

AmigaFlight Assembly

Andrew Duffy Morris

COLLABORATORS

	<i>TITLE :</i> AmigaFlight Assembly		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Andrew Duffy Morris	July 20, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	AmigaFlight Assembly	1
1.1	AmigaFlight® Help: 68000 Assembly Language	1
1.2	AmigaFlight® Help: The 68000 Family	1
1.3	AmigaFlight® Help: The 68000 Address Range	3
1.4	AmigaFlight® Help: The 68000 Chip Characteristics	3
1.5	AmigaFlight® Help: The 68000 Data Types	5
1.6	AmigaFlight® Help: The 68000 Addressing Modes	6
1.7	AmigaFlight® Help: The 68000 Status Register Flags	6
1.8	AmigaFlight® Help: The 68000 Stack	7
1.9	AmigaFlight® Help: 68000 Interrupts	7
1.10	AmigaFlight® Help: The 68000 Instruction Set	7
1.11	AmigaFlight® Help: Data Types	7
1.12	AmigaFlight® Help: Data Types	8
1.13	AmigaFlight® Help: Data Types	8
1.14	AmigaFlight® Help: Data Types	8
1.15	AmigaFlight® Help: Data Types	8
1.16	AmigaFlight® Help: Addressing Modes	8
1.17	AmigaFlight® Help: Addressing Modes	9
1.18	AmigaFlight® Help: Addressing Modes	9
1.19	AmigaFlight® Help: Addressing Modes	10
1.20	AmigaFlight® Help: Addressing Modes	10
1.21	AmigaFlight® Help: Addressing Modes	11
1.22	AmigaFlight® Help: Addressing Modes	12
1.23	AmigaFlight® Help: Addressing Modes	13
1.24	AmigaFlight® Help: Addressing Modes	13
1.25	AmigaFlight® Help: Addressing Modes	15
1.26	AmigaFlight® Help: Addressing Modes	15
1.27	AmigaFlight® Help: About The Stack	15
1.28	AmigaFlight® Help: Manipulating The Stack	16
1.29	AmigaFlight® Help: Flags	17

1.30 AmigaFlight® Help: Flags	17
1.31 AmigaFlight® Help: Z Flag	17
1.32 AmigaFlight® Help: N Flag	17
1.33 AmigaFlight® Help: X Flag	18
1.34 AmigaFlight® Help: T Flag	18
1.35 AmigaFlight® Help: S Flag	18
1.36 AmigaFlight® Help: Interrupt Handling	18
1.37 AmigaFlight® Help: Interrupt Masks	19
1.38 AmigaFlight® Help: 68000 Assembly Index	19

Chapter 1

AmigaFlight Assembly

1.1 AmigaFlight® Help: 68000 Assembly Language

Documentation for
Motorola 68000 Assembly Language

Introduction to the 68000 Family
Address Space
Chip Characteristics
Data Types
Addressing Modes
The Stack
Status Register Flags
Interrupts
Instruction Set
AmigaFlight Help

Written by XCNT :

Andrew Duffy
Chris Morris

Copyright © 1994, XCNT Productions

1.2 AmigaFlight® Help: The 68000 Family

The 68000 Family

=====

In 1979, Motorola introduced the first implementation of the M68000 16/32-bit microprocessor architecture - the MC68000. The MC68000, with a 16-bit data bus and 24-bit address bus, was only the first in a now extensive family of processors which implement a comprehensive, extensible computer architecture. The MC68000 was soon followed by the MC68008, with an 8-bit data bus and 20-bit address bus, and by the MC68010, which introduced the virtual machine aspects of the M68000 architecture.

When the 32-bit MC68020 was introduced in 1984, the MC68000 had already been established as a popular and successful microprocessor. The MC68020 extended the basic characteristics of the MC68000 to yield an enhanced microprocessor in the 32-bit class possessing many capabilities not found in the earlier 16-bit processors. One important advance was the ability to be combined with coprocessors such as the floating-point mathematics coprocessor (MC68881) and the memory management coprocessor (MC68851).

The MC68030 went one step further by combining an improved version of the MC68020 with the memory management capability of the MC68851 coprocessor on one chip. This combination increased the performance of a system while reducing the number of components. The MC68030 also has a MC68882 mathematics coprocessor to allow this two-chip set to function as a complete processing unit.

The MC68040 combines all the features of the MC68030 with the abilities of the MC68882 mathematics coprocessor into a single chip for even more speed and performance.

Evolution of the Motorola Microprocessors

=====

1979	MC68000	6Mhz CPU
1980	MC68000	8Mhz CPU
1981	MC68000	10Mhz CPU
1982	MC68000	12.5Mhz CPU
	MC68008	CPU (Reduced bus MC68000)
1983	MC68010	CPU (Virtual machine M68000)
1984	MC68012	CPU (Extended virtual machine M68000)
	MC68020	CPU (Advanced 32-bit M68000)
1987	MC68030	CPU
1990?	MC68040	CPU

The M68000 or the i8086

=====

The MC68000 has a 16-bit data bus and 16-bit arithmetic logic unit (ALU), but all of the internal registers are 32 bits wide. This increased register size is one of the most important ways in which Motorola provides a clear upgrade path. Programs written for the 68000 family take full advantage of 32-bit operations and will run on true 32 bit machines. This upward compatibility is an enormously powerful concept, especially when contrasted with the approach that Intel took with its 8086 family.

