SAMPLE PROGRAM LISTING:

```
dumbo 29: more labtest.s

        ORG       $1000       ; start at $1000

main    MOVE.W    #$00FFFE,D0 ; x:=y
        ADD.W     #$0001,D0   ; x:=x+1
        ADD.W     #$0001,D0
        ADD.W     #$00FFFE,D0 ; x:= x + $FFFE
        ADD.W     #$0002,D0   ; x:= x+ 2
        LEA.L     $6000,A0    ;
        LEA.L     adr,A2
        MOVEA.W   $2000,A1
        MOVE.W    D0,(A0)

        ORG       $6000
stuff   DC.W      $2000
adr     DC.W      $4000
        DC.W      $5000
        END       main        ; forces debug start at main
```

This program has two parts:
code — begins at $1000 and contains the program instructions
data — begins at $6000 and contains data and storage for program variables

The normal sequence of running a program is:

1. assemble the program
   The input file has a .s extension and the output file has a .o extension
2. link the program
   The input file(s) has a .o extension and the output file has a .x extension.
3. debug the program
   The input file has a .x extension. There is no output file.

ASSEMBLE THE PROGRAM:

```
dumbo 24: as68k -L demoprog.s > demoprog.lis
```

The input file has the extension .s. The assembler creates an output file demoprog.o for use by the linker and a list file demoprog.lis from the assembler. This list file shows the program, the assembled form of each instruction, memory locations used by the program, and all symbols defined in the program and their values. In particular, note the values of `stuff`, `main` and `adr` in the following listing.

```
dumbo 25: more demoprog.lis
Hewlett Packard AS68000 V(01.20 10Mar88) Page 1 Thu May 21 22:50:52

Cmdline - as68k -L demoprog.s
Line Address
1
2
3                                       ORG     $1000         ;start at $1000
4
5   00001000 303C FFFE          main   MOVE.W  #$00FFFE,D0  ;x:=y
6   00001004 5240                      ADD.W   #$0001,D0     ;x:=x+1
7   00001006 5240                      ADD.W   #$0001,D0
8   00001008 0640 FFFE                 ADD.W   #$00FFFE,D0  ;x:= x + $FFF2
10  0000100E 41F8 6000                 LEA.L   $6000,A0      ;
11  00001012 45F8 6002 4E71            LEA.L   adr,A2
12  00001018 3278 2000                 MOVEA.W $2000,A1
13  0000101C 3080                      MOVE.W  D0,(A0)
14
15                                      ORG     $6000
16  00006000 2000               stuff  DC.W    $2000
17  00006002 4000               adr    DC.W    $4000
18  00006004 5000                      DC.W    $5000
19                                      END     main          ;starts at main


                    Symbol Table

Label       Value

adr        00006002
main       00001000
stuff      00006000
```

NOTE: There is an important difference between the LEA commands in the above program. The instruction `LEA.L $6000,A0` puts the number $6000 into address register A0. The instruction `LEA.L adr,A2` puts the symbol "`adr`" (which has a value of $6002) into the instruction and has the effect of putting the number $6002 into A2. The `4E71` is a filler for the possibility that the linker would change the absolute value of "`adr`" from a word length $6002 to a different long word value.

LINK THE PROGRAM:

```
dumbo 26: ld68k -L -o demoprog.x demoprog >demoprog.llis
```

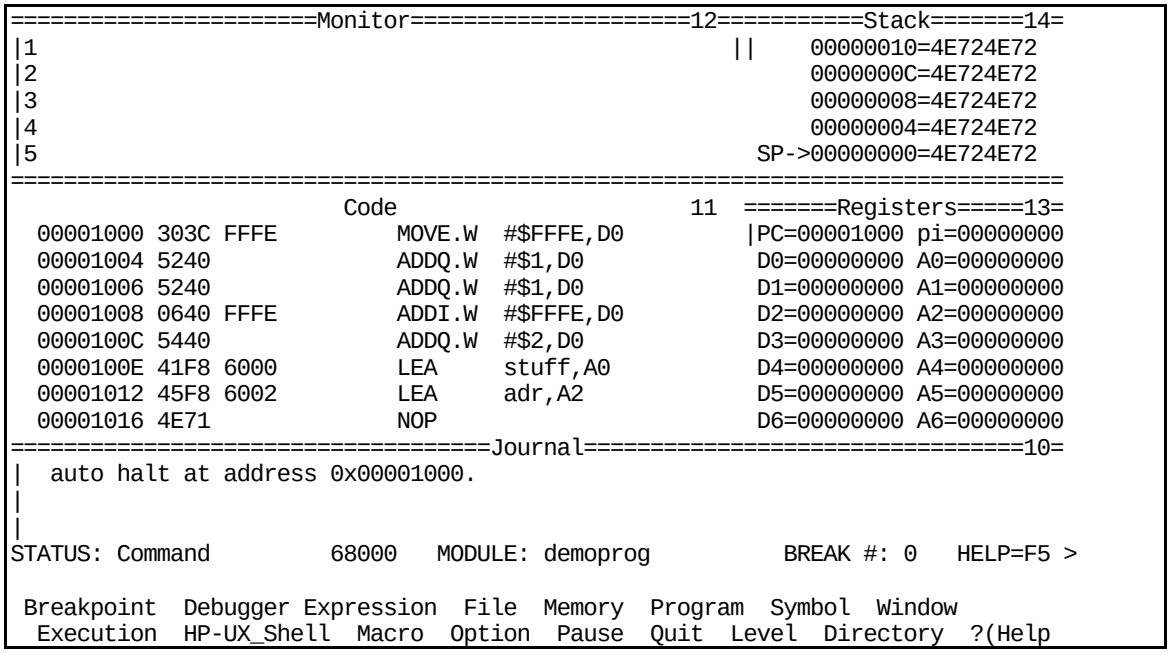
The input file is assumed to have the extension .o. The assembler creates an output file `demoprog.x` for use by the debugger and a list file `demoprog.llis` from the linker. This list file shows the location in memory of the program and it's components. In particular, note that there are two components of the program: one begins at address $1000 and the other begins at address $6000. These are the code and data components of the program as defined by the ORG statements in the program. A printout of `demoprog.llis` is shown below:

```
dumbo 27: more demoprog.llis
Hewlett-Packard LD68000 V(01.20 10Mar88)         Thu May 21 22:57:02

Command line: ld68k  -L -o demoprog.x demoprog

LOAD demoprog


OUTPUT MODULE NAME:    demoprog
OUTPUT MODULE FORMAT:  IEEE


SECTION SUMMARY
---------------

SECTION    ATTRIBUTE      START      END        LENGTH     ALIGN

           ABSOLUTE       00001000   0000101D   0000001E   0 (BYTE
           ABSOLUTE       00006000   00006005   00000006   0 (BYTE


MODULE SUMMARY
--------------

MODULE     SECTION:START     SECTION:END        FILE

demoprog          :00001000         :0000101D  /users/merat/eeap282/labs
                  :00006000         :00006005



START ADDRESS:   00001000

Load Completed
```

DEBUGGING/TESTING THE PROGRAM:

`dumbo 28: db68k demoprog`

It is assumed that the input file has the extension .x. The debugger uses this file which contains the program, memory assignments, and symbol definitions and values to simulate the operation of the program on a 68000 microprocessor.



The above screen is what you will see as a result of the db68k command. The monitor and stack areas of the display show information that will be important in future labs. The code area shows the next instruction to be executed and, approximately, the eight instructions following it. This code area displays your program in a format that is very similar to the assembler listing. Shown are the memory address (`00001000`), the assembled form of the instruction (`303C FFFE`), and the mnemonic form of the instruction (`MOVE.W #$FFFE,D0`). The register window shows the current contents of the data and address registers, the program counter (PC), and the status register (SR). When the debugger first starts not all registers and the SR are shown in the register area but will appear after the first instruction is executed. The PC was automatically set to $1000 by using the `END main` instruction in the program. If your program is not visible when you enter the debugger you will need to set the current value of the PC to the correct starting location of your program using the command

Memory **R**egister @PC=&&&&h

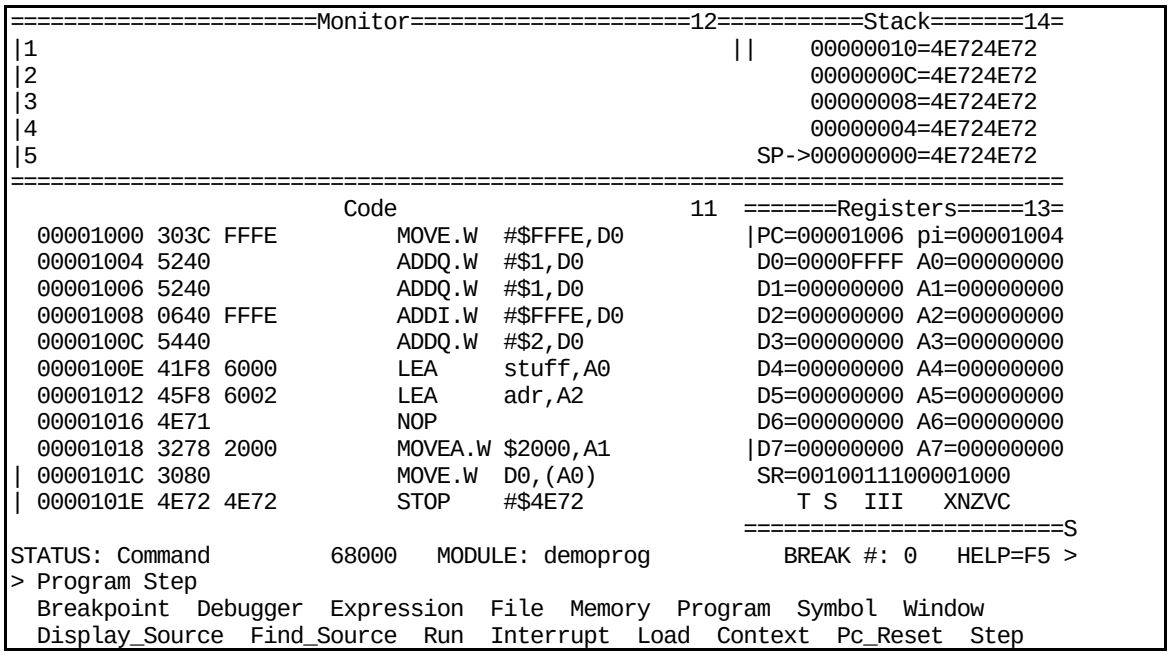
where &&&& is the hexadecimal address of the first instruction to be executed.

