (in general all registers should be saved) and ensure the return addess is in IXH, IX. Ensure a copy of the flags (with User mode bits set) is in the memory location referenced to Y, ensure that IY points to the required scratchpad area and then execute the RTI instruction.

Multiple sources of hardware interrupt can be supported by a more complex interrupt controller block which latches and prioritises the interrupts and provides interrupt status information to the software.

2.14.3 Example Interrupt Code

```
        ORG 4
INTCONT EQU h'4000    ; Hardware address of interrupt controller
; A very basic interrupt routine, which operates with the default
; interrupt controller. See document c6086-S-003a for further details.
?interrupt:
        ; {IXH,IX} holds return address
        ; IY holds scratchpad offset for user mode (IY-1 to IY-32)
        ; (IY+1 to IY+6 used by interrupt routine)
        ; AL, AH must be restored if modified
        ST flags, @(1,Y)
        ST XH, @(2,Y)
        ST X, @(3,Y)
        ST AL, @(4,Y)
        ST AH, @(5,Y)
        ; First find out the cause of the interrupt
        LD AL, @(0,Y)
        AND AL, #h'0020      ; b'00100000 - ie I bit set
        BNE ?intcause
        ; If we were interrupted and I not set then it
        ; must be because we had a software exception.
        ; ie B or T must be set.
        LD AL, @(1,Y)
        AND AL, #h'00c0      ; b'11000000 - ie B/T bits set
        BNE ?exceptcause
        ; We should never get here, because there was
        ; no cause to interrupt
        brk
        bra ?intrestore
?intcause:
        ; Actual interrupt processing happens here.
        ; We can switch to user mode if we want.
        ; Or, we can call a function *but* if we remain
        ; in interrupt mode, it must be within the 64K boundary.
        ;  Cancel the cause of the hardware interrupt
        LD X, #INTCONT
        LD AL, #h'feed  ; Magic number
        ST AL, @(0,X)
```

```
?intOK:
        ; Increment a counter so we can see what we are doing
        LD AH, @(6,Y)
        ADD AH, #1
        ST AH, @(6,Y)
        bra ?intrestore
?exceptcause:
        nop
?intrestore:
        LD AL, @(1,Y)
        OR AL, #h'0010  ; Set usermode
        AND AL,#h'00df  ; Clr interrupt
        ST AL, @(1,Y)
        ; Restore reg's
        LD AH, @(5,Y)
        LD AL, @(4,Y)
        LD X, @(3,Y)
        LD XH, @(2,Y)
        RTI @(1,Y)
```

2.15 Miscellaneous

2.15.1 Generating Output

The %OUT directive instructs the assembler to display text on the standard output device.

```
%OUT text
```

The text can be any line of ASCII characters. If you want to display multiple lines, you must use a separate %OUT directive for each line.

The %OUT directive can be used within a macro to print the results of an expression evaluation. The PASC (Print ASCII) operator is used to convert the null characters of an unpacked string into spaces.

```
        MODULE

        %OUT
        %OUT    Outputs fixed text only

        ; Defining %OUT in a macro in conjunction with %
        ; allows variable information to be output
printe  MACRO   text, val
        %OUT    &text &val
        ENDMAC

sum     EQU     2+3
        printe  <sum is>, %sum

        ; Care needs to be taken with strings as the
        ; assembler evaluates them as packed and stops at
        ; the first null
        printe  <Packed Strings are OK:>, %P'Packed String'
        printe  <Unpacked Strings disappear:>, %'Unpacked String'
        printe  <Use PASC to show an>, %PASC('Unpacked String')

        ENDMOD
```

Figure 2.15.1-1 %OUT Example

will produce the following result on the display

```
Outputs fixed text only
sum is 5
Packed Strings are OK: Packed String
Unpacked Strings disappear:
Use PASC to show an  U n p a c k e d   S t r i n g
```

2.15.2 INCLUDE files

The INCLUDE directive inserts source code from a specified file into the source file from which the directive is given.

```
INCLUDE    filespec
```

The filespec must specify an existing file containing valid assembler statements. When the assembler encounters an INCLUDE directive, it opens the specified source file and begins processing its statements. When all statements have been read, the assembler continues with the statements immediately following the INCLUDE directive.

The filespec can be given either as a file name, or as a complete or relative file specification. For relative file paths, the current directory is that of the file doing the including.

An include file is treated like any other source file. It can be used to define macros, structures and equates or contain standard code. An include file can also include other include files, allowing nesting of source files to any level.

The FILE directive is used to record the name of the current source file. This directive is used internally by the assembler to mark the end of included files, and is not normally visible to the user.

```
FILE    filespec
```

2.15.3 Original Line Number Propagation

Line numbers can be propagated from high-level languages by including them as unique symbols within the source code. The assembler will assign a code address to the symbol allowing the symbolic debugger to equate an address with an original line number. Symbols must be of the following format:

^*module*.'*file_name*'.?*Orig_line_number.occurrence*:

The occurrence field allows different addresses to be mapped to the same source line number. This allows files to be included more than once within a program.

2.15.4 The Continuation Character

As an aid to readability, expressions can be split over several lines by placing a continuation character '\' at the end of each partial line. The continuation character must be preceded by a space and can not have any characters following it.

The continuation character can not be used in the middle of Symbols, Instructions or Directives, and is ignored in Strings and Comments.

```
        MODULE first

a_very_long_name_for_a_macro            \
        MACRO first
        nop
        ld    y, #&first
        ENDMAC

        .CONST
a_very_long_name_for_a_var                                              \
        DC    P'Error Message: \"number too big\"', 0,     \
              P'Error Message: \"name not valid\"', 0

        .CODE

start_address_of_a_long_line:
        ld    x, #30
        a_very_long_name_for_a_macro 3

        ENDMOD
```

Figure 2.15.4-1 Continuation Character Example

2.15.5 Maximum Line Length

The maximum line length any source file can have is 500 characters. This includes any comments, and continuation lines.

2.15.6 Forcing a Page Break in a Listing File

The .PAGE directive can be used to insert a 'form feed' character into the listing file.

2.15.7 Scratchpad RAM Access Relative to IY Register

The C compiler uses 16 locations at the top of data space (h'fffe0 to h'ffff) to store temporary variables. In order to allow fast task switching routines, this memory is mapped to a location in memory determined by the IY register. This allows different workspaces for each task. The RAM is mapped to the 16 byte block immediately below the IY address (i.e. IY-1 to IY-16). The memory is directly mapped when in interrupt mode.

2.15.8 Source File Parsing

The parser uses whitespace and line breaks to determine whether a token is a label or a command. The rule is that a label or the name of an assembler variable or equate always starts immediately after a line break, while a command is separated from a line break by one or more characters of whitespace.