The 8086 family supports compatibility by crippling its high-end processors. In other words, the 80286 processor runs 8086 programs by

disabling many of the 80286 features. Each generation in Intel's processor family advances by "gluing" more features onto the new chip.

Motorola, however, designed a full 32-bit architecture from the beginning, kept that structure as the programming model and implemented it on 16-bit machines. As a result, most programs written for the 68000 family run equally well, with very little modification on all members of the family.

1.3 AmigaFlight® Help: The 68000 Address Range

68000 Address Space

=====

The 68000 family supports a full 32-bit (4 gigabyte) address space. Only the 68020 and upwards brings all of the address lines out of the chip package, but the 16 megabytes supported by the 68000 is quite respectable. This is due to the 24-bit address bus :

```

    24
2 bits == 16777216 bits == 16 Mb

```

1.4 AmigaFlight® Help: The 68000 Chip Characteristics

The 68000 Chip Characteristics

=====

The Motorola 68000 has a 16/32 bit architecture. It has a 16-bit data bus and a 24-bit address bus, while the full architecture provides for 32-bit address and data buses. It is completely code compatible with the HCMOS MC68000 (a variation of the 68000), MC68008 (an 8-bit data bus implementation of the 68000) and is upward code compatible to the MC68010/MC68012 virtual extension and the MC68020, the MC68030EC, the MC68030, the MC68040EC, the MC68040, and finally, the yet to be released MC68060 (better than the Pentium any day!). This is possible because the user programming model is identical for all six processors and the instruction sets are proper sub-sets of the complete architecture.

Resources available to the MC68000 user consist of the following :-

```

17 x 32-bit Data and Address registers
16Mb direct addressing range (no pages or extended etc.)
56 powerful instruction types
Operations on 5 main data types
Memory mapped I/O
14 addressing modes

```

```

    3    1 1
    1    6 5 8 7 0
+-----+-----+-----+
|      |      |      | D0
|      |      |      | D1
|      |      |      | D2

```

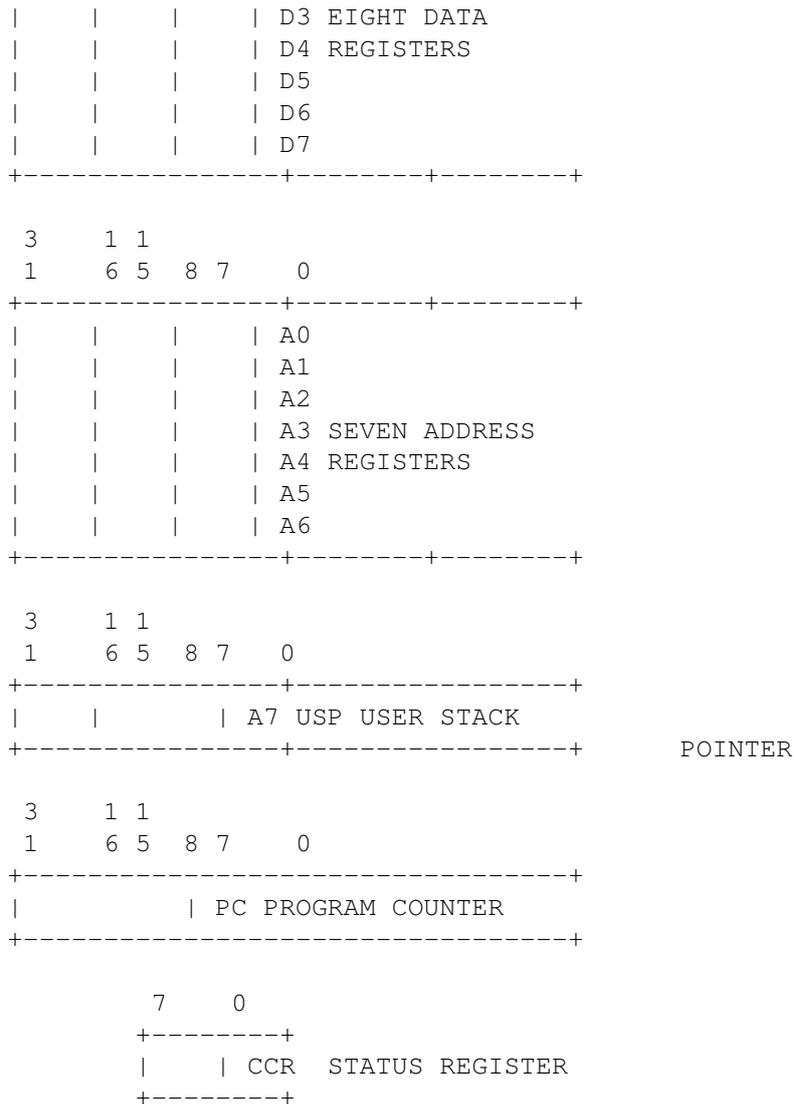


Figure 1. User Programming Model

As shown in the user programming model (Figure 1.), the MC68000 offers 16 32-bit registers and a 32-bit program counter.