The MOVE instruction at $00001000 can be executed using the debugger command
Program **S**tep
which will execute the instruction pointed to by the PC (in this the MOVE instruction at $00001000), update all register and memory values affected by the instruction, advance the PC to the next instruction and display the new values as shown below. Note that the register window has been enlarged to show the remaining registers and the SR.

```
======================Monitor====================12==========Stack=======14=
|1                                                ||    00000010=4E724E72
|2                                                      0000000C=4E724E72
|3                                                      00000008=4E724E72
|4                                                      00000004=4E724E72
|5                                                   SP->00000000=4E724E72
================================================================================
                        Code                      11  =======Registers=====13=
  00001000 303C FFFE         MOVE.W  #$FFFE,D0         |PC=00001004 pi=00001000
  00001004 5240              ADDQ.W  #$1,D0             D0=0000FFFE A0=00000000
  00001006 5240              ADDQ.W  #$1,D0             D1=00000000 A1=00000000
  00001008 0640 FFFE         ADDI.W  #$FFFE,D0          D2=00000000 A2=00000000
  0000100C 5440              ADDQ.W  #$2,D0             D3=00000000 A3=00000000
  0000100E 41F8 6000         LEA     stuff,A0           D4=00000000 A4=00000000
  00001012 45F8 6002         LEA     adr,A2             D5=00000000 A5=00000000
  00001016 4E71              NOP                        D6=00000000 A6=00000000
  00001018 3278 2000         MOVEA.W $2000,A1          |D7=00000000 A7=00000000
| 0000101C 3080              MOVE.W  D0,(A0)            SR=0010011100001000
| 0000101E 4E72 4E72         STOP    #$4E72                T S  III   XNZVC
                                                       ======================S
STATUS: Command         68000   MODULE: demoprog          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

The X,N,Z,V and C bits of the SR are all updated as a result of the MOVE instruction. For a MOVE instruction the XNZVC bits will be updated according to _**00 (See the Programmer's Reference Manual). This means that the X bit doesn't change (indicated by _) from the previous value of zero, the N and Z bits are set according to the number being moved (as indicated by the *, in this case $FFFE is negative so the N bit becomes 1 and the Z bit remains 0), and the V and C bits are always set to 0 (as indicated by the 0) for a MOVE instruction.

You can continue to type
Program **S**tep
to execute the remaining program instructions one at a time. In this case the debugger will execute the ADD.W $0001,D0 instruction which adds a word length $0001 to the contents of D0 to get (D0)=$0000FFFF

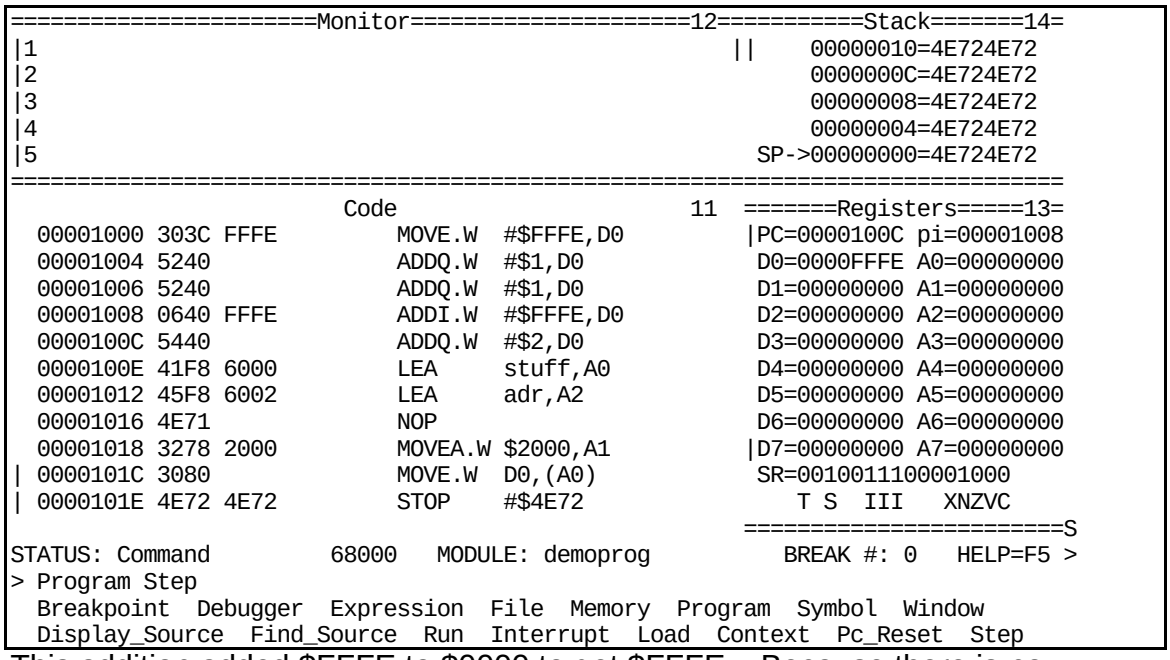


For an ADD instruction the XNZVC bits will be updated according to ***** (See the Programmer's Reference Manual). This means that these bits are set according to the results of the addition. The addition added $0001 to $FFFE to get $FFFF. There is no overflow or carry. The X , C and V bits are accordingly set to zero. Because the result is negative the N bit is set to 1. The result is not zero so the Z bit is set to 0.

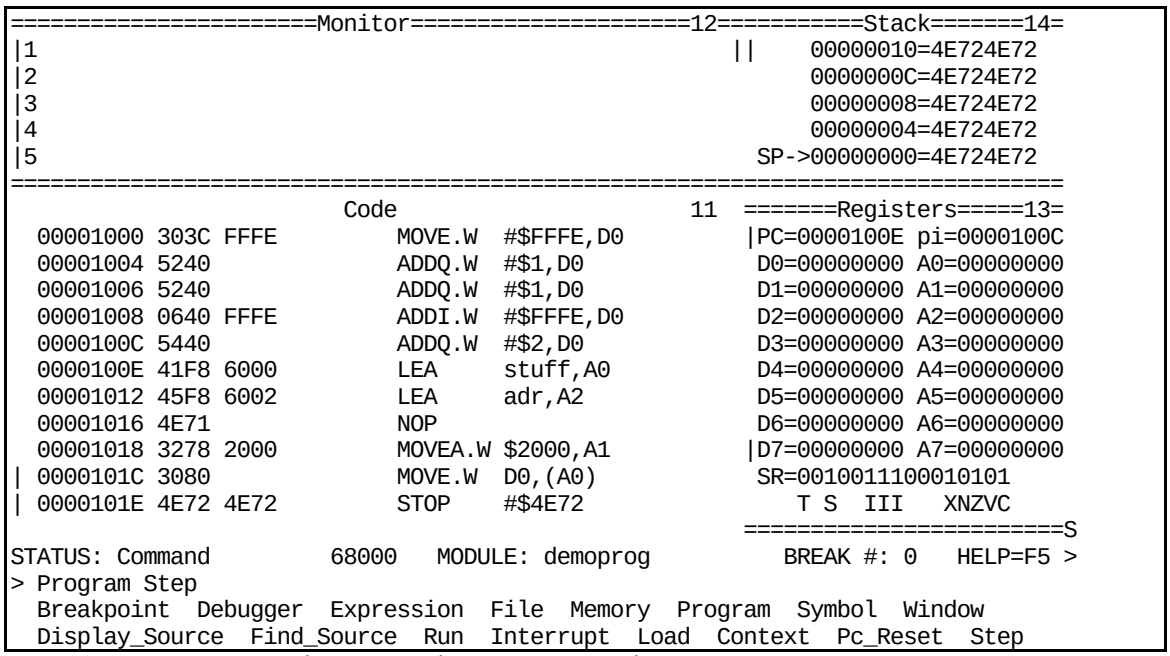
You can continue to type
Program **S**tep
to execute the remaining program instructions one at a time.

```
======================Monitor====================12==========Stack=======14=
|1                                                ||    00000010=4E724E72
|2                                                      0000000C=4E724E72
|3                                                      00000008=4E724E72
|4                                                      00000004=4E724E72
|5                                                   SP->00000000=4E724E72
================================================================================
                         Code                       11  =======Registers=====13=
  00001000 303C FFFE          MOVE.W  #$FFFE,D0          |PC=00001008 pi=00001006
  00001004 5240               ADDQ.W  #$1,D0              D0=00000000 A0=00000000
  00001006 5240               ADDQ.W  #$1,D0              D1=00000000 A1=00000000
  00001008 0640 FFFE          ADDI.W  #$FFFE,D0           D2=00000000 A2=00000000
  0000100C 5440               ADDQ.W  #$2,D0              D3=00000000 A3=00000000
  0000100E 41F8 6000          LEA     stuff,A0            D4=00000000 A4=00000000
  00001012 45F8 6002          LEA     adr,A2              D5=00000000 A5=00000000
  00001016 4E71               NOP                         D6=00000000 A6=00000000
  00001018 3278 2000          MOVEA.W $2000,A1           |D7=00000000 A7=00000000
| 0000101C 3080               MOVE.W  D0,(A0)             SR=0010011100010101
| 0000101E 4E72 4E72          STOP    #$4E72                 T S  III   XNZVC
                                                        ======================S
STATUS: Command          68000   MODULE: demoprog          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

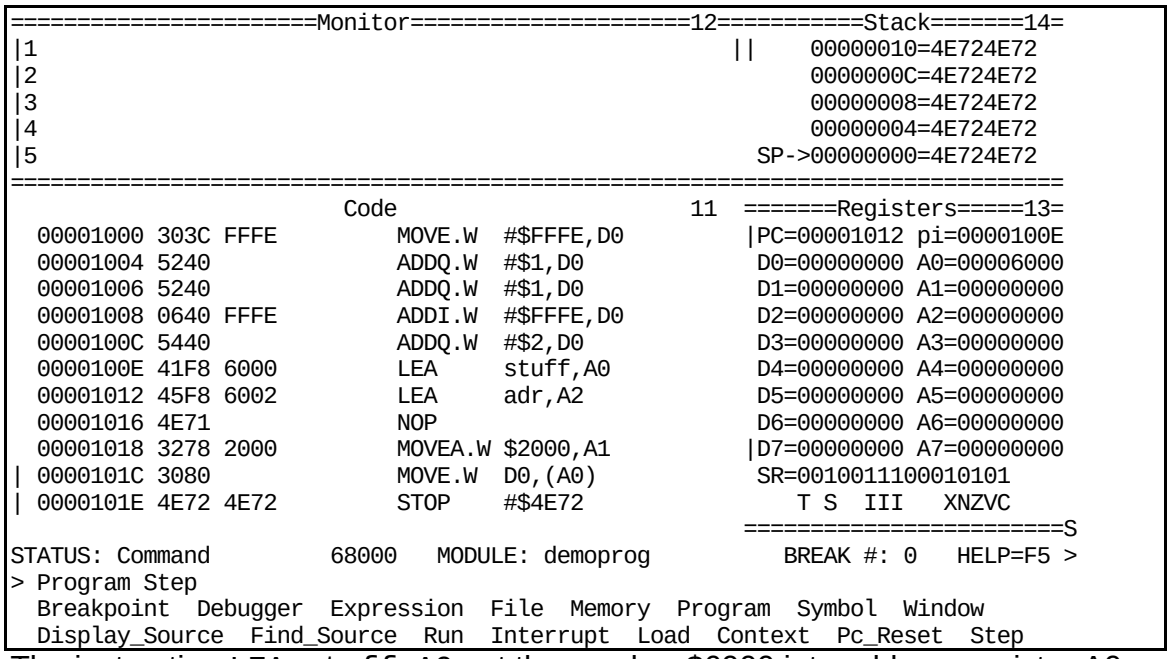
This addition added $0001 to $FFFF to get $10000. However, because the addition is word length the 1 cannot be placed in the lower word of D0 and a carry occurs. This sets the C and X bits to 1. Because the signs of the numbers are mixed no overflow occurs and V is set to 0. Because the result put into D0 is zero the Z bit is set to 1. The N bit is set to 0.



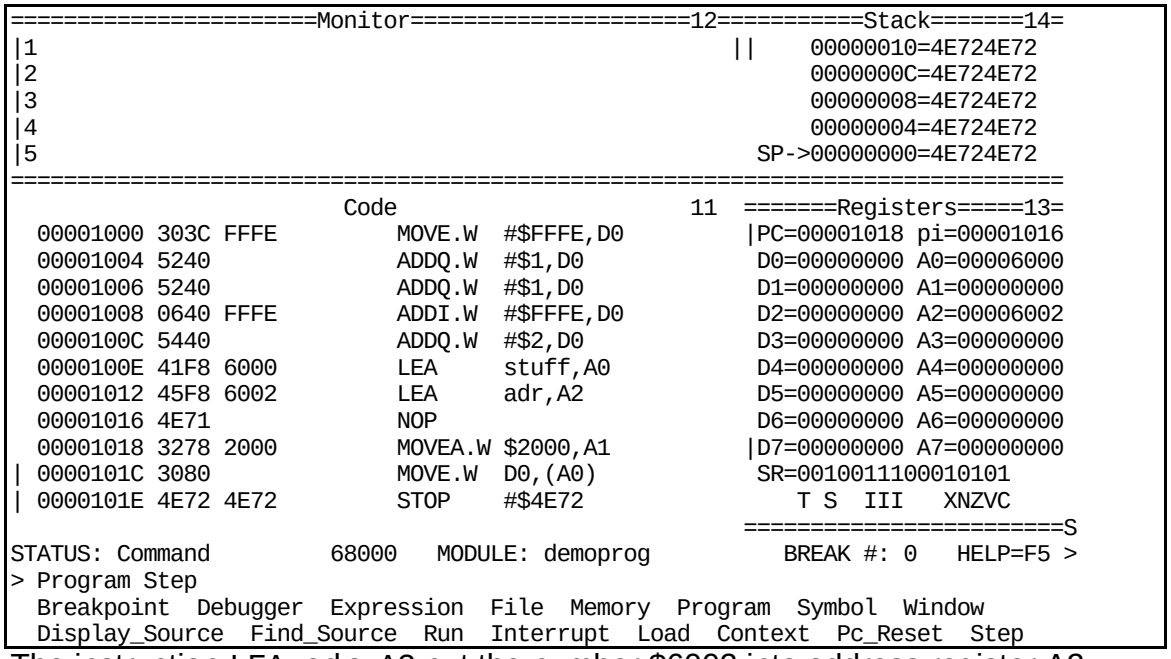
This addition added $FFFE to $0000 to get $FFFE. Because there is no overflow or carry the X , C and V bits are set to zero. Because the result is negative the N bit is set to 1. The number is not zero so the Z bit is set to 0.



This addition added $0002 to $FFFE to get $10000. Note that additions take place in 2's complement. Because $FFFE is -2, adding +2 to to -2 should result in zero and that is what happened. Because the addition is word length the 1 cannot be placed in the lower word of D0 and a carry occurs. This sets the C and X bits to 1. Because the signs of the numbers are mixed no overflow occurs and V is set to 0. Because the result put into D0 is zero the Z bit is set to 1. The N bit is set to 0.



The instruction LEA `stuff,A0` put the number $6000 into address register A0. Note that the original instruction in our program was `LEA.L $6000,A0.` The debugger recognized that $6000 was the value of the symbol stuff and automatically substituted it. The LEA instruction does not affect the SR so no SR values are changed.