This can lead to cryptic error messages from the assembler – if a command (such as an assembler instruction or a directive) starts immediately after a line break, it will not be interpreted correctly. Error messages such as “Invalid command, “”” are likely to be a result of this. Care should be taken that whitespace is inserted where appropriate to avoid errors of this nature.

2.16 Error Message List

When an error occurs the assembler records the type of error and position within the source file. Assembly continues until an error is detected that makes further assembly impossible, a summary of the errors is then displayed. The error summary is appended to the end of the listing file and an error log. Error messages are displayed in a format compatible with text editors to allow jumps to erroneous lines.

The following errors can occur:

Source File Errors

"No source files specified in command line",
"No output File Name given",
"Source Stack Overflow",
"Invalid File Name",
"Unable to open file",
"Unable to set the file position",
"Unable to get the file position",
"Unexpected End Of Working File",
"Can't delete FILE - FILE open or write protected"

Macro Errors

"Invalid MACRO Name",
"Invalid MACRO parameter name",
"Unknown MACRO",
"Unexpected End of MACRO definition",
"Not part of a MACRO definition",
"Parameter/Label name already defined",
"Parameter name not found",
"Too many MACRO parameters",
"Badly formed parameter",
"Substitute parameter not found",
"Not in MACRO expansion",
"Too many MACRO LOCAL labels",
"ENDMAC/EXITM not found"

Symbol Errors

"Invalid SYMBOL Name",
"Undeclared SYMBOL",
"Invalid SYMBOL type",
"Invalid SYMBOL format",
"Invalid STRUC expression",
"Invalid STRUC assignment",
"Invalid structure format",
"Invalid local symbol name",
"Invalid field size in STRUC declaration",
"Different scopes",
"Attempting to update a symbol with a different type",
"Attempting to update a symbol with a different value",
"Not a STRUC type SYMBOL",
"Not a STRUC Definition type SYMBOL",
"Not a STRUC Field type SYMBOL",
"STRUC definition has no length allocated",
"STRUC defined in the CODE Segment",
"STRUC assignment larger than field",
"STRUC too large",
"ENDS not found",
"Too many STRUC assignment parameters",
"Defined as both a MACRO and STRUC",
"Undeclared Global SYMBOL",
"No LABEL given",
"Label defined",
"Inappropriate label definition",
"Can not assign a valid Data/Address to the symbol",
“Symbol assumed to be of word size, it is used in a forward reference”,
"Symbol Redefined",

Directive Errors

"Invalid command",
"Invalid MODULE Name",
"MODULE not defined",
"MODULE already exists",
"Non-Matching MODULE name",
"ENDMOD not found",
"DEQU defined using a code symbol",
"CEQU defined using a data symbol"

Conditional Assembly Errors

"Forced Error : .ERR Instruction encountered",
"Not in Conditional Assembly",
"No associated IF statement for ELSE"

Segment Errors

"Code in the DATA segment",
"Unable to resolve the Code Address",
"$??CODE Address must always increment",
"location counter should increment",
"location counter not defined",
"Litvals are not needed, and therefore not supported on the XAP2",
"Invalid SEGMENT name",
"Badly formed expression",
"Invalid (C)PACK assignment",
"Attempting to (C)PACK Segment twice",
"Invalid SEGMENT",
"Packed Segments too large",
"Attempt to ORG within a Packed Segment",
"Attempt to pack a Code segment in data space",
"Attempt to pack a Data segment in code space",

System Errors

"Can't get memory",
"Store Full",
"Stack Overflow",
"Stack Underflow",
"Assembler Internal Error",
"Too Many Errors",
"No Storage Defined",

Source Errors

"Source line too long",
"Inappropriate expression definition",
"Expression expansion too long",
"Unknown Argument",
"Invalid Argument Format",
"No arguments in command line",
"Argument is missing quote ',\"",
"\"-o\" File name not found",
"\"-cmdfile\" File name not found",
"More than one \"-o\" argument",
"Nested \"-cmdfile\" argument"
"Invalid number of passes in command line arguments (2-20)",
"Invalid optimisation level in command line arguments (>0)",
"Invalid code size in command line arguments (1-24)",
"Invalid data size in command line arguments (1-24)",
"No Memory Model defined within Module",
"Mixed Memory Model defined within Code",
"Module not Large Memory Model Compliant",
"Both Assembler Memory Model flags are set",
"Generated by XAP C Compiler (not XAP2 compatible)",

Expression Errors

"Invalid expression",
"Lex Buffer Full",
"Unexpected token",
"Badly formed Text Literal - No '>'",
"Converting Float (not Frac) to Integer",
"More than one parameter was given",
"Operator requires two Floats",
"Operator requires two Integers",
"Operator requires an Integer",
"Operator requires two Integers or Floats",
"Operator requires a string",
"Operator requires an Integer or Float",
"Operator requires a Float",
"Can't resolve forward reference",
"Fraction outside range: -1.0 <= x < 1.0",
"Shift value out of range: 0 <= x < 32",
"Invalid Escape Sequence",
"Unmatched parenthesis in expression",
"Too many nested STRING EQUATEs"

Instruction Errors

"Data Address map conflict",
"Address not valid for code/data generation",
"Invalid REGISTER",
"Invalid ADDRESS/DATA operand",
"No ADDRESS or DATA operand given",
"No REGISTER specified",
"Can't resolve data",
"Branch to DATA Symbol",
"Branch to contents of CODE Symbol",
"Not an Integer Expression",
"Address overflow - Address greater than H'FFFF",
"Integer value out of range",
"Number generated is too large",
"Number out of range - greater than H'FFFF",
"Data Out of Range",
"Address Out of Range",
"CODE Symbol",
"Non-zero ROM Start Address",
"No Code generated",
"Invalid XAP2 instruction",
"Code Symbols occupy two words in the Large Memory Model”,

3 LINKER SUPPORT

The assembler has two special modes of operation which are used to allow the assembler to work with an external linker. Extensive use of these facilities is made by the ‘C’ environment and the program XAPCL.

The modes are selected by command line options ‘-c’ and ‘-s’. In these modes normal error checking and file generation is suppressed. Instead the assembler produces information in a “.lif” (linker information) file.

The ‘-c’ mode produces information on declared and undeclared global symbols.
The ‘-s’ mode produces segment type information.

xap2asm file1 file2 ... **-c**

The “-c” option forces the assembler to construct a list of all global symbols declared or referenced in the source code. Symbols can be labels, variables or structures (MACROs are not included). The list can be used by the linker to extract missing functions from libraries or identify undeclared variables / functions.