Data Registers

The first eight registers (D0-D7) are used as data registers for byte (8-bit), word (16-bit), and long word (32-bit) operations. The least significant bit is addressed as bit zero, the most significant bit is addressed as bit 31.

Address Registers

The second set of seven registers (A0-A6) and the user stack pointer (USP) may be used as software stack pointers and base registers. In addition, the registers may be used for word and long word operations. All of the 16 registers may be used as index registers.

In supervisor mode, the upper byte of the status register and the supervisor stack pointer (SSP) are also available to the programmer.

These registers are shown in Figure 2. below:

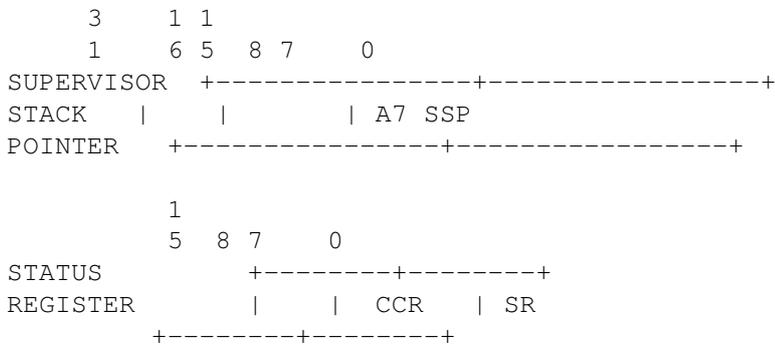


Figure 2. Supervisor Programming Model

The status register (Figure 3.) contains the interrupt mask (eight levels available) as well as the condition codes: eXtend (X), Negative (N), Zero (Z), oVerflow (V), and Carry (C). Additional status bits indicate that the processor is in Trace (T) mode and in Supervisor (S) or user state.

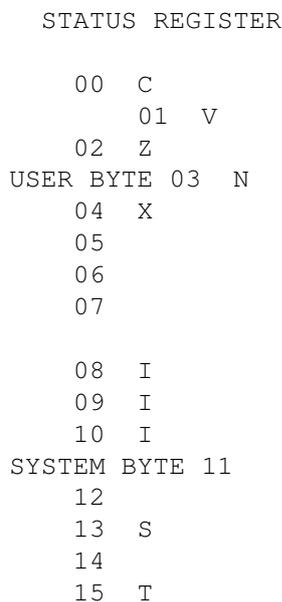


Figure 3. Status Register

1.5 AmigaFlight® Help: The 68000 Data Types

Motorola 68000 Assembly Language

Data Types

- Bits
- BCD Digits
- Bytes
- Words
- Long Words

1.6 AmigaFlight® Help: The 68000 Addressing Modes

Motorola 68000 Assembly Language

Addressing Modes

- Data Register Direct
- Address Register Direct
 - Absolute Short
 - Absolute Long
 - Relative with Offset
 - Relative with Index Offset
 - Register Indirect
 - Postincrement Register Indirect
 - Predecrement Register Indirect
 - Register Indirect with Offset
- Indexed Register Indirect with Offset
- Immediate
- Quick Immediate
- Implied Register

1.7 AmigaFlight® Help: The 68000 Status Register Flags

Motorola 68000 Assembly Language

Status Register Flags

- Interrupt Masking
 - eXtended Flag
 - Negative Flag
 - Zero Flag
 - oVerflow Flag
 - Carry Flag
 - Trace Flag
 - Supervisor Flag

1.8 AmigaFlight® Help: The 68000 Stack

Motorola 68000 Assembly Language

The Stack

About The Stack
Manipulating The Stack

1.9 AmigaFlight® Help: 68000 Interrupts

Motorola 68000 Assembly Language

Interrupts

Interrupt Handling
The Interrupt Mask

1.10 AmigaFlight® Help: The 68000 Instruction Set

Motorola 68000 Assembly Language

Instruction Set

Data Movement Instructions
Integer Arithmetic Instructions
Logical Instructions
Shift and Rotate Instructions
Bit Test and Manipulation Instructions
Binary Coded Decimal Instructions
Flow Control Instructions
System Control Instructions

1.11 AmigaFlight® Help: Data Types

Bits
====

A single bit can only have the value 0 or 1.

1.12 AmigaFlight® Help: Data Types

BCD Digits
=====

Binary Coded Decimal (BCD) digits are made up of four bits.

In binary-coded decimal, BCD, each of the decimal digits of a number is represented by four bits. The decimal number 72509 for example, is represented in BCD as 0111 0010 0101 0000 1001.

1.13 AmigaFlight® Help: Data Types

Bytes
=====

A byte is made up of eight bits and allows 256 different values.

1.14 AmigaFlight® Help: Data Types

Words
=====

A word is made up of sixteen bits and allows 65,536 values.

1.15 AmigaFlight® Help: Data Types

Long Words
=====

A long word is made up of thirty two bits and allows 4,294,967,296 values.

1.16 AmigaFlight® Help: Addressing Modes

Register Direct Rn
=====

In the register direct mode addressing mode, the operand is in the specified address, or data register. Most instructions use either a data register or an address register as one of the operands. Registers are most commonly intermediate values or heavily used variables in a section of code.