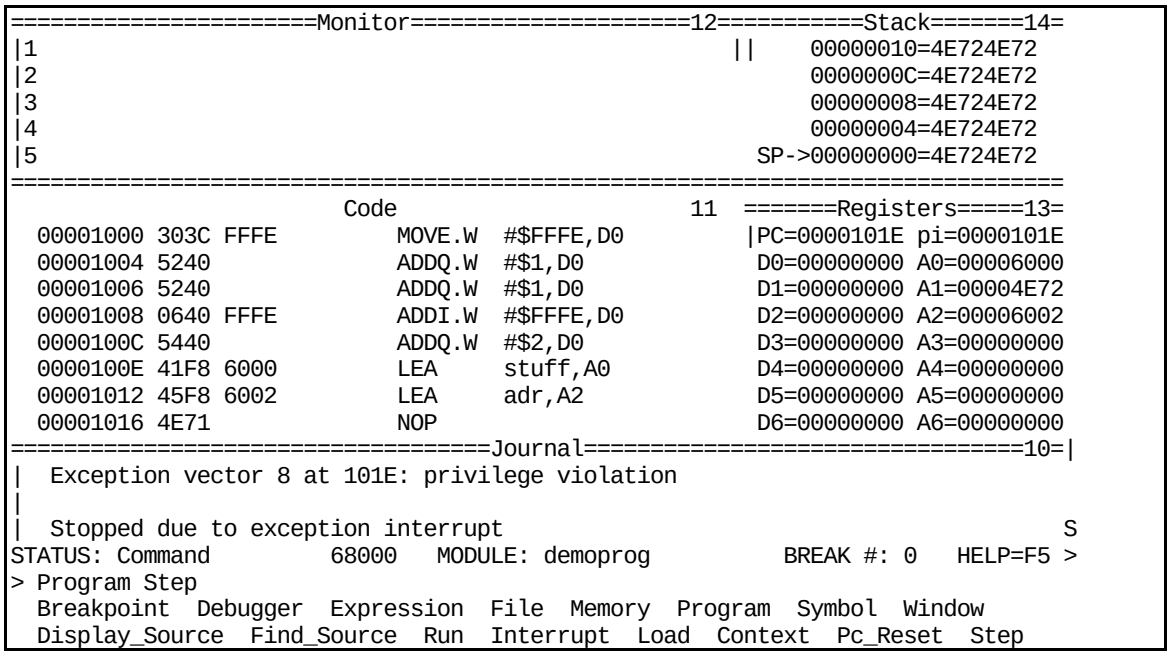
The instruction LEA `adr,A2` put the number $6002 into address register A2. Note that the PC automatically advances beyond the NOP (the $4E71) inserted by the assembler. No long word address was needed since our address $6000 is representable by a single word. The LEA instruction does not affect the SR so no SR values are changed.

```
======================Monitor====================12==========Stack=======14=
|1                                                 ||    00000010=4E724E72
|2                                                       0000000C=4E724E72
|3                                                       00000008=4E724E72
|4                                                       00000004=4E724E72
|5                                                    SP->00000000=4E724E72
================================================================================
                        Code                        11  =======Registers=====13=
  00001000 303C FFFE         MOVE.W  #$FFFE,D0         |PC=0000101C pi=00001018
  00001004 5240              ADDQ.W  #$1,D0             D0=00000000 A0=00006000
  00001006 5240              ADDQ.W  #$1,D0             D1=00000000 A1=00004E72
  00001008 0640 FFFE         ADDI.W  #$FFFE,D0          D2=00000000 A2=00006002
  0000100C 5440              ADDQ.W  #$2,D0             D3=00000000 A3=00000000
  0000100E 41F8 6000         LEA     stuff,A0           D4=00000000 A4=00000000
  00001012 45F8 6002         LEA     adr,A2             D5=00000000 A5=00000000
  00001016 4E71              NOP                        D6=00000000 A6=00000000
  00001018 3278 2000         MOVEA.W $2000,A1          |D7=00000000 A7=00000000
| 0000101C 3080              MOVE.W  D0,(A0)            SR=0010011100010101
| 0000101E 4E72 4E72         STOP    #$4E72                T S  III   XNZVC
                                                       ======================S
STATUS: Command         68000   MODULE: demoprog          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

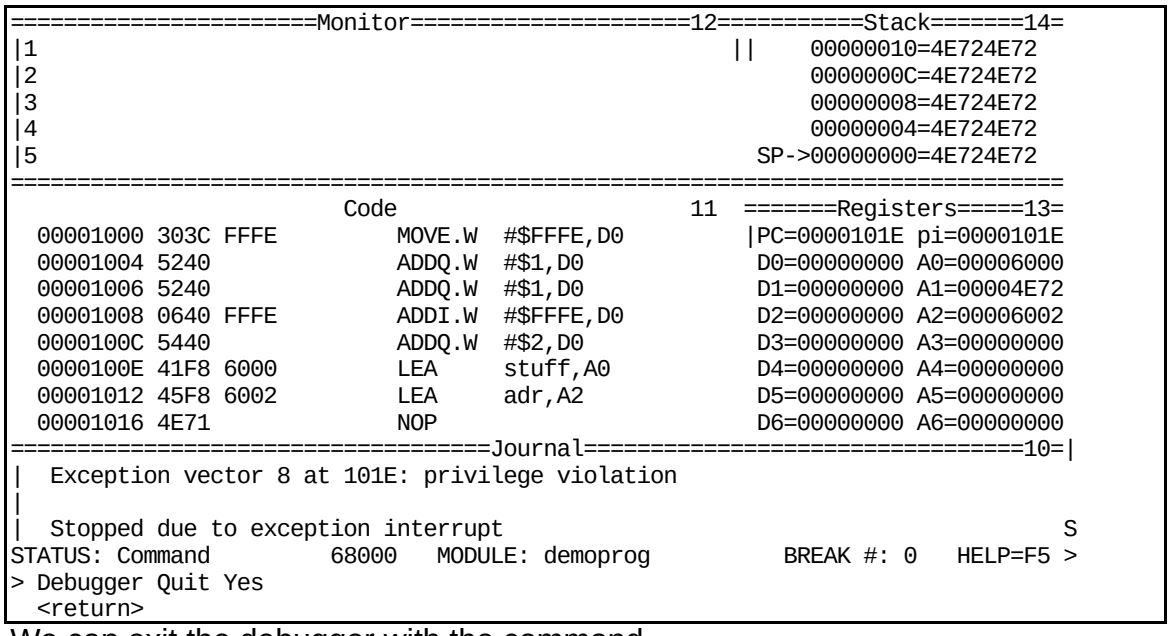
The instruction `MOVEA.W $2000,A1` put the contents of memory location $2000 into address register A1. Since we did not put anything into that particular memory location the debugger found the default value of $4E72 there. (The debugger initializes all memory to $4E72.) The MOVEA instruction does not affect the SR so no SR values are changed.

```
======================Monitor====================12==========Stack=======14=
|1                                                 ||    00000010=4E724E72
|2                                                       0000000C=4E724E72
|3                                                       00000008=4E724E72
|4                                                       00000004=4E724E72
|5                                                    SP->00000000=4E724E72
================================================================================
                        Code                        11  =======Registers=====13=
  00001000 303C FFFE         MOVE.W  #$FFFE,D0         |PC=0000101E pi=0000101C
  00001004 5240              ADDQ.W  #$1,D0             D0=00000000 A0=00006000
  00001006 5240              ADDQ.W  #$1,D0             D1=00000000 A1=00004E72
  00001008 0640 FFFE         ADDI.W  #$FFFE,D0          D2=00000000 A2=00006002
  0000100C 5440              ADDQ.W  #$2,D0             D3=00000000 A3=00000000
  0000100E 41F8 6000         LEA     stuff,A0           D4=00000000 A4=00000000
  00001012 45F8 6002         LEA     adr,A2             D5=00000000 A5=00000000
  00001016 4E71              NOP                        D6=00000000 A6=00000000
  00001018 3278 2000         MOVEA.W $2000,A1          |D7=00000000 A7=00000000
| 0000101C 3080              MOVE.W  D0,(A0)            SR=0010011100010100
| 0000101E 4E72 4E72         STOP    #$4E72                T S  III   XNZVC
                                                       ======================S
STATUS: Command         68000   MODULE: demoprog          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

The instruction `MOVE.W D0,(A0)` puts the contents of D0 (in this case, $0000) into the memory location whose address is in A0 (in this case, $6000). Because the number is zero the Z bit is set to 1. The N, V and C bits are set to 0. The X bit remains unchanged.



When we attempt to execute the next instruction we get a error. This is because our program did not extend to this memory location and it contained the initial value of $4E72 put there by the debugger. There is no elegant way to end your program in the debugger (that we have shown you yet).



We can exit the debugger with the command

Debugger **Q**uit **Y**es

ADDRESSING MODES

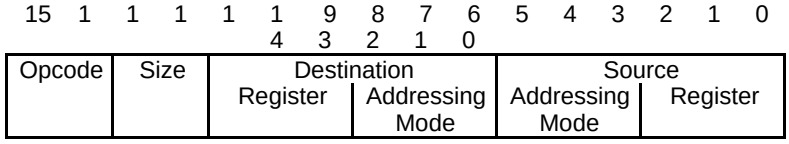
An addressing mode tells the computer where to get/place a number.

Basic form of a MC68000 instruction:

Instruction Source, Destination

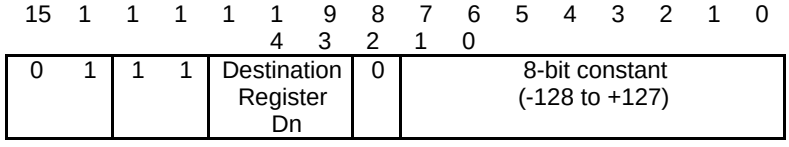
The source and destination can use DIFFERENT addressing modes.

Addressing modes on the MC68000 are usually specified in the first 12 bits of the 68000 instruction word:



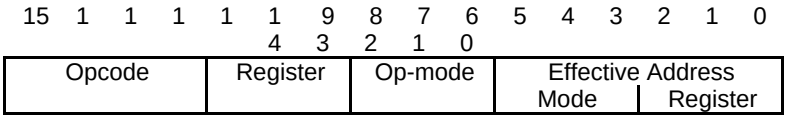
A few specialized instructions make assumptions about the addressing modes and use different formats. For example, MOVEQ assumes that the source is an 8-bit immediate constant and the destination is a data register.

MOVEQ

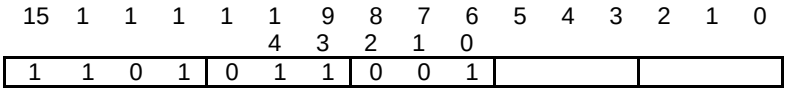


For the following examples we will consider the different source addressing modes for a word length ADD instruction which will put the resulting word into data register D3.

The general form of the ADD instruction:



The general form of the ADD instruction which adds the contents of D3 to something and puts the results into D3 is:



where

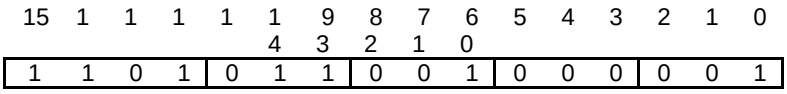
- bits 15-12 indicate the op code 1101 for an ADD
- bits 11-9 indicate that data register D3 (i.e. 3) is the destination for the result of the ADD ($3_{10}=011_2$)
- bits 8-6 indicate the op-mode of the ADD. In this case the calculation will be a word length ADD of the form (<Dn>)+(<ea>) → <Dn>. This is indicated by 001; see the Programmer's Reference Manual for information about the other modes. [NOTE: This translates into add the contents of D3 to the contents of the effective address <ea> and put the result into D3.]
- bits 5-3 indicate the addressing mode of the source
- bits 2-0 indicate the register (if applicable) of the source

We will examine bits 5-0 in detail in the following examples.

Source is data register direct (mode=000, register=register#)

general form ADD D1,D3 ← Both source and destination are data register direct

Assembled instruction:

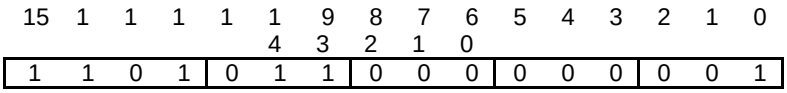


where mode=000_2 to indicate a data register and register=$1_{10}=001_2$ to indicate register D1.

Address register direct (mode=001, register=register#)

general form ADD A1,D3 ← Source is address register direct; destination is data register direct

Assembled instruction:



where mode=001_2 to indicate an address register and register=$1_{10}=001_2$ to indicate register A1.

Immediate
(mode=000, register=100)
Although this looks like an ordinary ADD instruction

general form ADD #7,D3 ← Source is immediate; destination is data register direct

this instruction is always re-coded by the assembler into the more specific ADDI (ADD immediate) instruction format

general form ADDI #7,D3 ← Source is immediate; destination is data register direct

which has the instruction format:

15	1	1	1	1	14	13	12	11 10	9 8 7 6	5 4 3	2 1 0
0	0	0	0	0	1	1	0	Size	Effective Address		
									Mode	Register	
Word Data (16 bits)								Byte Data (8 bits)			
Long Data (32 bits, including previous word)											

where the immediate constant is stored in one or two extension words according to the above conventions.

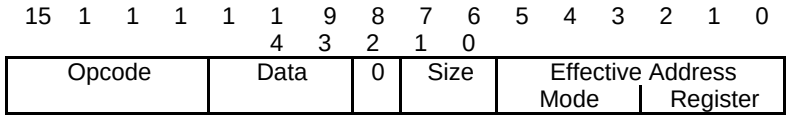
In this particular case the assembled instruction will become:

15	1	1	1	1	14	13	12	11	10	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

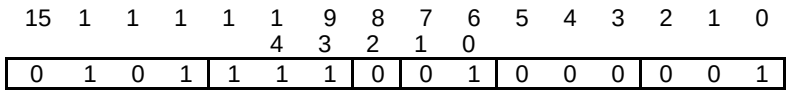
where the constant is stored in one extension word. Note that it is the instruction (default is .W) which determines the mode, NOT the size of the constant.

NOTE: As shown the instruction ADDI #7,D3 takes two words of computer memory. This instruction could be shortened to one word using the ADDQ (ADD quick) instruction. A quick instruction uses a special instruction word format which can include constants represented as 3-bit binary numbers (the constants are limited to the range 1 to 8 decimal)

general format:



Assembled instruction:



where the size = 01_2 to indicate a word length operation and the effective address is that of the source. In this case it is mode=000_2 to indicate a data register and register=011_2 to indicate D3.

Note that if the immediate constant was 1234567810 that the ADDQ instruction could NOT be used. The ADDI with two (2) extension words would be used because the binary constant is so large.