The linker information file has the following format:

.DECLARED
<Declared Global Symbol List>
.UNDECLARED
<Undeclared Global Symbol List>
.END

```
.DECLARED
$main
$testfunc
$retstat
$in
.UNDECLARED
$setup
$post
$blenter
$_sprintf
$blexit
$strncmp
$strcpy
.END
```

Figure 3-1 Global Symbols File Example

xap2asm file1 file2 **-s**

The "-s" option forces the assembler to construct a file containing the segment information. The list can be used by the linker to automatically define segment start addresses and sizes, making the best use of the available address space. This mode allows the linker to provide the same functionality as the .PACK directive.

The linker information file has the following format:

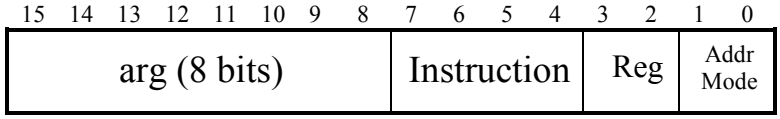
.SEGMENTS
<Segment Name> <Start Address> <Size>
.END

```
.SEGMENTS
$??C_RESERVED, 0000, 10000
$??XINITC, 0260, 0009
$??XCONST, 0F60, 1091
$??XINIT, 9000, 0009
$??XVAR, 9D00, 00CE
$??ZVAR, FE60, 0008
.END
```

Figure 3-2 Segment Size File Example

4 INSTRUCTION SET

General Instruction Format:



Data Addressing Modes

Assembler Syntax	Name	'Data' (Source / Destination)	Address Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Box 4-1 Data Instruction Format

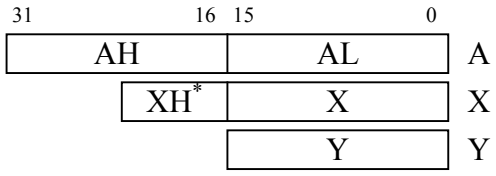
Branch Addressing Modes

Assembler Syntax	Name	'Branch Address' (new Program Counter value)	Address Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Box 4-2 Branch Instruction Format

General Registers

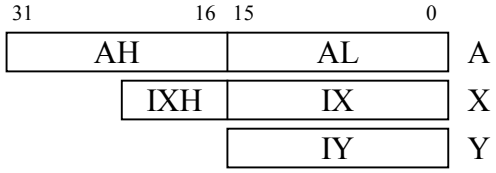
Register	Reg
AH	00
AL	01
X	10
Y	11



* Register XH is accessible through special instructions in the *large memory* model. In the *small memory* model XH has the value zero.

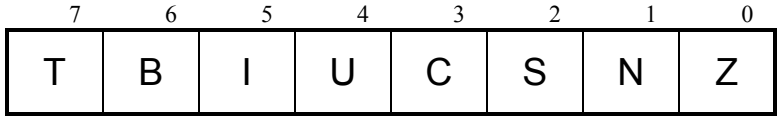
Interrupt Mode Registers

Register	Reg
AH	00
AL	01
IX	10
IY	11



User registers UX, UXH, UY, and the Flags register are accessible through special instructions.

Flags Register



T If set and h/w debug is disabled a single user mode instruction will be executed and then an interrupt occurs.
B If set and h/w debug is disabled an interrupt is generated on a Break Event.
I Sampled version of the IRQ_IN hardware signal.
U Set in User Mode - Cleared in Interrupt Mode.
C Set if a carry is generated.
S Set if true sign of result negative.
N Set if result negative.
Z Set if result zero.

Symbols

@	Contents of.
#	Immediate.
arg^{16}	16 bit sign extended argument.
arg^{24}	24 bit sign extended argument.
{XH, X}	24 bit address composed of {high 8 bits, low 16 bits}
~	NOT
&	AND
|	OR
^	XOR

Instruction Set Summary

Data Transfer

BC	Block Copy
LD	Load
ST	Store

Indexed Y Single Register Data Transfer

ST flags	Store
LD flags	Load
ST UX	Store
LD UX	Load
ST XH	Store
LD XH	Load
ST UY	Store
LD UY	Load

Arithmetic Operations

ADD	Add
ADDC	Add with Carry
CMP	Compare
DIV	*Superseded XAP Instruction* - use SDIV
MULT	*Superseded XAP Instruction* - use SMULT
NADD	Negated Add
SDIV	Signed Divide
SMULT	Signed Multiply
SUB	Subtract
SUBC	Subtract with Carry
TST	Test
UDIV	Unsigned Divide
UMULT	Unsigned Multiply

Logic Operations

AND	And
OR	Or
XOR	Exclusive Or

Shift Operations

ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
LSL	Logic Shift Left - same as ASL
LSR	Logic Shift Right
ROL	Rotate Left
ROR	Rotate Right
SAL	*Superseded XAP Instruction* – use ASL
SAR	*Superseded XAP Instruction* – use ASR
SCL	*Superseded d XAP Instruction* – use ROL
SCR	*Superseded XAP Instruction* – use ROR

Branch

BCC	Branch if Carry flag Clear
BCS	Branch if Carry flag Set
BEQ	Branch if Equal
BLT	Branch if Less Than
BMI	Branch if Minus
BNE	Branch if Not Equal
BPL	Branch if Plus
BRA	Branch Always
BRA2	Branch Always - 2 word fixed instruction
BRA3	Branch Always - 3 word fixed instruction
BRXL	Branch Always relative to X (low)
BSR	Branch to Sub-Routine
RTI	Return from Interrupt
RTS	Return from Sub-Routine

System Control

BRK	Break
NOP	No Operation
PRINT	Print
SIF	Serial Interface
SLEEP	Sleep

Operation: Register + Data → Register

Syntax: ADD <Register>, <Data Specifier>

Examples:
```
add   x, #H'3F
add   al, @(34,x)
```

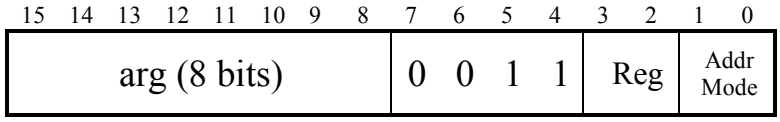
Description: Add the contents of the register and the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register + Data + Carry → Register

Syntax: ADDC <Register>, <Data Specifier>

Examples:
```
addc    y, #H'010A
addc    ah, @(23, y)
```

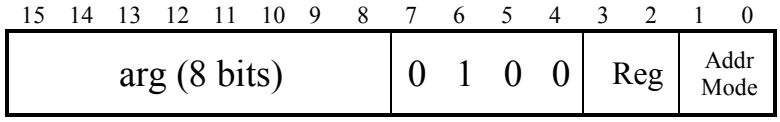
Description: Add the contents of the register, the carry flag and the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register & Data → Register