1.17 AmigaFlight® Help: Addressing Modes

Absolute value

=====

The absolute addressing mode has two variations - absolute short and absolute long. With absolute short mode, the lower half of the effective address follows the opcode in memory as a word value. The specified word value is sign extended and then used as the address of the operand in question.

The absolute short addressing mode can only access the lowest or highest 32K memory locations. This mode provides a short, quick way to use programs or temporary storage. It is short and quick because it saves a word of memory and a read cycle.

The following example loads the TRAPV vector, and probably would only be executed in Supervisor Mode.

```
LEA.L MY_TRAPV_ROUTINE,A0 ;GET A SUBROUTINE ADDRESS
MOVEA.L A0,$001C.W      ;SAVE IT IN TRAPV VECTOR
```

With absolute long addressing, the effective address occupies two words of memory immediately after the opcode. This addressing mode gives the user access to any memory location. The labels used are commonly called global variables. For example, if you assigned the label DATALOC to a memory location, you could store information in that location with the following line of code (which transfers a word of information from data register D7 to the memory location DATALOC):

```
MOVE.W D7,DATALOC      ;SAVE SOME DATA
```

NOTE:

In a multiprocessing environment (such as an Amiga or Apple Mac) a machine code program should always use labels when referring to any type of absolute data. This allows the assembler to generate the correct relocation information. Without relocation information, the machine language program cannot execute correctly in the multiprocessing environment.

A multitasking environment must be able to move programs around in memory. If you assign absolute constants (such as telling the program to jump to a specific address, "hitting the hardware") without relocation information, your program will crash whenever it is moved to a different location in memory. All the absolute addresses will be wrong.

1.18 AmigaFlight® Help: Addressing Modes

Program Counter Relative with Displacement d16(PC)

=====

With this addressing mode, the 16-bit displacement value is added to the program counter and used as the address of the operand fetched or stored. The program counter is unmodified by this addressing mode.

This has three important uses:

- When the source code makes reference to a label, and the

referenced label is within 32768 bytes of the current location counter, as in the statement JUMP LABEL

- For constant jumps through a table
- To find the address of the current instruction as in the following statement: LEA.L -4(PC),A0

1.19 AmigaFlight® Help: Addressing Modes

Program Counter Relative with Index and Displacement d8(PC,Rn)

=====

In program counter relative with index and displacement addressing, the eight bit displacement, the program counter, and the specified secondary register are added together to generate the address of the operand. This calculated value is used to fetch or store data used by the instruction.

Either a data or address register can play the role of the secondary register, commonly known as the index register. This register may act as a 16- or 32-bit value. By default, the register will be accessed as a 16-bit value. To specify the size of this register, append to the opcode either .L (as in LEA.L) for a 32-bit value, or .W (as in LEA.W) for a 16-bit value.

This addressing mode is most useful when doing a variable jump through a jump table as in this example:

```
MOVE.W INDEX,D0 ;GET JUMP TABLE INDEX
LSL.L #2,D0 ;MULTIPLY BY TWO
JMP 2(PC,D0) ;CALL SUBROUTINE IN TABLE
BRA EXIT
```

```
DC.L SUBROUTINE0 ;INDEX 0
DC.L SUBROUTINE1 ;INDEX 1
DC.L SUBROUTINE2 ;INDEX 2
```

EXIT:

1.20 AmigaFlight® Help: Addressing Modes

Address Register Indirect (An)

=====

In the address register indirect addressing mode, the address of the operand is in the specified address register. This 32-bit value is used to fetch the operand for the calculation. On the MC68000 and MC68010, only the lowest 24 (out of a possible 32) bits of the address are used. On the MC68008, only the bottom 20 bits are used. The programmer, however, should not use the upper 8 bits of the address register for flag bits or non address data. This trick was used in some early MC68000 programs - much to their detriment when they were ported to the MC68020, a microprocessor that uses all 32 address bits.

The address register indirect mode is commonly used just after an address has been calculated, or when the same address is used repeatedly.

For example, the following code uses the same address multiple times in a loop, but only calculates it once. After it's calculated, it's placed in address register A0:

```

    LEA USEDALOT,A0
LAB: MOVE.W (A0),D0 ; LOAD A COMMON VARIABLE THAT
      ; GETS TRASHED
;·
;· ; DO SOME WORK
;·
    BRA LAB

```

This addressing mode does not modify the specified address register.

1.21 AmigaFlight® Help: Addressing Modes

Address Register Indirect with Postincrement (An)+

Address register indirect with postincrement is similar to address register indirect, but as the name implies, the value in the address register is automatically increased after each use. If you use this addressing mode with a long-word instruction (like MOVE.L), the address register will be incremented by four. If you use it with a word instruction (like MOVE.W), it will be incremented by two. And if you use it with a byte instruction (like MOVE.B), it will be incremented by one. This addressing mode provides an easy means of processing arrays, stacks, queues, and other data structures.

If the address register is the stack pointer (SP or A7) and the operand size is a byte, then the stack pointer is automatically incremented by two instead of one. This keeps the stack properly aligned at all times, ie. on an even address.