Different methods of specifying immediate constants:

```
ADD.W   #$452,D3      ;specifies a hexadecimal
                       constant
ADD.L   #I,D3         ;specifies the address of I
                       as a constant
ADDI.W  #%1011,D3     ;specifies a binary
                       constant
```

IMMEDIATE ADDRESSING

#xxx indicates immediate addressing in the Programmer's Reference Manual

Immediate mode addressing can only be used for source addressing; it is NOT allowed for destination addressing.

examples of immediate mode addressing:

```
MOVE.W    #$452,D0      ;moves the number
                        $452 to D0
MOVE.L    #I,D0         ;moves the value of I
                        (the address in the
                        symbol table) into D0
                        as a number
ADDI.W    #%1011,D0     ;add the binary
                        constant 1011 (13₁₀)
                        to the contents of D0
MOVE.L    I,D0          ;move the contents of
                        the memory location I
                        into D0.  NOT
                        IMMEDIATE MODE
                        ADDRESSING.
;Yes, you can do calculations with labels
```

ABSOLUTE MODES OF ADDRESSING

absolute

xxx	indicates absolute short addressing in the Programmer's Reference Manual
xxxxxx	indicates absolute long addressing in the Programmer's Reference Manual

Whether an absolute addressing mode is long or short is usually assigned by the assembler.

examples of absolute long addressing:

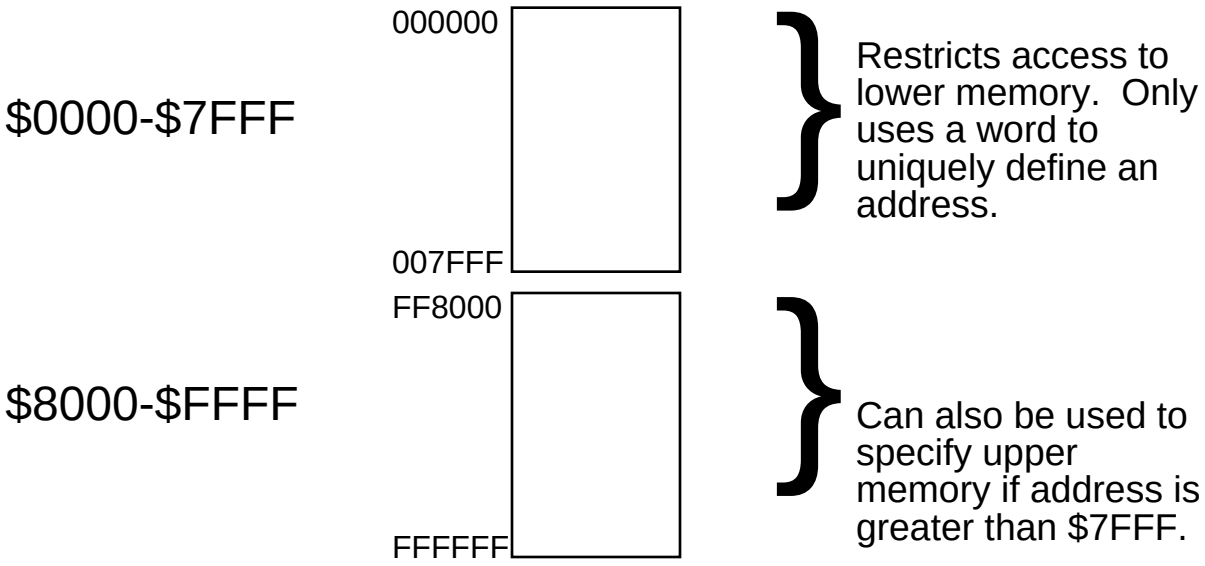
```
MOVE.W    I,D0          ;moves the contents
                        of address (longword)
                        I to D0
ADD.W     $500,D0       ;add the contents of
                        $0000 0500 to D0
MOVE.W    D0,I-4        ;move the contents of
                        D0 to (long word)
                        address I-4
;Yes, you can do calculations with labels
```

examples of absolute short addressing:
(usually only the assembler does this)

```
MOVE.W    $1000.W,D0    ;address is sign
                        extended to 24 bits
                        and stored in a single
                        extension word
ADD.L     I.W,D0        ;takes only the lower
                        16 bits of I to form the
                        extension word
```

Absolute long addressing is generated by default.
Absolute short addressing is usually generated by the assembler rather than directly by the programmer. The assembler does this to generate shorter code but it restricts the memory range you can access.

As absolute short uses a 16-bit extension word it can only be used to access memory at the bottom of memory or at the top of memory as shown below. Absolute short cannot be used to access the memory in between.



```
        ADD.L   I,D3    ;not immediate, specifies moving
                        the contents of I (I is a label)
I       DC.L    75      ;I specifies a memory address,
                        the DC.L instruction instructs the
                        assembler to reserve 75
                        longwords in memory beginning
                        at this address
```

absolute long
(mode=111, register=001)

The primary difference between Absolute Short and Absolute Long is that the address is bigger than 16-bits and must be represented as a full 32-bit address.

general form ADD LABEL,D3 ← Source is absolute (an address specified by a symbol);

The assembled form of an absolute long ADD instruction is:

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

where xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx is a 32-bit binary address indicating the value of LABEL and is used as the address of the data to be used in the ADD. Compared to immediate mode, this is the address of the data.

68000 ADDRESSING MODES

addressing mode	mode	register	length of instruction	generation	assembler syntax	example
data register direct	000	Dn	1	EA=Dn	Dn	ADD D0,D3
address register direct	001	An	1	EA=An	An	ADD A7,D3
address register indirect	010	An	1	EA=(An)	(An)	ADD (A0),D3
address register indirect with post increment	011	An	1	EA=(An) An¨An+N N=1 byte N=2 words N=3 longwords	(An)+	ADD (A0)+,D3
address register indirect with predecrement	100	An	1	EA=(An) An¨An-N N=1 byte N=2 words N=3 longwords	-(An)	ADD -(A0),D3
address register indirect plus (16 bit) displacement	101	An	2	EA=(An)+d_{16}	d_{16}(An)	ADD 4(A0),D3 ADD LABEL(A0),D3
address register indirect plus index (data or address register) plus (8 bit) displacement	110	An	2	EA=(An)+(Xn)+d_8	d_8(An,Xn)	ADD 2(A0,D6.W),D3 ADD LABEL(A0,A6.W),D3
absolute short (16 bits)	111	000	2	EA=(next word)	xxx	ADD LABEL,D3
absolute long (24/32 bits)	111	001	3	EA=(next two words)	xxxxxx xxx.L	ADD HIGHAD,D3
						PC relative after RORG
PC+16 bit displacement	111	010	2	EA=(PC)+d_{16}	d_{16}(PC)	ADD LABEL,D3
PC+index (data or address register)+displacement (8 bit)	111	011	2	EA=(PC)+(Xn)+d_8	d_8(PC+Xn)	ADD 4(A0),D3
immediate	111	100	2 or 3	Data=Next Word(s)	#xxx	ADD #9,D3 ADD.L #$123456,D3

NOTE: Addresses are ALWAYS handled differently than data.

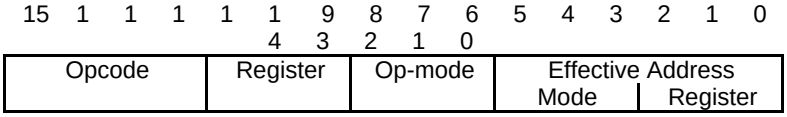
absolute short
(mode=111, register=000)

As shown above this addressing mode LOOKS like immediate but is NOT because no # precedes the label.

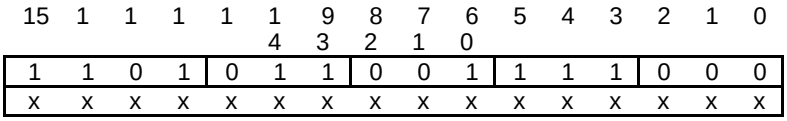
general form ADD LABEL,D3 ← Source is absolute (an address specified by a symbol); destination is data register direct

The source is a memory location. Specifically, LABEL will have an assigned value in the symbol table and this value will be used as the address of the source data. Unlike the immediate mode, this mode uses the standard ADD instruction format.

Recall the general form of the ADD instruction:



The assembled form for absolute short source addressing is:



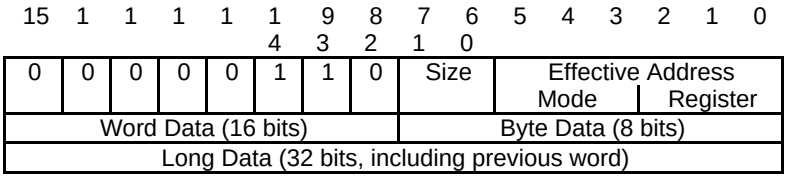
where xxxxxxxxxxxxxxxx is be a 16-bit extension word containing the value of label and is used as the address of the data to be used in the ADD. Compared to immediate mode, this is the address of the data. Immediate mode would put the data in the extension word.

IMPORTANT

general form ADD #LABEL,D3 ← The source is NOT absolute, it is IMMEDIATE.

THIS IS THE #1 STUDENT ERROR.

The correct interpretation of this instruction is:



where Word Data or Long Data is the value of LABEL interpreted as a constant. For example, if LABEL had the value $7EFF the above instruction would add the value $7EFF to the contents of D3. IT WOULD NOT ADD THE CONTENTS OF MEMORY ADDRESS $7EFF to D3.

ADDRESSING MODES WHICH INVOLVE ADDRESS REGISTERS

Only a restricted set of MC68000 instructions permit an address register destination operand. These instructions usually have an instruction mnemonic that ends in A, i.e. ADDA, MOVEA, SUBA, etc.

MOVE.size <ea>,<ea>

As a destination <ea> can be a data register or a memory location. An address register (An) destination is NOT allowed.

MOVEA.size <ea>,An

The destination for a MOVEA instruction can only be an address register.

The MC68000 handles address calculations in a different manner than simple math calculations:

- if size=word then the source word is sign-extended prior to the calculation
- only word or long word sizes are allowed

Examples of Using Symbols and MOVEA:

```
            ORG         $6000
FT    DS.L          1

      ORG           $7000
      MOVEA.L       FT,A0       ;moves 32-bit long word
                                  at $6000 into A0, i.e.
                                  (A0)=$1
      MOVEA.L       #FT,A1      ;moves 32-bit long word
                                  address $6000 into A0,
                                  i.e. (A0)=$00006000
```

The point is that FT=$6000 in the symbol table. If you refer to FT it simply replaces the symbol with the hex number $6000. Hence, the first instruction is really

```
MOVEA.L     $6000,A0
```

which you know will put the contents of address $6000 into A0. The second instruction is treated as

```
MOVEA.L     #$6000,A1
```

which you know will put the number $6000 into A1.

MORE ADDRESS REGISTER SPECIFIC INSTRUCTIONS:

MOVEA.L <ea>,An
The MOVEA instruction moves the contents of the source operand, i.e. the contents of <ea>, to the address register An.

LEA<ea>,An
The LEA instruction simply moves the source operand, i.e. <ea>, to the address register. At this point in your knowledge of MC68000 addressing you can think of the LEA instruction treating the <ea> as an immediate constant.

Example:

```
        ORG     $6000
TE      DC      $ABCD

        ORG     $7000
        MOVEA.L TE,A0       ;will put $ABCD into A0
        MOVEA.L #TE,A1      ;will put $6000 into A1
        LEA     TE,A0       ;will put $6000 into A0
        LEA     #TE,A0      ;NOT ALLOWED
        LEA     16(TE),A2   ;can compute addresses
                            as will be shown later in
                            course
```

COMMENTS:
1. Use MOVEA to initialize address registers
2. Use LEA to calculate dynamic addresses such as found in arrays.

USING ADDA AND SUBA

These instructions are used to manipulate addresses

```
ADDA.<size>     <ea>,An
SUBA.<size><ea>,An
```

where <size> can only be word or long word. As shown <ea> can be determined by any addressing mode but the destination can only be an address register. If <size> is word, then the source is sign extended to a long word and all calculations and the result are long word.

Examples:

```
ADDA.L      #100,A0
```

The source is immediate. The destination is address register direct as it should be. This instruction adds long word $64 ($64=100_{10}) to the long word contents of A0.

```
SUBA.W      ALPHA,A1
```

Since ALPHA is a label the source is absolute long. The destination is address register direct as it should be. This instruction adds the sign-extended (ALPHA) to the long word contents of A1.

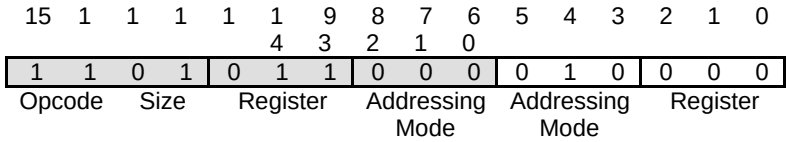
Suppose (ALPHA) = $8C07 and (A1)=$0000 A04B. Then, sign-extending (ALPHA) to $FFFF 8C07 and adding $FFFF 8C07 to $0000 A04B gives (A0) = $0001 1444. If you did not sign-extend you would get the incorrect result (A0) = $0000 1444.

ADDRESS REGISTER INDIRECT ADDRESSING MODES

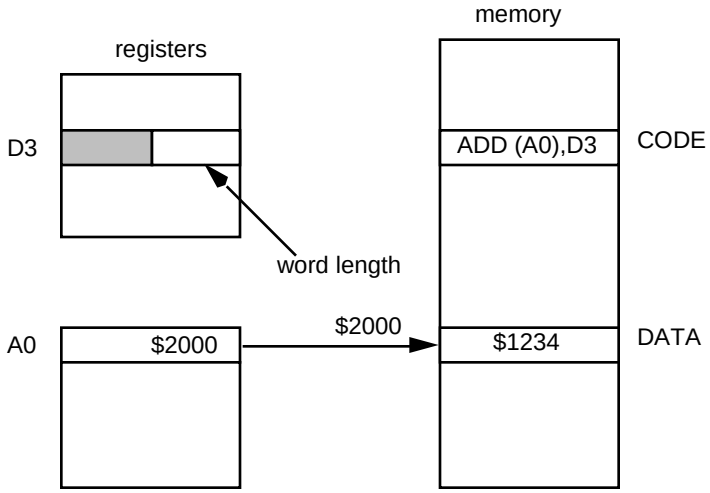
Address register indirect
(mode=010, register=register#)

general form ADD (A0),D3 ← Source is address register indirect

Assembled instruction:



where mode=010_2 to indicate address register indirect and register=$0_{10}=000_2$ to indicate register A0.



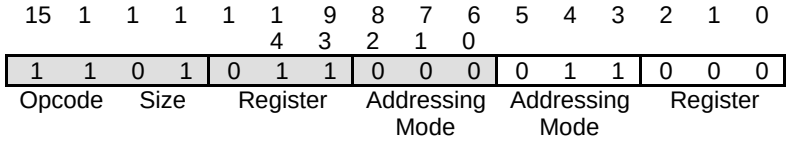
This instruction references the contents of the memory location whose address is in A0 adds $1234 to the low 16-bit word contents of D3.