Syntax: AND <Register>, <Data Specifier>

Examples:
```
and   al, #H'0A
and   ah, @H'0102
```

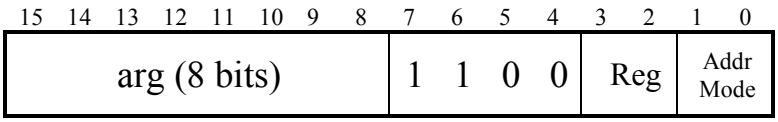
Description: AND the contents of the register with the effective address data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Shift A left by count, (C ← [AH:AL] ← 0)

Syntax: ASL <Count Specifier>

Examples:
```
asl   #3
asl   @shift_count
```

Description: Shift the A register (combined AH and AL) left by count. Feed 0 into the least-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

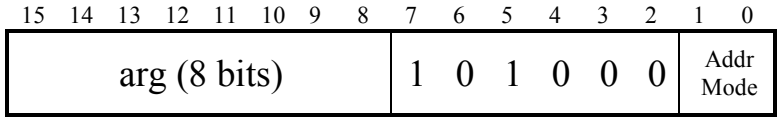
Count values greater than 32 will have the same effect as a count of 32.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Shift A right by count, (AH[15] → [AH:AL] → C)

Syntax: ASR <Count Specifier>

Examples:
```
asr   #3
asr   @shift_count
```

Description: Shift the A register (combined AH and AL) right by count. Feed AH[15] into the most-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

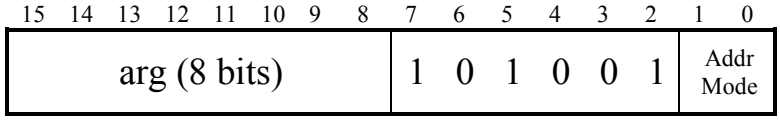
Count values greater than 32 will have the same effect as a count of 32.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: @(src_addr … src_addr + count) → @(dest_addr … dest_addr + count)

Syntax: BC

Examples:

```
ld      x,  #src_addr
ld      y,  #dest_addr
ld      al, #count
bc
```

Description: Copies a block of data from one part of the data address space to another. The source and destination addresses are held by the X and Y registers respectively - the count is held in the AL register. The data cannot be copied onto itself.

Notes: The X, Y, and AL registers are set according to the following pseudo code:

```
start:  ld      ah, @(0, x)
        st      ah, @(0, y)
        add     x, #1
        add     y, #1
        sub     al, #1
        bne     start
```

The AH register is not used by BC and its contents are unaltered.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	0	0	0	0	1	0	0	1

Operation: If (C = 0) then Branch Addr → PC

Syntax: BCC <Branch Address Specifier>

Examples:
```
bcc   $ + H'0A
bcc   loop
```

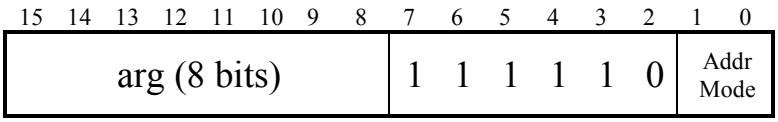
Description: If the Carry flag is Cleared, program execution continues from the branch address. If the Carry flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (C = 1) then Branch Addr → PC

Syntax: BCS <Branch Address Specifier>

Examples:
```
bcs   H'0A, x
bcs   loop
```

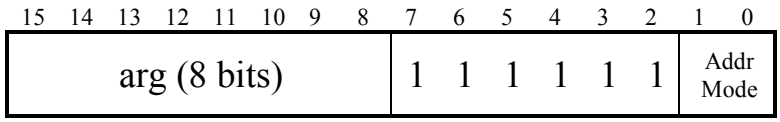
Description: If the Carry flag is Set, program execution continues from the branch address. If the Carry flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (Z = 1) then Branch Addr → PC

Syntax: BEQ <Branch Address Specifier>

Examples:
```
beq   @H'0A
beq   loop
```

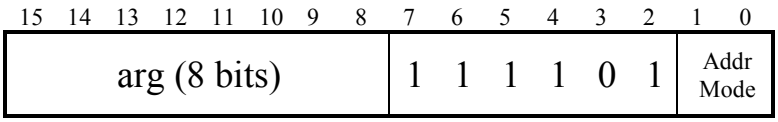
Description: If the Zero flag is Set, program execution continues from the branch address. If the Zero flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (S = 1) then Branch Addr → PC

Syntax: BLT <Branch Address Specifier>

Examples:
```
blt   @(H'0A, y)
blt   $ - 34
```

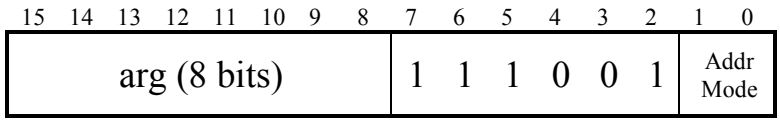
Description: If the Sign flag is Set, program execution continues from the branch address. If the Sign flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (N = 1) then Branch Addr → PC

Syntax: BMI <Branch Address Specifier>

Examples:
```
bmi   @(H'0A, y)
bmi   $ - 34
```

Description: If the Negative flag is Set, program execution continues from the branch address. If the Negative flag is Cleared then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4 3 2	1 0
arg (8 bits)	1 1 1 0 1 1	Addr Mode

<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (Z = 0) then Branch Addr → PC

Syntax: BNE <Branch Address Specifier>

Examples:
```
bne   @(H'0A, y)
bne   $ - 34
```

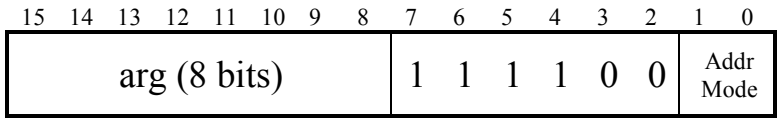
Description: If the Zero flag is Cleared, program execution continues from the branch address. If the Zero flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: If (N = 0) then Branch Addr → PC

Syntax: BPL <Branch Address Specifier>

Examples:
```
bpl   @(H'0A, y)
bpl   start
```

Description: If the Negative flag is Cleared, program execution continues from the branch address. If the Negative flag is Set then the next instruction is executed.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4 3 2	1 0
arg (8 bits)	1 1 1 0 1 0	Addr Mode

<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: Branch Addr → PC

Syntax: BRA <Branch Address Specifier>

Examples:
```
bra   H'03A, x
bra   start
```

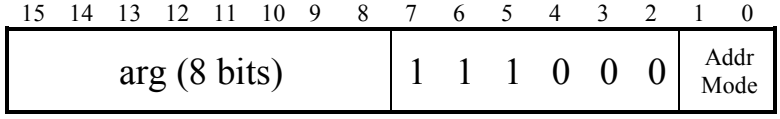
Description: Program execution continues from the branch address.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: Branch Addr → PC