If the assembly program has created an upward growing stack, then a stack push operation may be performed in the following way. (Although stacks normally grow downward on the Amiga, it is not necessary that the programmer use stacks in this manner.)

```

; ; STACK PUSH (STACK GROWING UPWARD)
; ; (ASSUMING A3 IS STACK POINTER)
MOVE.L D0,(A3)+ ; PUSH D0 TO THE TOP OF THE STACK FOR
; ; FUTURE USE

```

Some more examples:

```

; ; QUEUE SAVE/RETRIEVE (ASSUME A2 IS HEAD
; ; OF QUEUE, A3 IS TAIL OF QUEUE)
; ; CHECK QUEUE LIMITS
MOVE.W D1,(A3)+ ; SAVE ITEM
; ; CHECK QUEUE LIMITS
MOVE.W (A2)+,D1 ; GET ITEM
MOVE.L (A0)+,(A1)+ ; MOVE LONG WORD POINTED TO BY A0 TO LONG
; ; WORD POINTED TO BY A1, THEN INCREMENT
; ; BOTH A0 AND A1 BY 4 AFTER THE
; ; INSTRUCTION.
; ; THIS IS VERY USEFUL FOR COPYING LARGE

```

```
    ; CHUNKS OF DATA IN A LOOP.
```

Remember that the amount of increment depends upon on the size specifier on the actual instruction.

1.22 AmigaFlight® Help: Addressing Modes

Address Register Indirect with Predecrement -(An)

=====

Using address register indirect with predecrement causes the address of the operand contained in the address register to be decremented by one, two, or four, depending upon the size of the operand specified, before the operation takes place. The address in the specified address register is used to fetch the operand or store data. If the address register is the stack pointer (SP or A7), and the operand size is a byte, the stack pointer is automatically decremented by two instead of one. This keeps the stack properly aligned at all times, ie on an even address.

Register indirect with predecrement mode has many uses. These include, among other things, array, stack, and queue manipulation.

If the assembly program uses a downward growing stack, automatically available with the SP register, a stack push operation is readily available.

```
    ; STACK PUSH (STACK GROWING DOWNWARD)
MOVE.L D0,-(SP) ; PUSH D0 TO THE TOP OF THE STACK FOR
    ; FUTURE USE
```

If the assembly program has created an upward growing stack, then a stack pop operation may be performed in the following manner:

```
    ; STACK POP (STACK GROWING UPWARD)
    ; (ASSUMING A3 IS STACK POINTER)
MOVE.L -(SP),D0 ; TAKE THE TOP ELEMENT OFF THE STACK
    ; AND SAVE IT IN D0 FOR LATER USE
```

Some more examples:

```
    ; QUEUE SAVE/RETRIEVE (ASSUME A2 IS
    ; HEAD OF QUEUE, A3 IS TAIL OF QUEUE)
    ; CHECK QUEUE LIMITS
MOVE.W D1,-(A3) ; SAVE ITEM
    ; CHECK QUEUE LIMITS
MOVE.W -(A2),D1 ; GET ITEM
MOVE.W -(A0),-(A1) ; DECREMENT BOTH A0 AND A1 BY TWO, THEN
    ; MOVE THE WORD POINTED TO BY A0 TO THE
    ; WORD POINTED TO BY A1.
    ; THIS IS VERY USEFUL FOR COPYING LARGE
    ; CHUNKS OF DATA IN A LOOP.
```

Remember that the amount of decrement depends upon the size specifier on the actual instruction.

1.23 AmigaFlight® Help: Addressing Modes

Address Register Indirect with Displacement d16(An)

Address register indirect with offset uses the address contained in the specified address register added to a 16-bit displacement value as the address of the operand to be fetched or stored. The address register is not modified by this addressing mode.

This addressing mode has many uses. The most common use is accessing stack variables that exist at constant locations.

Consider the following example:

```
MOVE.L  VAR3,-(SP)  ; SAVE THIRD VARIABLE
MOVE.B  VAR2,-(SP)  ; SAVE SECOND VARIABLE
MOVE.W  VAR1,-(SP)  ; SAVE FIRST VARIABLE
JSR SUBROUTINE
```

The stack will now look like this:

	Even Byte	Odd Byte	
SP+10	Variable 3	High Word	SP+11
SP+8	Variable 3	Low Word	SP+9
SP+6	0 0 0 0 0 0 0 0	Variable 2	SP+7
SP+4	Variable 1		SP+5
SP+2	Return program counter	High Word	SP+3
SP+0	Return program counter	Low Word	SP+1

```
        ; RETURN VARIABLES FROM STACK
JSR SUBROUTINE
MOVE.W  4(SP),D0  ; TO ACCESS VARIABLE 1
ADD.B   7(SP),D0  ; TO ACCESS VARIABLE 2
MOVE.L  8(SP),A0  ; TO ACCESS VARIABLE 3
MOVE.L  D0,(A0)   ; (VARIABLE 3) = VARIABLE 1 + VARIABLE 2
```

```
        ; ANOTHER WAY TO RETURN VARIABLES
JSR SUBROUTINE
TST.W  (SP)+      ; POP VARIABLE 1
TST.B  (SP)+      ; POP VARIABLE 2
TST.L  (SP)+      ; POP VARIABLE 3
```

```
        ; MORE EFFICIENT WAY TO RETURN VARIABLES
JSR SUBROUTINE
ADDQ.L #8,SP      ; POP ALL THREE SIMULTANEOUSLY
```

1.24 AmigaFlight® Help: Addressing Modes

Address Register Indirect with Index and Displacement d8(An,Rn)

=====
 In this addressing mode, the eight-bit displacement, the specified address register, and the specified index register are added together to generate the address of the operand. This calculated value is used to fetch or store the data used by the instruction.