This type of addressing is indicated in the Programmer's Reference Manual by EA=(An)

Address register indirect with **post**increment
(mode=011, register=register#)

general form ADD (A0)+,D3 ← Source is address register indirect with postincrement

Assembled instruction:



This instruction performs the same ADD but after the ADD is performed will increment the word length contents of A0 by one word (2 bytes). In the previous example this would change the address in A0 from $2000 to $2002

This is indicated in the Programmer's Reference Manual by the notation:

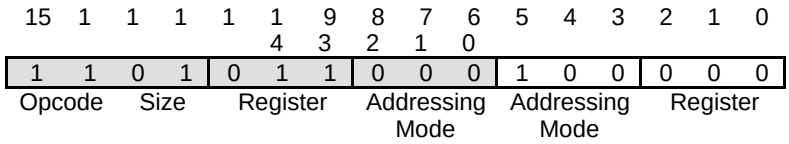
EA=(An)
An←An+N

where N is determined by the instruction size.
N=1 for byte length adds, 2 for word length adds, and 4 for long word length adds.

Address register indirect with **pre**decrement
(mode=100, register=register#)

general form ADD -(A0),D3 ← Source is address register indirect with predecrement

Assembled instruction:



This instruction performs an ADD but before the ADD is performed the word length contents of A0 are decremented by one word (2 bytes). In the previous example this changes the address in A0 from $2000 to $1FFE and does a word length add of the contents of $1FFE to D3.

The manipulation of the source is indicated in the Programmer's Reference Manual by the notation:

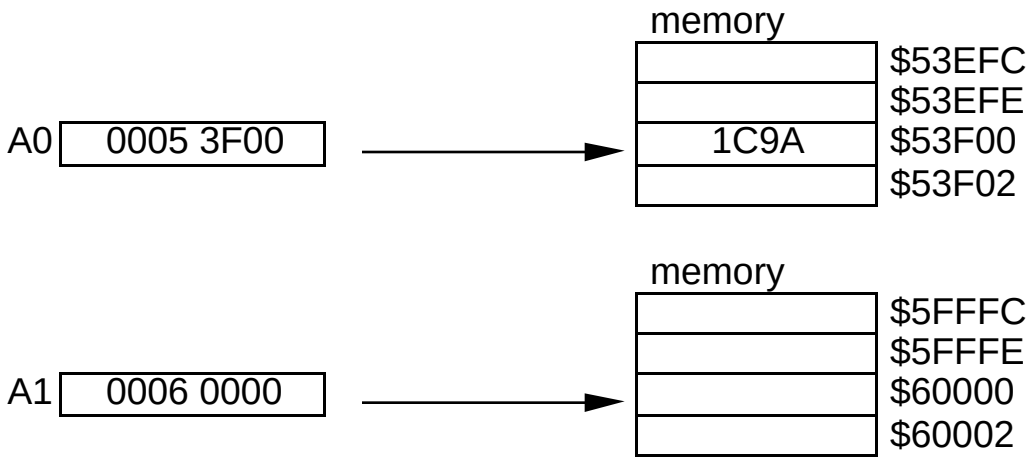
An←An-N
EA=(An)

where N is determined by the instruction size.
N=1 for byte length adds, 2 for word length adds, and 4 for long word length adds. Note that the subtraction is shown before the effective address to indicate that that contents of the address register are decremented and then used to determine the source address.

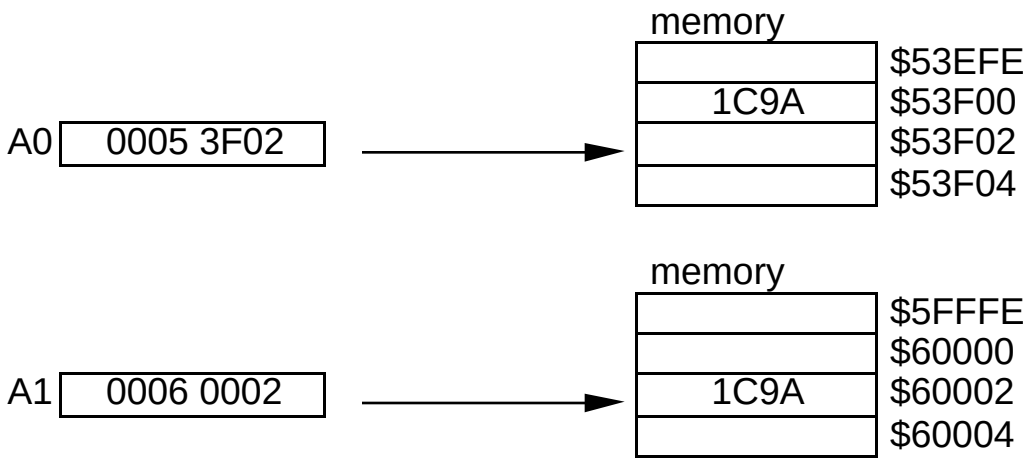
interesting forms of combining indirect addressing:

MOVE.W (A0)+,(A1)+

Before:



After:



These modes are good for moving blocks of data from one area of memory to another.

More interesting instructions:

MOVE.W (A0)+,(A1)+
Copies one data word from (A0) to (A1)

1. reads word contents of (A0)
2. increments A0 by 2 bytes
3. moves word to (A1)
4. increments A1 by two bytes

MOVE.W -(A0),-(A1)
Copies one data word from (A0) to (A1)

1. decrements A0 by 2 bytes
2. reads word at new (A0)
3. decrements A1 by 2 bytes
4. writes word at new (A1)

MOVE.W -(A0),-(A0)
Copies one data word from (A0-2) to (A0-4)

1. decrements A0 by 2 bytes
2. reads word at new (A0)
3. decrements A0 by 2 bytes
4. writes word to new (A0), i.e. original A0 - 4 bytes

MOVE.L (A2)+,(A2)
Copies one long word from (A2) to (A2+4)

1. read long word at (A2)
2. increments A2 by 4 bytes
3. writes long word to new (A2)

MOVE (A7)+,(A7)
Copies the word at (A7) to (A7+2)

The above MOVE instructions only affect memory; no data registers are affected.

ADD.W (A0)+,D0
Adds word at (A0) to D0

1. reads word at (A0)
2. increments A0 by 2 bytes
3. adds word to D0

MOVE.W -(A2),D0
Adds word at (A2-2) to D0

1. decrements A2 by 2 bytes
2. read word at new (A2)
3. writes word to D0

MOVE.W D1,-(A1)
Moves a word from D1 to the new (A1) after A1 is decremented

MOVE.W (A1)+,D2
Moves (A1) to D2; A1 is incremented by 2 bytes

A simple rule to remember for computing addresses is to evaluate the expression from left to right.

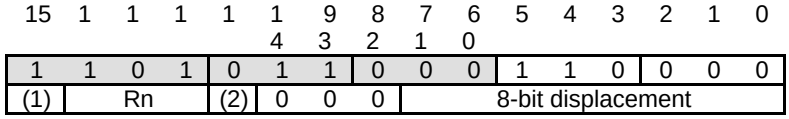
Address register indirect with index and 8-bit displacement*
(mode=110, register=register#)

*also called offset by Motorola

Examples

```
ADD    4(A0,D6),D3        ← uses A0 as the
                             base address
ADD    LABEL(A0,D6),D3    ← uses the symbol
                             LABEL as the
                             base address
```

Assembled format of instruction:

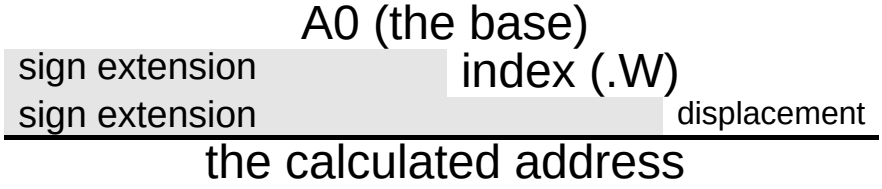


Notes:
Rn index register number, 0≤Rn≤7
(1) type of index register, 0=data register, 1=address register
(2) size of index register for address computation, 0=word length, 1=long word length
8-bit displacement is 2's complement number

The source address is computed as the sum of the base address (the contents of A0, in these examples), the displacement, and the offset. The addition is performed using all 32-bit numbers and the result is a 32-bit address. The index (either an address or data register) is either a 32-bit number or a 16-bit number sign extended to 32 bit. The displacement is an 8-bit number in 2's complement notation sign extended to 32 bits for the address calculation.

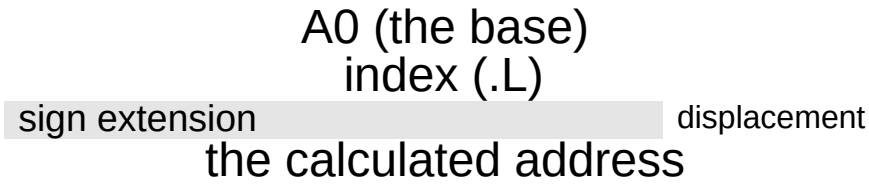
```
ADD    4(A0,D6.W),D3      case 1
ADD    4(A0,D6.L),D3      case 2
```

case 1:



In this case the word-length index register is sign extended to a long word, and the byte-length displacement is extended to a long word. The actual address calculation is performed using all long word numbers.

case 2



In this case the index register is long word needing no sign extension, and the byte-length displacement is extended to a long word. The actual address calculation is again performed using all long word numbers.

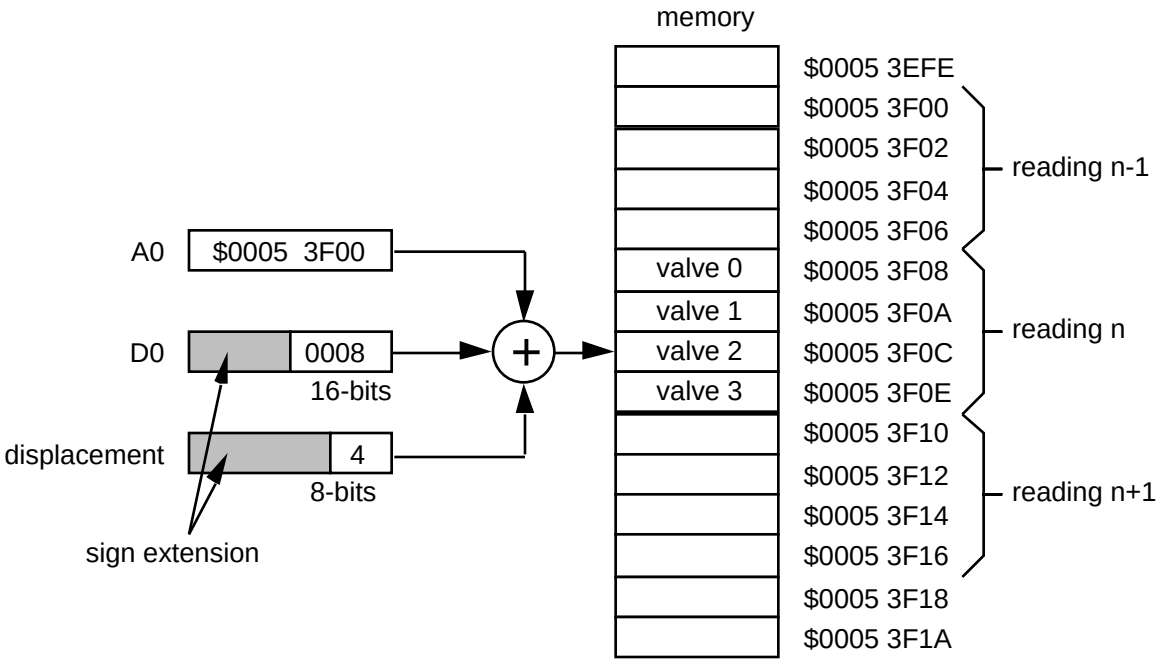
Example:
Using address register indirect addressing to access a table of numbers.

A MC68000-based system monitors four pressure valves in a chemical processing plant. Each valve's pressure is recorded every half-hour. These readings are sequentially stored in memory as shown below.