Syntax: BRA <Branch Address Specifier>

Examples:
```
bra   H'03A, x
bra   start
```

Description: Program execution continues from the branch address. This instruction has a fixed size of two words.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
arg (8 bits)	0 0 0 0 0 0 0 0
arg (8 bits)	1 1 1 0 0 0 Addr Mode

<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: Branch Addr → PC

Syntax: BRA <Branch Address Specifier>

Examples:
```
bra   H'03A, x
bra   start
```

Description: Program execution continues from the branch address. This instruction has a fixed size of three words.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
arg (8 bits)	0 0 0 0 0 0 0 0
arg (8 bits)	0 0 0 0 0 0 0 0
arg (8 bits)	1 1 1 0 0 0 Addr Mode

<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: Stop for Debug

Syntax: BRK

Examples: brk

Description: Stops the processor if Break events are enabledThe program counter (PC) is not incremented. The BRK instruction is interpreted as a NOP ifBreak events are disabled.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

Operation: PC + 1 + X^{24}[15:0] → PC

Syntax: BRXL

Examples:
```
ld      x, #H'4
brxl
```

Description: Program execution continues from the branch address, which is given relative to X[15:0] sign extended to 24 bits.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	0	0	0	0	0	1	0	0	1

Operation: PC + 1 → {XH, X}, Branch Addr → PC

Syntax: BSR <Branch Address Specifier>

Examples:
```
bsr   H'03A, x
bsr   start
```

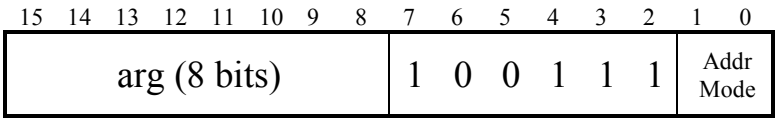
Description: The program counter (PC) + 1 is stored in the X and XH registers. Program execution then continues from the branch address.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Branch Address Specifier>

Syntax	Name	'branch addr'	Addr Mode
arg	PC Relative	PC + arg^{24}	00
@arg	Direct	{XH, Contents of address (arg^{16})}	01
arg, X	X Relative	{XH, X} + arg^{24}	10
@(arg, Y)	Indexed Y	{XH, Contents of address (arg^{16} + Y)}	11

Operation: Register - Data → Flags

Syntax: CMP <Register>, <Data Specifier>

Examples:
```
cmp   y, #H'010A
cmp   ah, @(23, y)
```

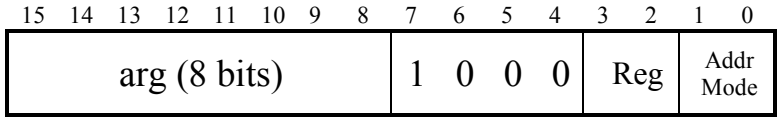
Description: Subtract the specified data from the contents of the register to set the flags. The result is not stored. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Data → Register

Syntax: LD <Register>, <Data Specifier>

Examples:
```
ld   x, #H'3F
ld   ah, @(23, y)
```

Description: Load the register with the specified data.

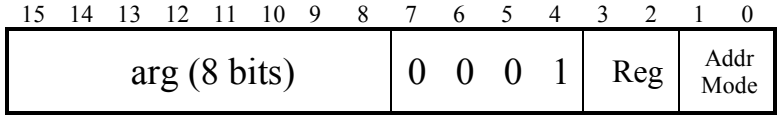
Notes: A number of special load instructions are available for accessing the flags register and interrupt mode registers. These instructions have a different format and are described on the next page.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if data negative. N = Data[15]
Z Set if data zero. Z = ~(Data[15] | ... | Data[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Notes: These special load instructions are accessible only in the Indexed Y address mode. No condition codes are affected.

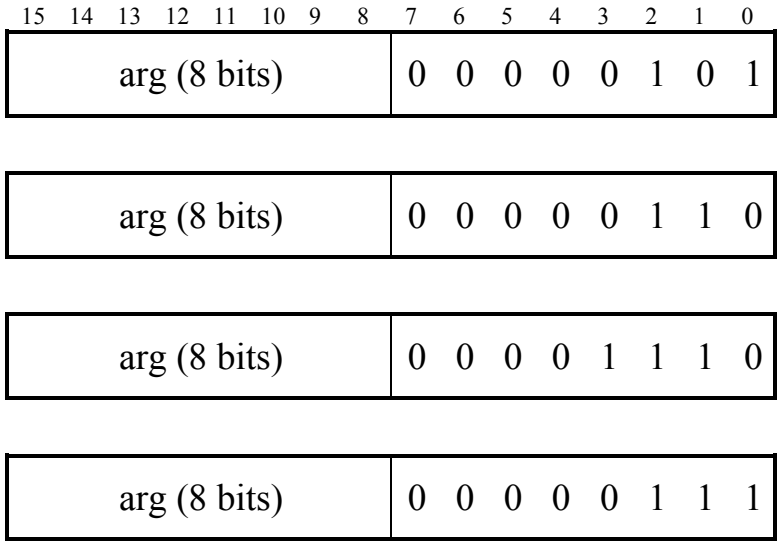
Syntax:

LD	flags, @(arg, Y)	@(arg^{16} + Y) → Flags[7:0]
LD	UX, @(arg, Y)	@(arg^{16} + Y) → UX[15:0]
LD	XH, @(arg, Y)	if User mode LSB @(arg^{16} + Y) → XH[7:0] else @(arg^{16} + Y) → {IXH[7:0], UXH[7:0]}
LD	UY, @(arg, Y)	@(arg^{16} + Y) → UY[15:0]

Examples:

```
ld   flags, @(-3, y)
ld   ux, @(-4, y)
```

Instruction Format:



Operation: Shift A left by count, (C ← [AH:AL] ← 0)

Syntax: LSL <Count Specifier>

Examples:
```
lsl   #3
lsl   @shift_count
```

Description: Identical to ASL. Shift the A register (combined AH and AL) left by count. Feed 0 into the least-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

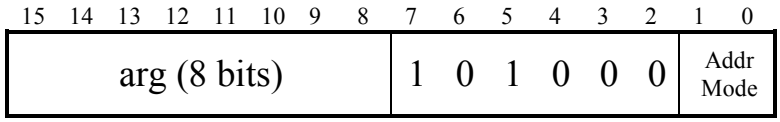
The count values greater than 32 will have the same effect as a count of 32.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Shift A right by count, (0 → [AH:AL] → C)

Syntax: LSR <Count Specifier>

Examples:
```
lsr   #3
lsr   @shift_count
```

Description: Shift the A register (combined AH and AL) right by count. Feed zero into the most-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

The base instruction is two words long (an ASR instruction with a special Unsigned Prefix).