The second register, commonly known as the index register, is either a data register or an address register. This register is referenced as a 16-bit or a 32-bit value. By default, the register acts as a 16-bit value. To specify the size of this register, append to the opcode either .L for a 32-bit value, or .W for a 16-bit value.

This addressing mode is very useful for array indexing. Consider an array of data structures, in which each structure is 16 words long. The following code fragment totals the second words of the array.

```
LEA.L ARRAY_OF_STRUCTS,A0
MOVE.L NUM_OF_STRUCTS,D1
LSL.L #4,D1      ; GET MAXIMUM INDEX (NUM*16)
MOVEQ #0,D0     ; INIT INDEX
LOOP: ADD.W 2(A0,D0),D2 ; SUM = SUM + NEXT_ELEMENT
      ADD.W #16,D0    ; INDEX = INDEX + STRUCT_SIZE
      CMP.W D1,D0    ; IS D0-D1 < 0 ?
      BLT LOOP      ; YES, DO NEXT ITERATION
      .
      .
      .
rest of the code
```

```
WORD
  1 1 1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 1 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 2 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 3 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 4 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      .
      .
      .
      .
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 23 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 24 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 25 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
Structure 26 | | |$ \times $ | | | | | | | | | | | | | | | | |
  +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
      ^
      |
      Program adds these words together
```

Another example is a quick multiply by two in an address register:

```
LEA.L 0(A0,A0.L),A0 ; MULTIPLY A0 BY TWO
```

1.25 AmigaFlight® Help: Addressing Modes

Immediate #value

=====

The specified value is used as the source operand for the instruction. This addressing mode is used to load a constant value. Every time this addressing mode is used, there is one less constant to store in data space. The data follows immediately after the opcode. The data can be a byte, a word, or a long word.

The MC68000 has a special immediate mode for small operands. In this mode, the data is actually contained within the opcode itself. This quick mode can move a number in the range of -128 to +128 to a register or memory location, or add or subtract numbers from 1 to 8.

The following is an example of immediate addressing mode.

```
AND.L #$7F,D0 ; MASK OUT UPPER 25 BITS
OR.W  #$8000,D0 ; TURN ON SIGN BIT (WORD SIZED)
BMI CONT ; BRANCH IF MINUS
NOP ; THIS IS NEVER EXECUTED
CONT:
rest of program here
```

1.26 AmigaFlight® Help: Addressing Modes

Inherent or Implied

=====

Inherent addressing is the easiest of all - the microprocessor knows from the opcode alone which addresses to use. For example, an RTS instruction has no operand field, yet the microprocessor knows to fetch the return address from the stack. Some instructions that require no operands are NOP (NoOperation), RESET, RTE, TRAP, etc.

1.27 AmigaFlight® Help: About The Stack

About The Stack

=====

A stack is a data structure in which the first element put in is the last one to be taken out (FILO) or conversely, the last item to be stacked is the first to be unstacked (LIFO).

A simple use of a data stack is to reverse the order of a list of numbers or characters.

- The 68000 stacks grow towards the low end of memory (address 0).
- All eight address registers can maintain a stack i.e. be used as stack pointers.
- Address register 7 is only special in that it is used automatically by the hardware for subroutine linkage. The

- mnemonic SP may be used for A7.
- The stack pointer contains the address of the last item on the stack.