MOVE.W VALVE(A0,D0.W),D1

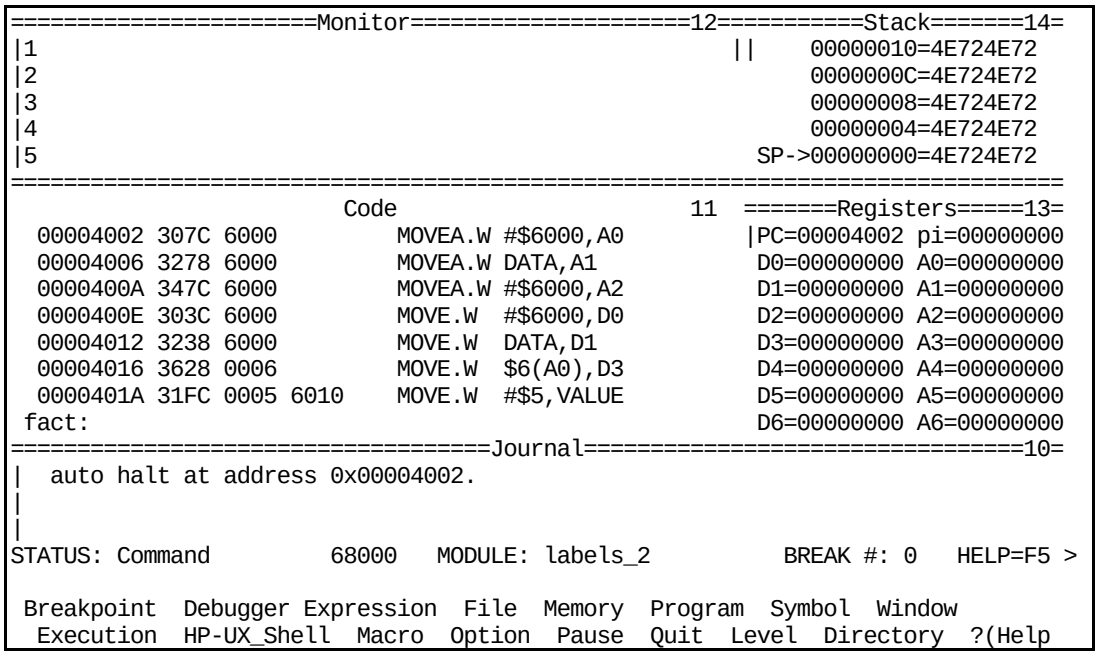
retrieves any selected valve reading into D1 where:

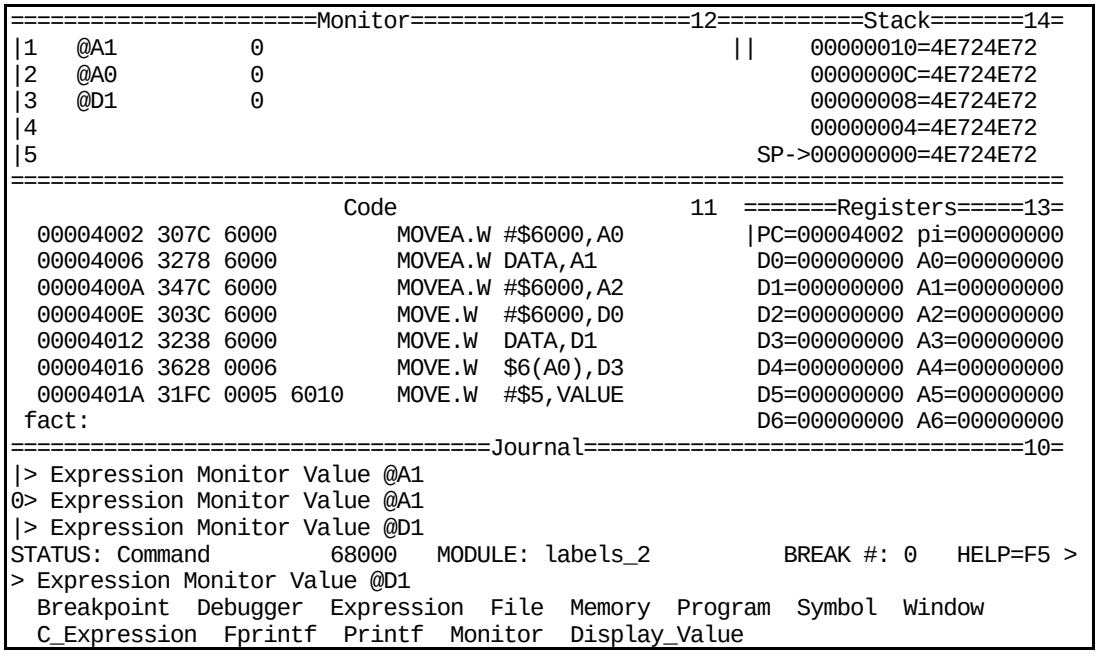
A0	contains the beginning address of the table, in this case $53F00
D0.W	contains the number of the reading, in this case n=8
VALVE	is the reference to a particular valve. In this case VALVE=4.



EXAMPLE PROGRAM USING LABELS

```
Hewlett Packard AS68000 V(01.20 10Mar88) Page 1 Mon Oct 29 13:46:500

Cmdline - as68k -L labels.s
Line Address
1   00006000                DATA     EQU     $6000
2   00004000                PROGRAM  EQU     $4000
3   00000006                FFSET    EQU     $6
4
5                                    ORG     DATA
6                           * TABLE OF FACTORIALS
7   00006000 0001           FTABLE   DC      1           0!=1
8   00006002 0001                    DC      1           1!=1
9   00006004 0002           test     DC      2           2!=2
10  00006006 0006                    DC      6           3!=6
11  00006008 0018                    DC      24          4!=24
12  0000600A 0078                    DC      120         5!=120  ($78)
13  0000600C 02D0                    DC      720         6!=720
14  0000600E 13B0                    DC      5040        7!=5040
15  00006010                VALUE    DS.B    1           input to factorial
16  00006011                         DS.B    1           align on word boundary
17  00006012                RESULT   DS.W    1           result of factorial
18
19
20                                   ORG     PROGRAM
21  00004000 4E71                    NOP
22  00004002 307C 6000      main     MOVEA.W #FTABLE,A0  gets $6000
23  00004006 3278 6000               MOVEA.W FTABLE,A1   gets $1
24  0000400A 347C 6000               MOVE.W  #FTABLE,A2  gets $6000
25  0000400E 303C 6000               MOVE.W  #FTABLE,D0  gets $6000
26  00004012 3238 6000               MOVE.W  FTABLE,D1   gets $1
27
28  00004016 3628 0006               MOVE.W  FFSET(A0),D3 test displacemene
30  0000401A 31FC 0005 6010          MOVE.W  #5,VALUE    input to fact is 5
32  00004020 3A38 6010      fact     MOVE.W  VALUE,D5    get input
33  00004024 DA45                    ADD.W   D5,D5       double for word offset
34  00004026 47F8 6000               LEA     FTABLE,A3   get base address
35  0000402A 3C33 5000               MOVE.W  0(A3,D5),D6 get result
36  0000402E 31C6 6012               MOVE.W  D6,RESULT   output
37
38                                   END main

Hewlett Packard AS68000 V(01.20 10Mar88) Page 2 Mon Oct 29 13:46:)


Symbol Table

Label        Value

DATA        00006000
FFSET       00000006
FTABLE      00006000
PROGRAM     00004000
RESULT      00006012
VALUE       00006010
fact        00004020
main        00004002
test        00006004
```