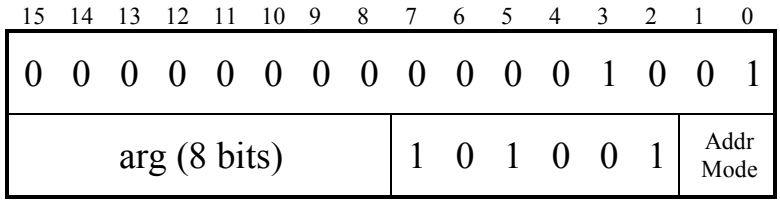
Count values greater than 32 will have the same effect as a count of 32.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Unsigned Prefix

<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: -Register + Data → Register

Syntax: NADD <Register>, <Data Specifier>

Examples:
```
nadd    x, #H'3F
nadd    al, @(34,x)
```

Description: Add the negated contents of the register to the specified data and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = ~Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4	3 2	1 0
arg (8 bits)	0 1 1 1	Reg	Addr Mode

<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: None

Syntax: NOP

Examples: nop

Description: Performs no operation.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Operation: Register | Data → Register

Syntax: OR <Register>, <Data Specifier>

Examples:
```
or    al, #H'0A
or    ah, @H'0102
```

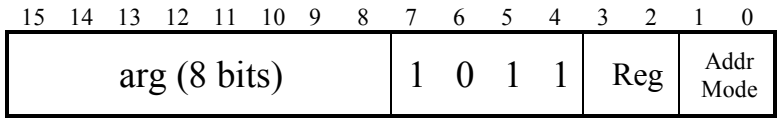
Description: OR the contents of the register with the specified data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: None

Syntax: PRINT <Register>, <Immediate>

Examples: print al, #H'123

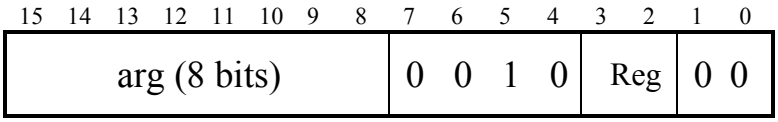
Description: A special debugging instruction only interpreted by the simulator, printing the immediate value and specified register on the display. The processor interprets the instruction as a NOP.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Shift A left by count, (C ← [AH:AL] ← C)

Syntax: ROL <Count Specifier>

Examples:
```
rol   #3
rol   @shift_count
```

Description: Shift the A register (combined AH and AL) left by count. Feed the old Carry flag into the least-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

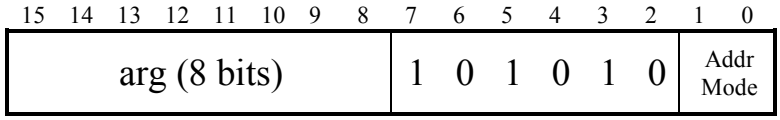
For shifts > 32 the count value is masked to the 5 LSBs so a shift of 33 is the same as a shift of 1.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Shift A right by count, (C → [AH:AL] → C)

Syntax: ROR <Count Specifier>

Examples:
```
ror   #3
ror   @shift_count
```

Description: Shift the A register (combined AH and AL) right by count. Feed the old Carry flag into the most-significant bit and feed the shifted out value into the Carry flag.

Notes: A shift of 0 sets the Carry flag as if a shift of 1 had been performed, but leaves the A register un-affected.

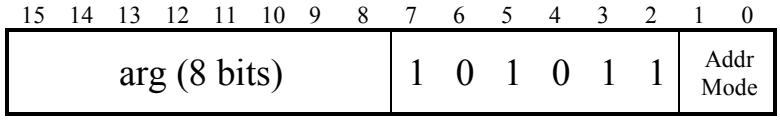
For shifts > 32 the count value is masked to the 5 LSBs so a shift of 33 is the same as a shift of 1.

Condition Codes:

C	S	N	Z
•	-	-	-

C Set with the last shifted out value.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Count Specifier>

Syntax	Name	'count'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: @(arg + Y) → Flags; {IXH, IX} → PC

Syntax: RTI @(arg, Y)

Examples: rti @(-1, y)

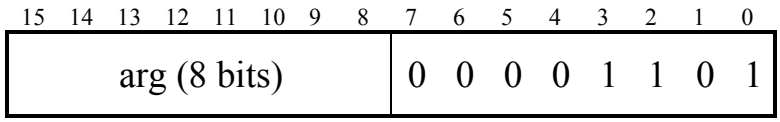
Description: Return from an interrupt by reloading the flags register from the stack and copying the X, XH registers into the program counter in a single atomic action.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Operation: {XH, X} → PC

Syntax: RTS

Examples: rts

Description: Program execution continues from contents of the X and XH registers. Same as BRA 0,X

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0

Operation: A ÷ Data → AL, Remainder → AH

Syntax: SDIV <Data Specifier>

Examples:

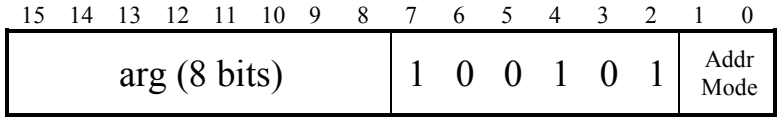
Description: The contents of the 32-bit A register (combined AH and AL) is divided by the specified data and stored in the AL register. The remainder of the division is stored in the AH register. All numbers are treated as signed values and the remainder always has the same sign as result (dividend).

Condition Codes:

C	S	N	Z
-	-	-	-

C Not Affected.
S Not Affected.
N Not Affected.
Z Not Affected.

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Process a SIF operation

Syntax: SIF

Examples: sif

Description: Processes a SIF operation during the instruction.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0

Operation: Enter Sleep mode

Syntax: SLEEP

Examples: sleep

Description: Force the processor into Sleep mode.

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Operation: AL x Data → A

Syntax: SMULT <Data Specifier>

Example:

Description: The contents of the AL register is multiplied by the specified data then stored in the A register. All numbers are treated as signed values.

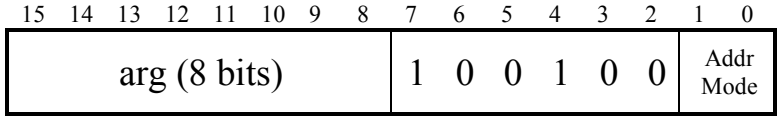
Notes:

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Register → Address

Syntax: ST <Register>, <Address Specifier>

Examples:
```
st   x, #H'3F
st   ah, @(23, y)
```

Description: Store the register with the specified address.