1.28 AmigaFlight® Help: Manipulating The Stack

Manipulating The Stack

=====

```

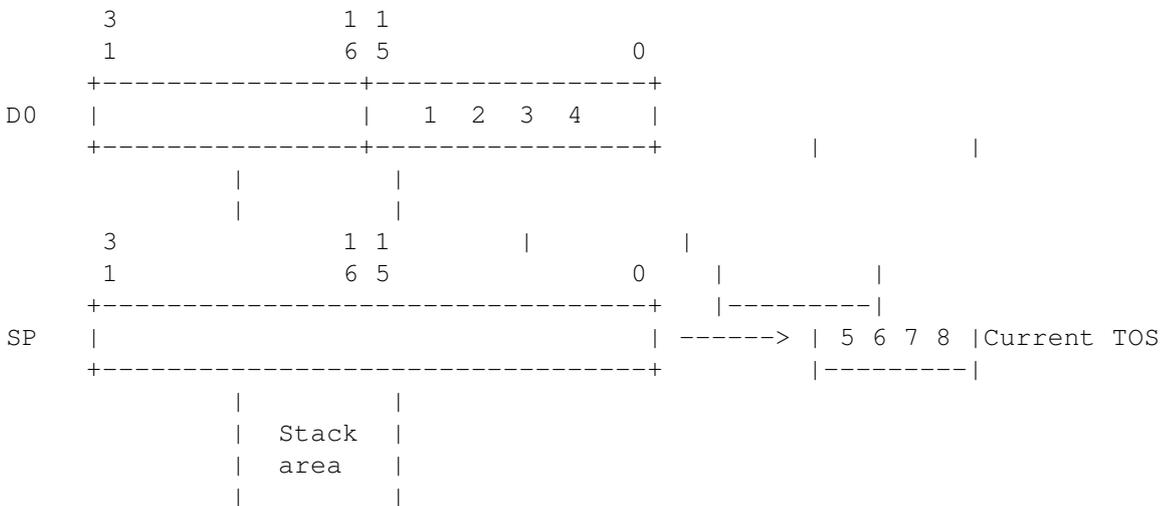
MOVE.W D0,-(SP)  Stack or push D0
    Copy the contents of the low order word of
    D0 onto the stack (where SP is pointing
    after being decremented by two (bytes).

MOVE.W (A7)+,D0  Unstack or pop into D0.
    Copy the contents of the top of the stack
    (where A7 ~is ~pointing) ~into ~D0 ~then
    increment A7 by two (bytes).

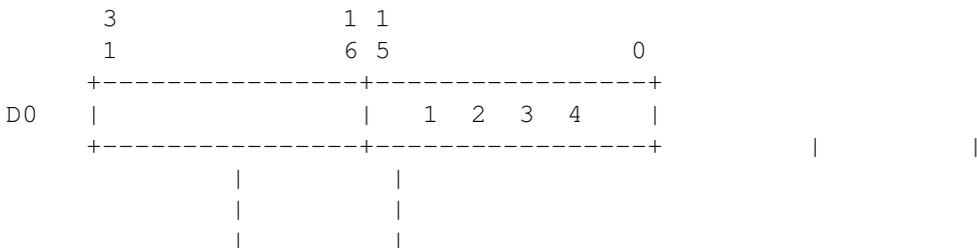
ADD.L (SP)+,D0  Unstack ~and ~add ~a ~~longword ~~to ~~D0,
    increment SP by a longword.
    
```

Consider the stack before the following instruction has executed:-

```
MOVE D0,-(SP)
```



Then, after MOVE D0,-(SP) is executed the stack will be :-



1.33 AmigaFlight® Help: X Flag

The Extend Flag X - bit 4

=====

This bit is used in many Extend instructions, such as ADDX. It provides a mechanism for multiprecision arithmetic. The extend bit is usually set or reset the same as the C bit.

1.34 AmigaFlight® Help: T Flag

The Trace Flag T - bit 15

=====

This bit, when set, is used to force the microprocessor into Trace Mode. The idea of the trace mode is to enable the microprocessor to go through the program one instruction at a time, enabling the contents of the registers to be examined between instructions. This is something that aids debugging of programs, and which can be achieved with virtually any microprocessor using a suitable monitor program, but it is something which is more easily implemented with the 68000 and its trace mode.

1.35 AmigaFlight® Help: S Flag

The Supervisor Mode Flag S - bit 13

=====

This bit selects the operation mode, when set (equal to 1) the microprocessor is operating in Supervisor Mode, and when clear, the microprocessor is operating in User Mode.

1.36 AmigaFlight® Help: Interrupt Handling

Interrupt Handling

=====

The 68000 provides 7 ~levels of ~interrupts, ~all ~of ~which ~are recognised and serviced based upon the priority set by the interrupt mask. The ~interrupt mask ~consists of ~three ~interrupt mask ~bits (I0, I1, I2) which are ~part ~of the 16-bit status ~register. These three bits indicate the current processor interrupt priority level which ranges between 0 and 7. Interrupt request level zero ~indicates ~that no interrupt service is requested. When ~an ~interrupt level ~from ~1 through 6 is ~requested, the processor compares the interrupt ~request level to the interrupt mask to determine whether the interrupt should be processed. Interrupt requests are ignored for all interrupt request levels that are less than or equal to the current processor ~priority ~level as determined by the interrupt mask bits.

Level 7 interrupts are non-maskable i.e. they cannot be interrupted even by another level 7 interrupt.

1.37 AmigaFlight® Help: Interrupt Masks

The Interrupts Mask I2 I1 I0 - bits 8, 9 and 10

=====
 This 3-bit number ranging from 000 to 111 indicates the current level of interrupts. This mask is set while in supervisor state: interrupts above a certain priority level are recognised, while the remaining lower interrupts are ignored. When the 68000 is reset, the initial mask is set to 111 so that the only interrupt that will be recognised is the non-maskable interrupt (NMI): it overrides any code in progress and cannot be turned off (masked off) by software.

Interrupt mask	Priority levels recognised	Priority levels ignored
111	7 (only NMI)	1-6
110	7	1-6
101	6-7	1-5
100	5-7	1-4
011	4-7	1-3
010	3-7	1-2
001	2-7	1
000	1-7 (all)	None

1.38 AmigaFlight® Help: 68000 Assembly Index

Motorola 68000 Assembly Language

Index

About the 68000 Chip

 Address Space
 Addressing Modes
 Chip Characteristics
 Chip Family
 Data Types
 The Stack
 Interrupt Handling

Addressing Modes

 Data Register Direct
 Address Register Direct
 Absolute Short
 Absolute Long
 Indexed Register Indirect with Offset
 Immediate
 Implied Register

Postincrement Register Indirect
Predecrement Register Indirect
Quick Immediate
Relative with Offset
Relative with Index Offset
Register Indirect
Register Indirect with Offset