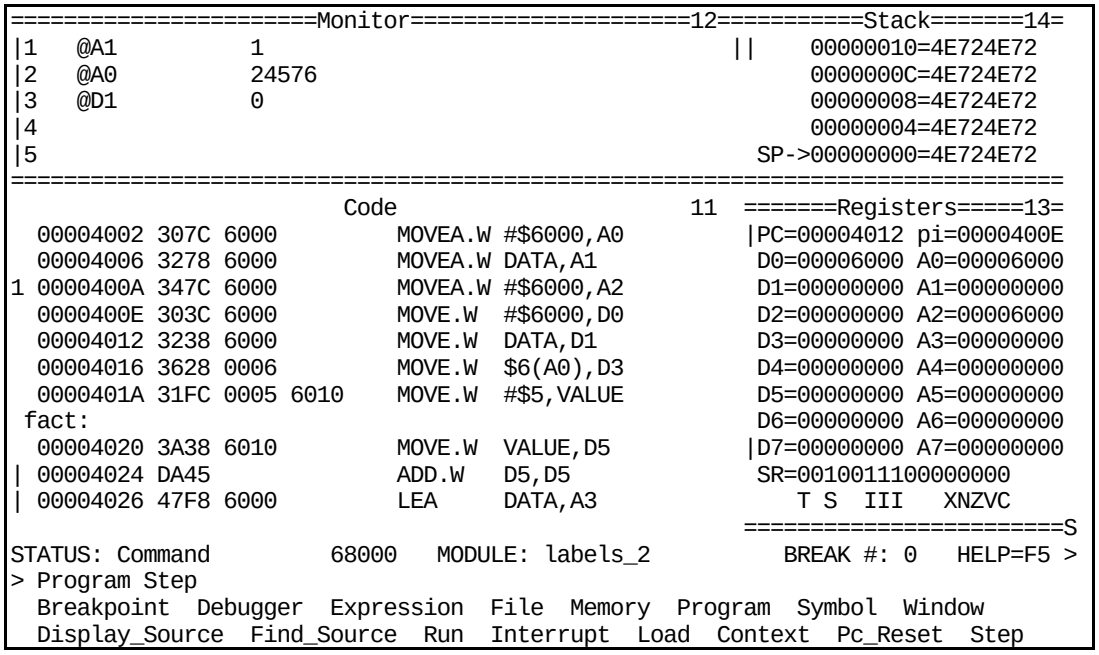


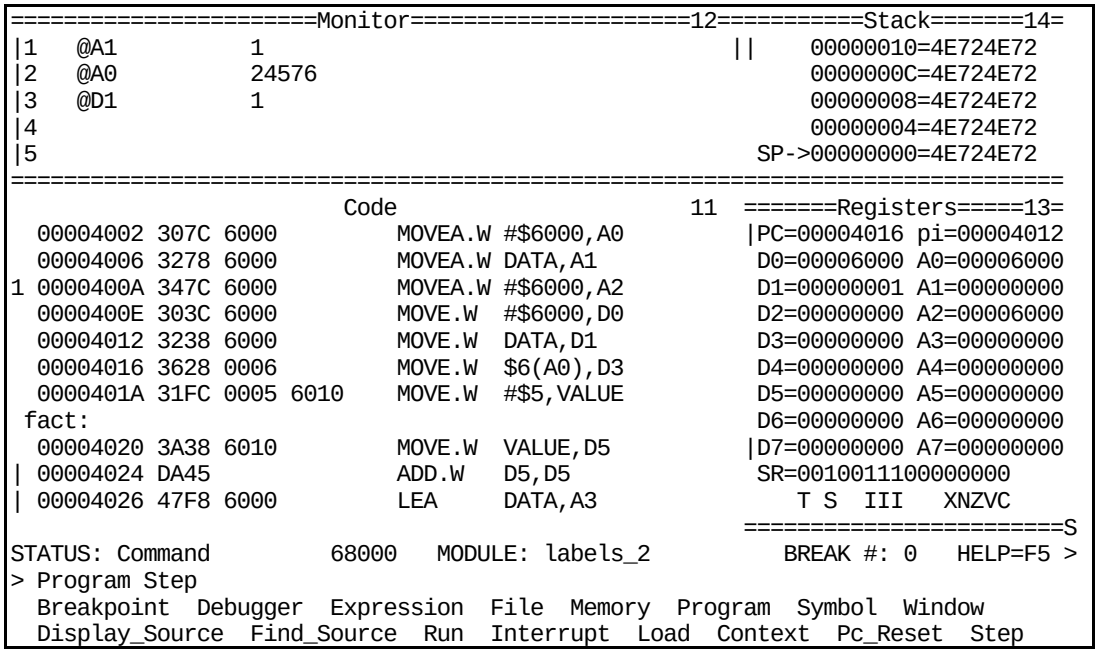


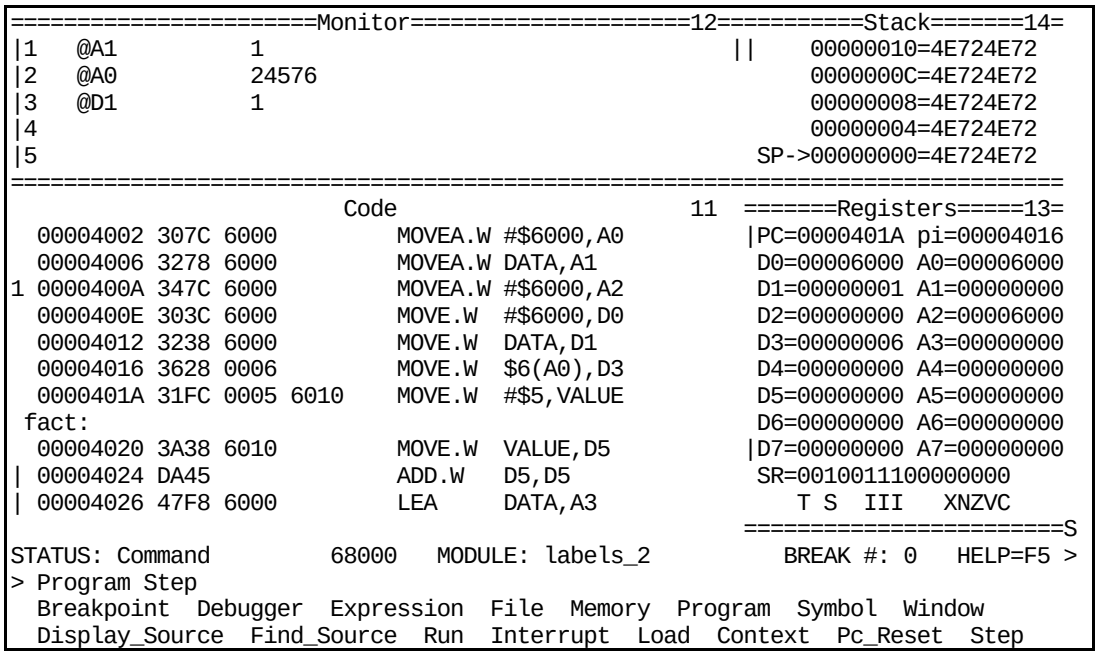
```
======================Monitor====================12==========Stack=======14=
|1   @A1          0                                ||    00000010=4E724E72
|2   @A0          24576                                 0000000C=4E724E72
|3   @D1          0                                     00000008=4E724E72
|4                                                      00000004=4E724E72
|5                                                    SP->00000000=4E724E72
================================================================================
                         Code                      11  =======Registers=====13=
  00004002 307C 6000         MOVEA.W #$6000,A0        |PC=00004006 pi=00004002
  00004006 3278 6000         MOVEA.W DATA,A1           D0=00000000 A0=00006000
  0000400A 347C 6000         MOVEA.W #$6000,A2         D1=00000000 A1=00000000
  0000400E 303C 6000         MOVE.W  #$6000,D0         D2=00000000 A2=00000000
  00004012 3238 6000         MOVE.W  DATA,D1           D3=00000000 A3=00000000
  00004016 3628 0006         MOVE.W  $6(A0),D3         D4=00000000 A4=00000000
  0000401A 31FC 0005 6010    MOVE.W  #$5,VALUE         D5=00000000 A5=00000000
 fact:                                                 D6=00000000 A6=00000000
  00004020 3A38 6010         MOVE.W  VALUE,D5         |D7=00000000 A7=00000000
| 00004024 DA45              ADD.W   D5,D5             SR=0010011100000000
| 00004026 47F8 6000         LEA     DATA,A3              T S  III   XNZVC
                                                      ======================S
STATUS: Command         68000   MODULE: labels_2          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

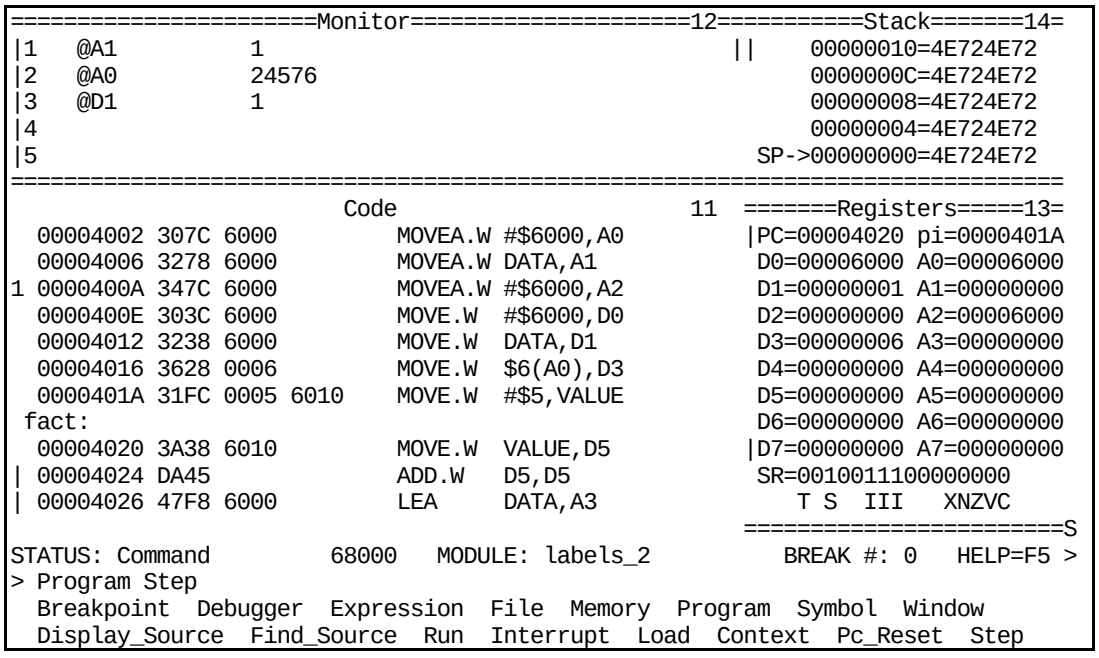
Debugger error occurs with instruction `MOVEA.W DATA,A1`

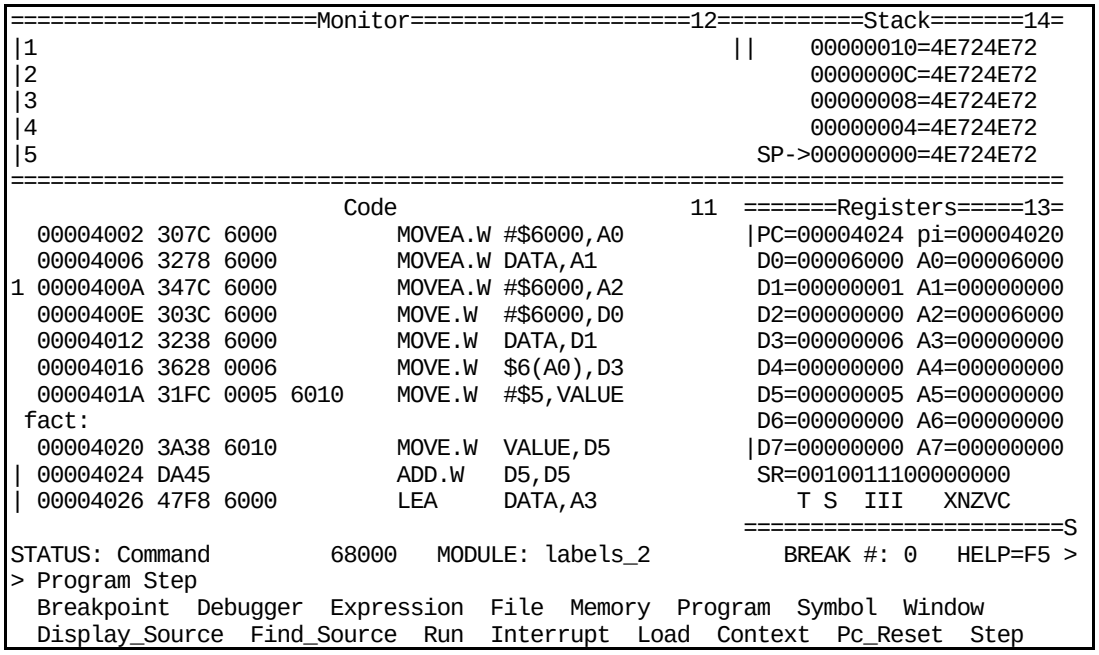
```
======================Monitor====================12==========Stack=======14=
|1   @A1          1                                ||    00000010=4E724E72
|2   @A0          24576                                 0000000C=4E724E72
|3   @D1          0                                     00000008=4E724E72
|4                                                      00000004=4E724E72
|5                                                    SP->00000000=4E724E72
================================================================================
                         Code                      11  =======Registers=====13=
  00004002 307C 6000         MOVEA.W #$6000,A0        |PC=0000400A pi=00004006
  00004006 3278 6000         MOVEA.W DATA,A1           D0=00000000 A0=00006000
1 0000400A 347C 6000         MOVEA.W #$6000,A2         D1=00000000 A1=00000000
  0000400E 303C 6000         MOVE.W  #$6000,D0         D2=00000000 A2=00000000
  00004012 3238 6000         MOVE.W  DATA,D1           D3=00000000 A3=00000000
  00004016 3628 0006         MOVE.W  $6(A0),D3         D4=00000000 A4=00000000
  0000401A 31FC 0005 6010    MOVE.W  #$5,VALUE         D5=00000000 A5=00000000
 fact:                                                 D6=00000000 A6=00000000
  00004020 3A38 6010         MOVE.W  VALUE,D5         |D7=00000000 A7=00000000
| 00004024 DA45              ADD.W   D5,D5             SR=0010011100000000
| 00004026 47F8 6000         LEA     DATA,A3              T S  III   XNZVC
                                                      ======================S
STATUS: Command         68000   MODULE: labels_2          BREAK #: 0   HELP=F5 >
> Program Step
  Breakpoint  Debugger  Expression  File  Memory  Program  Symbol  Window
  Display_Source  Find_Source  Run  Interrupt  Load  Context  Pc_Reset  Step
```

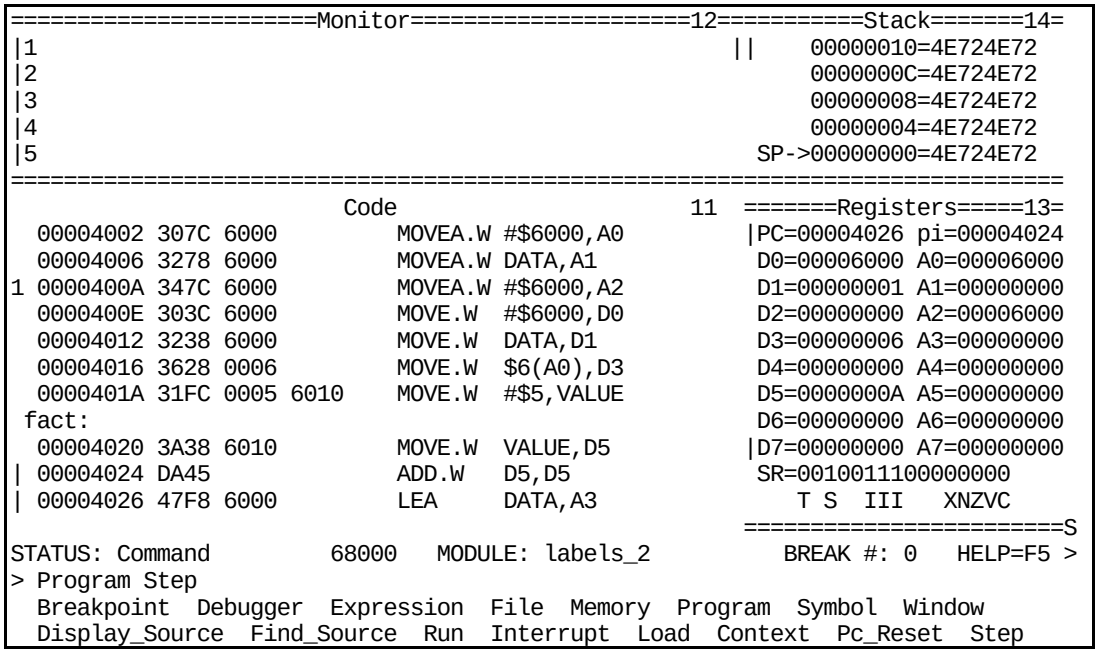


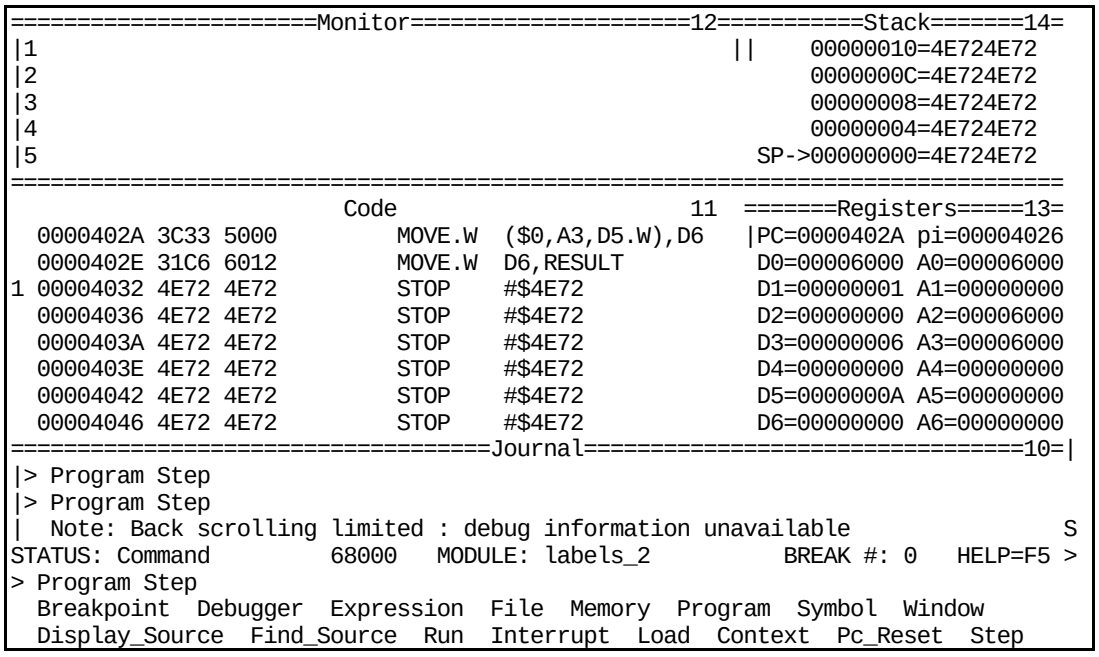


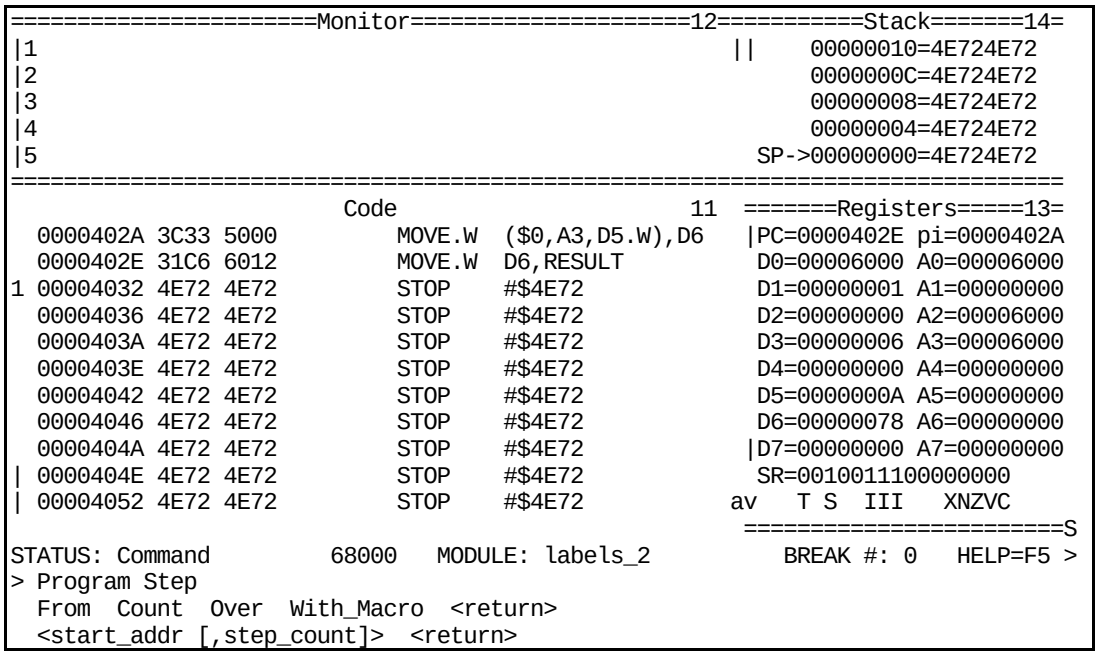


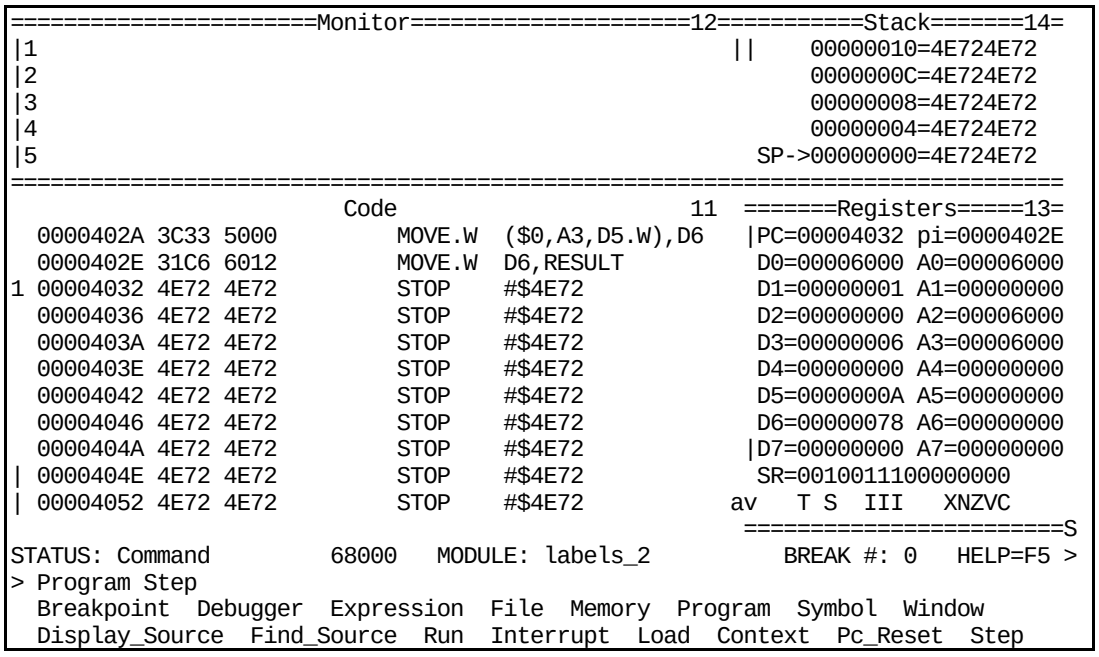


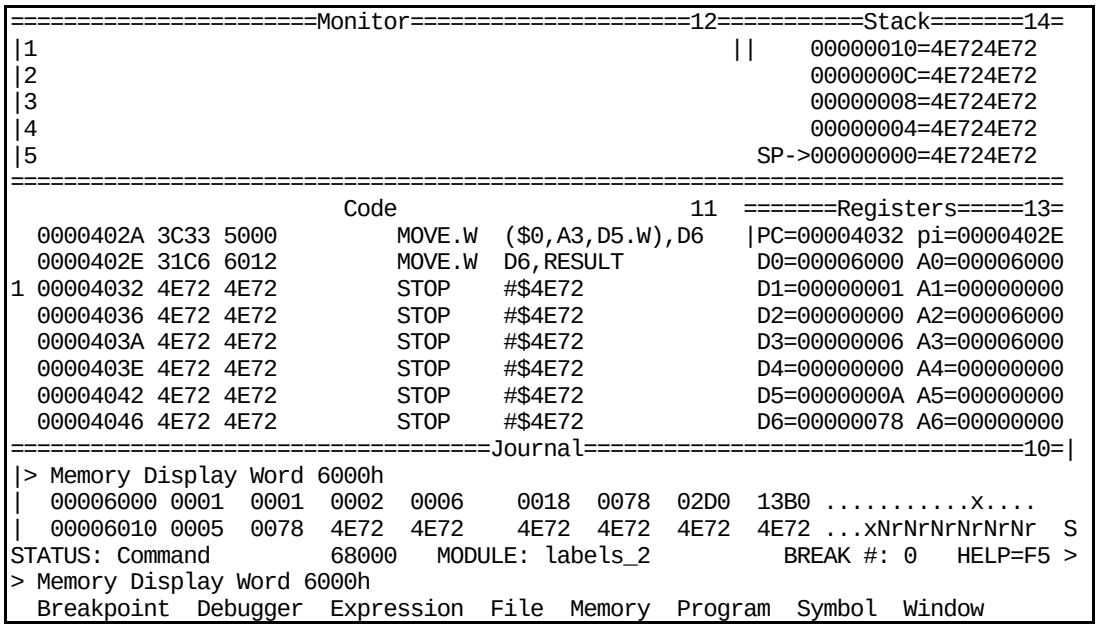












Assembler error messages

- error messages follow the line they are found on
- error messages are defined in lab manual Appendix

Error messages generated during an assembly may originate from the assembler or from a higher level language such as C (many assemblers are written in C) or from the operating system environment. Assembler-generated messages may be of two forms:

1. ****** ERROR xxx -- nnnn

where xxx is the number of the error (defined in the lab manual), and nnnn is the number of the line where the previous error occurred.

Errors indicate that the assembler is unable to interpret or implement the intent of a source line.

2. ****** WARNING xxx -- nnnn

where xxx is the number of the error (defined in the list in this appendix), and nnnn is the number of the line where the previous error occurred.

Warnings may indicate possible recoverable errors in the source code, or that a more optimal instruction format is possible.

Debugger error messages

- You will usually get an error message if the PC is not correctly set to where your assembly code begins. A typical error is to forget the ORG statement and start your program at $0.

- “Exception vector 8 at <address>“ is a very common error message. It usually comes from trying to execute a memory location containing 4E72. This is usually because the PC is not set correctly for your code or you are attempting to execute instructions from memory beyond the end of your program.

- You might get a blank screen while exiting the debugger. Type a Control-C followed by a control-Z. This is an internal debugger error and you might get a message like “Fatal Error: internal debugger error, segmentation violation.” which you should not worry about. Problems of this type should be reported to the TA’s.

- If you forget to set your environmental variable to the proper terminal type you will get the following error message:
  “Error: unable to open fast alpha output device”
  This can also occur when trying to use vi.

  You can set your environmental variable using the command “setenv TERM vt100” Typically you will this error message when logged in remotely.

Network Use:

When you are connecting across the network via fiber optics or modem the terminal type needs to be verified and changed as appropriate. It is a smart idea to check your assumed terminal type immediately after you login. To do this type **env** and the machine should respond with a message like