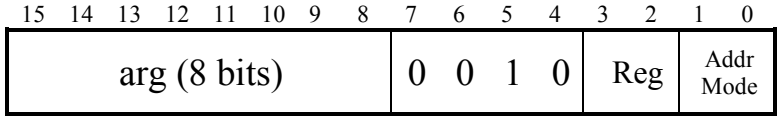
Notes: A number of special store instructions are available for accessing the flags register and interrupt mode registers. These instructions have a different format and are described on the next page.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if register negative. N = Reg[15]
Z Set if register zero. Z = ~(Reg[15] | ... | Reg[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
		Illegal mode	*00*
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

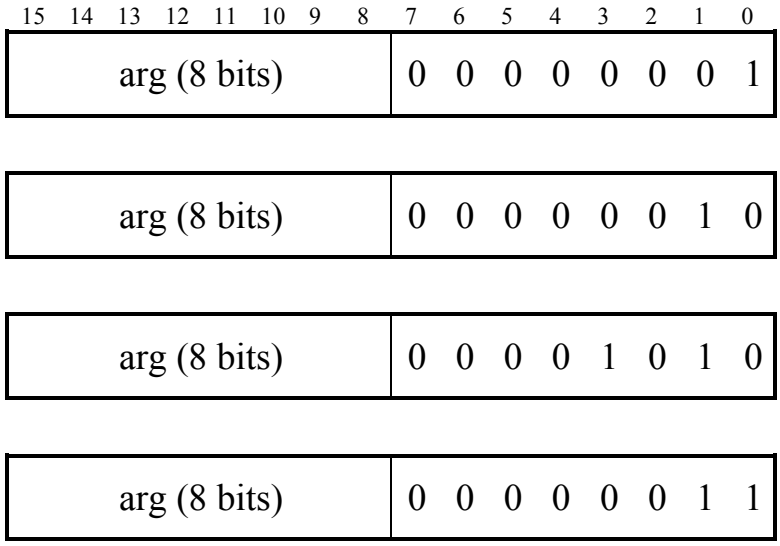
Register	Reg
AH	00
AL	01
X	10
Y	11

Notes: These special store instructions are accessible only in the Indexed Y address mode. No condition codes are affected.

Syntax:			
	ST	flags, @(arg, Y)	Flags[7:0] → @(arg^{16} + Y)
	ST	UX, @(arg, Y)	UX[15:0] → @(arg^{16} + Y)
	ST	XH, @(arg, Y)	if User mode LSB UXH[7:0] → @(arg^{16} + Y) else {IXH[7:0], UXH[7:0]} → @(arg^{16} + Y)
	ST	UY, @(arg, Y)	UY[15:0] → @(arg^{16} + Y)

Examples:
```
st   flags, @(-3, y)
st   ux, @(-4, y)
```

Instruction Format:



Operation: Register - Data → Register

Syntax: SUB <Register>, <Data Specifier>

Examples:
```
sub   y, #H'010A
sub   ah, @(23,y)
```

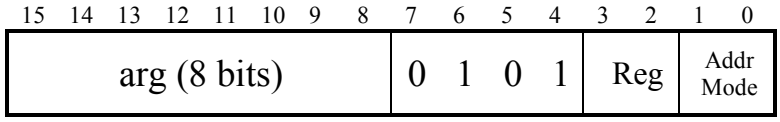
Description: Subtract the specified data from the contents of the register and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Register - Data - Carry → Register

Syntax: SUBC <Register>, <Data Specifier>

Examples:
```
subc    y, #H'010A
subc    ah, @(23,y)
```

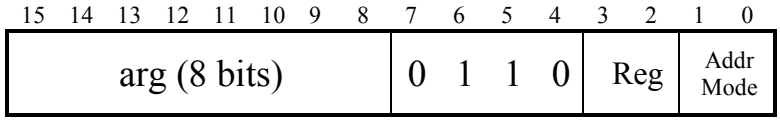
Description: Subtract the Carry and specified data from the contents of the register and store the result in the register. Can be used for signed or unsigned, integer or fractional values.

Condition Codes:

C	S	N	Z
•	•	•	•

C	Set if a carry is generated.	C = Result[16]
S	Set if true sign of result negative.	S = Reg[15] ^ Data[15] ^ Result[16]
N	Set if result negative.	N = Result[15]
Z	Set if result zero.	Z = ~(Result[15] | ... | Result[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

Operation: Data → Flag

Syntax: TST <Data Specifier>

Examples: tst #13

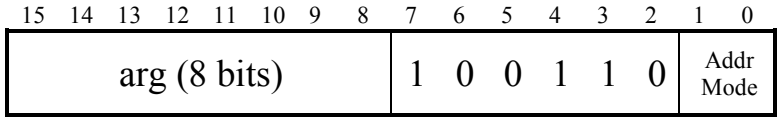
Description: Set the flags according to the specified data.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if data negative. N = Data[15]
Z Set if data zero. Z = ~(Data[15] | ... | Data[0])

Instruction Format:



<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: A ÷ Data → AL, Remainder → AH

Syntax: UDIV <Data Specifier>

Examples:

Description: The contents of the 32-bit A register (combined AH and AL) is divided by the specified data and stored in the AL register. The remainder of the division is stored in the AH register. All numbers are treated as unsigned values.

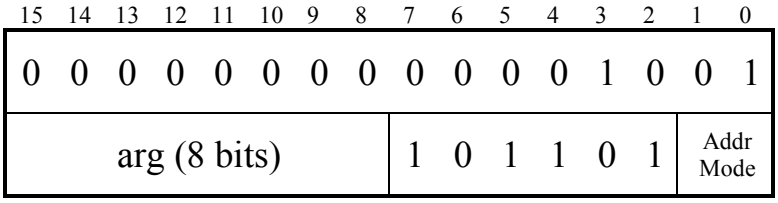
Notes: The base instruction is two words long (a SDIV instruction with a special Unsigned Prefix).

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Unsigned Prefix

<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: AL x Data → A

Syntax: UMULT <Data Specifier>

Example:

Description: The contents of the AL register is multiplied by the specified data then stored in the A register. All numbers are treated as unsigned values

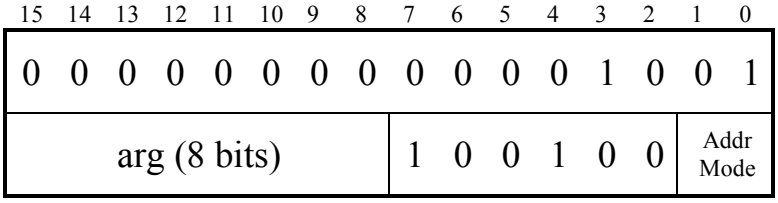
Notes:

Condition Codes:

C	S	N	Z
-	-	-	-

C Not affected.
S Not affected.
N Not affected.
Z Not affected.