Data Types

BCD Digits
Bits
Bytes
Long Words
Words

Instruction Types

Binary Coded Decimal Instructions
Bit Test and Manipulation Instructions
Data Movement Instructions
Flow Control Instructions
Integer Arithmetic Instructions
Logical Instructions
Shift and Rotate Instructions
System Control Instructions

Interrupts

Interrupt Handling
The Interrupt Mask

Stack

About The Stack
Manipulating The Stack

Status Flags

The Carry Flag
The Interrupt Mask
The Negative Flag
The Trace Flag
The Supervisor Mode Flag
The oVerflow Flag
The eXtend Flag
The Zero Flag

68000 Instruction Set

ABCD	Add Decimal with Extend
ADD	Add Binary
ADDA	Add Address
ADDI	Add Immediate
ADDQ	Add Quick
ADDX	Add Extended
AND	AND Logical
ANDI	AND Immediate
ASL	Arithmetic Shift Left
ASR	Arithmetic Shift Right
BCC	Branch if Carry Clear
BCS	Branch if Carry Set
BCLR	Test a Bit and Clear
BCHG	Test a Bit and Change
BEQ	Branch if Equal
BGE	Branch if Greater or Equal
BGT	Branch if Greater
BHI	Branch if High
BLE	Branch if Less or Equal
BLS	Branch if Low or Same
BLT	Branch if Less
BMI	Branch if Minus
BNE	Branch if Not Equal
BPL	Branch if Plus
BRA	Branch Always
BSR	Branch to Subroutine
BSET	Test a Bit and Set
BTST	Test a Bit
BVS	Branch if Overflow
BVC	Branch if No Overflow
CHK	Check register against bounds
CLR	Clear an Operand
CMP	Compare
CMPA	Compare Address
CMPI	Compare Immediate
CMPM	Compare Memory
DBT	No operation (condition always true)
DBF	Decr. and Branch Always unless Count = -1
DBHI	Decr. and Branch until High or Count = -1
DBLS	Decr. and Branch until Low or Same or Count = -1
DBCC	Decr. and Branch until Carry Clear or Count = -1
DBCS	Decr. and Branch until Carry Set or Count = -1
DBNE	Decr. and Branch until Not Equal or Count = -1
DBEQ	Decr. and Branch until Equal or Count = -1
DBVC	Decr. and Branch until No Overflow or Count = -1
DBVS	Decr. and Branch until Overflow or Count = -1
DBPL	Decr. and Branch until Plus or Count = -1
DBMI	Decr. and Branch until Minus or Count = -1
DBGE	Decr. and Branch until Greater or Equal or Count = -1

DBLT	Decr. and Branch until Less or Count = -1
DBGT	Decr. and Branch until Greater or Count = -1
DBLE	Decr. and Branch until Less or Equal or Count = -1
DBRA	Decr. and Branch Always unless Count = -1
DIVS	Signed Divide
DIVU	Unsigned Divide
EOR	Exclusive OR Logical
EORI	Exclusive OR Immediate
EXG	Exchange Registers
EXT	Sign Extend
ILLEGAL	Illegal Operation
JMP	Jump
JSR	Jump to Subroutine
LEA	Load Effective Address
LINK	Link and Allocate
LSL	Logical Shift Left
LSR	Logical Shift Right
MOVE	Move Data from Source to Destination
MOVE_CCR	Move to Condition Codes
MOVE_SR	Move from Status Register
MOVE_SR	Move to Status Register
MOVE_USP	Move User Stack Pointer
MOVEA	Move Address
MOVEM	Move Multiple Registers
MOVEP	Move Peripheral Data
MOVEQ	Move Quick
MULS	Signed Multiply
MULU	Unsigned Multiply
NBCD	Negate Decimal with Extend
NEG	Negate
NEGX	Negate with Extend
NOP	No Operation
NOT	Logical Complement
OR	Inclusive OR Logical
ORI	Inclusive OR Immediate
PEA	Push Effective Address
RESET	Reset External Devices
ROL	Rotate Left (without Extend)
ROR	Rotate Right (without Extend)
ROXL	Rotate Left with Extend
ROXR	Rotate Right with Extend
RTE	Return from Exception
RTR	Return and Restore Condition Codes
RTS	Return from Subroutine
SBCD	Subtract Decimal with Extend
SCC	Set if Carry Clear
SCS	Set if Carry Set

SEQ	Set if Equal
SF	Set Never
SGE	Set if Greater or Equal
SGT	Set if Greater
SHI	Set if High
SLE	Set if Less or Equal
SLS	Set if Lower or Same
SLT	Set if Less
SMI	Set if Minus
SNE	Set if Not Equal
SPL	Set if Plus
ST	Set Always
SVC	Set if No Overflow
SVS	Set if Overflow
STOP	Load Status Register and Stop
SUB	Subtract Binary
SUBA	Subtract Address
SUBI	Subtract Immediate
SUBQ	Subtract Quick
SUBX	Subtract with Extend
SWAP	Swap Data Register Halves
TAS	Test and Set and Operand
TRAP	Trap
TRAPV	Trap on Overflow
TST	Test an Operand
UNLK	Unlink