Instruction Format:



Unsigned Prefix

<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

Operation: Register ^ Data → Register

Syntax: XOR <Register>, <Data Specifier>

Examples:
```
xor   al, #H'0A
xor   ah, @H'0102
```

Description: XOR the contents of the register with the specified data and store the result in the register.

Condition Codes:

C	S	N	Z
-	-	•	•

C Not affected.
S Not affected.
N Set if result negative. N = Result[15]
Z Set if result zero. Z = ~(Result[15] | ... | Result[0])

Instruction Format:

15 14 13 12 11 10 9 8	7 6 5 4	3 2	1 0
arg (8 bits)	1 1 0 1	Reg	Addr Mode

<Data Specifier>

Syntax	Name	'data'	Addr Mode
#arg	Immediate	arg^{16}	00
@arg	Direct	Contents of address (arg^{16})	01
@(arg, X)	Indexed X	Contents of address (arg^{16} + X)	10
@(arg, Y)	Indexed Y	Contents of address (arg^{16} + Y)	11

<Register>

Register	Reg
AH	00
AL	01
X	10
Y	11

5 KEY WORDS

Directives

%OUT	.SMALL	ENDS	IFIDN
.ALL	.SEG	EQU	IFIDNI
.CODE	.VAR	EQU <…>	IFNB
.CONST	CEQU	EXITM	IFNDEF
.CPACK	DC	FILE	INCLUDE
.CSEG	DEQU	IF	LABEL
.DATA	DS	IFB	LOCAL
.ERR	ELSE	IFDEF	MACRO
.LARGE	ENDIF	IFDIF	MODULE
.LITVALS	ENDMAC	IFDIFI	ORG
.PACK	ENDMOD	IFE	STRUC
.PAGE			

Operators

ABS	FRMAN	LENGTH	PASC
AND	GE	LOGIC	SHL
E	GT	LT	SHR
EQ	HWRD	LWRD	XOR
FEXP	INDEX	MOD	+
FMAN	INT	NE	-
FRBIT	LDEXP	NOT	*
FREXP	LE	OR	/

Instructions

ADD	BPL	NOP	SDIV
ADDC	BRA	OR	SIF
AND	BRK	PRINT	SLEEP
ASL	BRXL	ROL	SMULT
ASR	BSR	ROR	ST
BC	CMP	RTI	SUB
BCC	*DIV*	RTS	SUBC
BCS	LD	*SAL*	TST
BEQ	LSL	*SAR*	UDIV
BLT	LSR	*SCL*	UMULT
BMI	*MULT*	*SCR*	XOR
BNE	NADD		

Italicized keywords are depreciated as of XAP2, but remain reserved words.

Registers

AL	SP	UY	XH
AH	UX	X	Y
FLAGS			

6 INDEX

!

! 60

$

$ 28
$??CODE 20
$??CONST 20
$??DATA 20
$??LARGE 16
$??SMALL 16
$??VAR 20
$??XAP2 16

%

% 61
%OUT 68

&

& 57, 60

.

.ALL 16
.CODE 20
.CONST 20
.DATA 20
.ERR 64
.LARGE 16
.PAGE 70
.SEG 21
.SMALL 16
.VAR 20

<

<> 61

=

= 32

A

Addressing modes 17
  Branch instructions 17, 78
  General instructions 17, 78
  Immediate 17
  Indexed X 17
  Indexed Y 17
  Indirect 17
  PC Relative 17
  X Relative 17
Arithmetic operators 40
Assembler Options
  -code n 15, 16
  -large 15
  -small 15, 16
Assignment operators 43

B

Bitwise Operators 41

C

CEQU 31
Code table 11
Command line 7, 9
Conditional assembly 38, 63
Constant table 11
Continuation character 70

D

DC 36
Define constant 36
Define storage 36
DEQU 31
DS 36

E

ELSE 63
ENDIF 63
ENDMAC 53
ENDMOD 26
ENDS 47
EQU 31
Equates
  Numeric 31
  Redefinable 32
  String 32
Error messages 72
EXITM 54, 57
Expressions 46

F

FILE 69
Floating point arithmetic 33
Fractional arithmetic 34

G

Generating errors 64
Generating output 68

I

IF 63

IFB 64
IFDEF 63
IFDIF 64
IFDIFI 64
IFE 63
IFIDN 64
IFIDNI 64
IFNB 64
IFNDEF 63
INCLUDE 26, 53, 69
Instruction set 78
Instructions
ADD 82
ADDC 83
AND 84
ASL 85
ASR 86
BC 87
BCC 88
BCS 89
BEQ 90
BLT 91
BMI 92
BNE 93
BPL 94
BRA 95
BRA2 96
BRA3 97
BRK 98
BRXL 99
BSR 100
CMP 101
LD 102
LSL 104
LSR 105
NADD 106
NOP 107
OR 108
PRINT 109
ROL 110
ROR 111
RTI 112
RTS 113
SDIV 114
SIF 115
SLEEP 116
SMULT 117
ST118
SUB 120
SUBC 121
TST 122
UDIV 123
UMULT 124
XOR 125

L

LABEL 30
Listing file 13
LOCAL 54, 57
Location counter 22
Logic operators 42

M

MACRO 53
Macro names 54
Macros 38, 53, 54
Expression operator 61
Literal character operator 60
Literal text operator 61
Nesting 62
Recursion 62
Redefining 62
Substitute operator 60
Maximum line length 70
Memory Model
Compliance Directives 16
Differences 15
Selection 15
Variables 16
MODULE 26

N

Number formats
ASCII 37
Binary 37
Decimal 37
Floating point 37
Fractional 37
Hex 37

O

Operands 39
Operator precedence 46
Operators 39
FRBIT 43
HWRD 43
INDEX 48
INT 43
LENGTH 48
LWRD 43
PASC 44, 68
ORG 20
Original line propagation 69

P

Page break in listing file 70

R

Relation operators 41

ROM start address 22

S

Segments 20, 37
- Code 20
- Constant 20
- Data 20
- User Defined 21
- Variables 20

String formats
- Double quotes 36
- Escape character 36
- Packed 36
- Single quotes 36

STRUC 47
- Defining 48
- Embedded equates 49
- Field definitions 47
- Structure types 47

Symbol names 28, 47
Symbol table 12
Symbols
- Global 28
- Local 28

V

Variable Length Instructions 18
Variables 31
Verilog 11