```
[5] % env
HOME=/users/merat
PATH=/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin:/us
r/hp64000/bin
LOGNAME=merat
SHELL=/bin/csh
MAIL=/usr/mail/merat
TZ=EST5EDT
TERM=unknown
```

The **TERM=unknown** indicates that the system does not know what your terminal type is. When you are connected to the Kern Lab through CWRUnet (either fiber optic or modem) you need to make sure that your terminal type has been set to **TERM=vt100**. You can do this by typing

```
[5] % setenv TERM vt100
```

This enables the vi editor and the debugger screens to work properly with your terminals. If you are interested, the vi editor and the debugger also support vt220 terminals and Hewlett-Packard 700/92 and 700/94 terminals.

You should immediately suspect that something is wrong with the TERM variable when your keys do not work properly. Keys with lots of problems are the backspace (delete) key and cursor keys. The DELETE key is very trickly; many terminal emulation programs will have a command or switch of the form MAP DEL-->BS which maps the delete command onto your keyboard backspace key. This option is also present in the CWRUnet software. Check this option if you have problems.

You may need to occasionally kill a UNIX process such as the debugger when you get into an infinite loop. You can only kill processes you have created.

At the bottom of the debugger window you will see

```
 Breakpoint  Debugger Expression  File  Memory  Program  Symbol  Window
  Execution  HP-UX_Shell  Macro  Option  Pause  Quit  Level  Directory  ?(Help
```

Typing the command

Debugger **H**P-UX_Shell

will open a Unix subshell with the following prompt:

```
> snowhite 3:
```

You can respond with

```
> snowhite 3: ps -ef
```

to determine what processes are running and what their ID's are. This will typically result in something like

```
UID   PID    PPID C    STIME TTY   TIME COMMAND
merat 23604 23602 5 14:05:13 ttyu0 0:00 ps -f
merat 23534 23533 0 14:04:03 ttyu0 0:02 -csh [csh]
merat 23602 23601 0 14:05:06 ttyu0 0:00 /bin/csh -i
merat 23601 23534 0 14:04:42 ttyu0 0:00 db68k exam1
```

where the db68k debugger process is the one you want to kill.

```
> snowhite 4: kill 23601

> snowhite 5: Stopped (tty output)

> snowhite 7: Reset tty pgrp from 23534 to 23602
Closed.
```

This kills the debugger and reassigns the terminal i/o to your current shell.

Common run-time errors on the 68000:

Note: You can only get run-time errors in the debugger

bus error	attempt to reference a non-existant memory address
illegal instruction	not a valid 68000 instruction op-code

ODD ADDRESS ERRORS:

```
        ORG     $1000
        MOVE.W  D0,D1
        DS.B    1
        MOVE.B  D0,D1
        ...
```

THERE ARE SEVERAL POTENTIAL PROBLEMS WITH THE ABOVE CODE:

(1) The byte extension forces the second MOVE to start on an odd address boundary;

(2) You should not intermix data and coded instructions. How does the 68000 know that the contents at the DS.B are data? if there is anything in the memory location it will be interpreted as a wrong instruction;

ANOTHER WAY OF GETTING ODD ADDRESS ERRORS:

```
            ORG     $7000
*TABLE OF FACTORIALS - NUMBERS ARE DECIMAL
FTABLE      DC      2000
            DC      7000
            DC      120
            DC      720
            DC      5040
VALUE       DS.B    1
RESULT      DS.W    1

            ORG     $4000
            CLR.L   D0
            CLR.L   D1
            MOVE    FTABLE,A0       ;base address is $0000 7000
            MOVE    1(A0),D0        ;computes 1($0000 7000) =
                                    $0000 7001, an ODD
                                    address which generates an
                                    ODD ADDRESS ERROR
            ADD     2(A0),D0
            MOVE    D0,RESULT
```

Example program: JUMP TABLE

```
            XREF    STOP
START:      LEA     COMMAND,A0      ;puts the address
                                    corresponding to the label
                                    COMMAND into A0
            ...
            ADDA.W  #$12,A0         ;coding error, you actually
                                    wanted 12 (decimal) as the
                                    offset
            MOVEA.L (A0),A0         ;get address of routine
            JMP     (A0)            ;jump off to routine

* this construction is called a jump table
CMD0:       code for routine #1     ; at address $0000 1000
CMD1:       code for routine #2     ; at address $0000 2000
CMD2:       code for routine #3     ; at address $0000 3000
CMD3:       code for routine #4     ; at address $0000 4000

                ORG     $500
* the following labels represent addresses of routines
COMMAND:    DC.L    CMD0,CMD1,CMD2,CMD3
I:          DC.L    $00000000, $10000000, $00005000
```

As shown the routine for CMD0 is located at $1000, the code for routine #2 is at $2000, the code for routine #3 is at $3000, and the code for routine #4 is at $4000. Then the DC.L beginning with the label COMMANDS

```
COMMANDS:   DC.L    CMD0,CMD1,CMD2,CMD3
I:          DC.L    $00000000, $10000000, $00005000
```

would look like this in memory:

	address	contents	
COMMANDS	00000500	00001000	address of CMD0 routine
	00000504	00002000	address of CMD1 routine
	00000508	00003000	address of CMD2 routine
	0000050C	00004000	address of CMD3 routine
I	00000510	00000000	
	00000514	10000000	
	00000518	00005000	

WHAT YOU WANTED THE PROGRAM TO DO:

1. You want to execute routine #3.
2. The LEA puts $500 into A0 for use as a base address.
3. You wanted an ADDA.W #12,A0 which would add 12_{10} = $C offset to the base address in A0 giving (A0)=$50C.
4. The MOVEA.L (A0),A0 will fetch the long word at $50C which is CMD3, the address of routine #3. This command leaves A0=$00003000.
5. JMP (A0) will cause the PC to be set to $00003000 and program execution will continue there.

WHAT ACTUALLY HAPPENS:

You typed ADDA.W #12,A0 in by mistake as shown

1. The LEA COMMAND,A0 instruction loads the same base address ($500) into A0 as before.
2. The ADDA.W #$12,A0 instruction adds the offset $12 to the base address in A0 to make (A0) = $0000 0512.
3. MOVEA.L (A0),A0 puts the value $0000 1000 into A0
4. JMP (A0) jumps to CMD1 rather than CMD3 as you were expecting and you don't know what is wrong!

If other data were at $514 then an odd address error might have resulted instead. Suppose ($514) = $1005, then MOVEA.L (A0),A0 would have put $00001005 into A0. JMP (A0) would have attempted to jump to $0000 1005 and an odd address error would have resulted since an instruction cannot begin on an odd address.

PROGRAMMING ERROR PROGRAM EXAMPLE

```
* Example 4.29
            XREF    STOP,HEXOUT  ;tells linker these are defined
                                    in other program modules
GO:         MOVE.W  $1006,D0     ;you actually wanted
                                    #1006,D0.  Suppose it puts
                                    $D079 into D0
            ADD.W   J,D0         ;add contents of J to D0
            ADD.W   D0,D0        multiply D0 by 2
            MOVE.W  D0,I         move 2*($D079+$FFF9) to
                                    memory location I

            JSR     HEXOUT       ;these are i/o routines that
                                    can be ignored
            MOVE.W  #8,D0
            ADD.W   I,D0
            ADD.W   J,D0
            JSR     HEXOUT
            JSR     STOP

I:          DS.W    1
J:          DC.W    $FFF9

            END
```

This error is insideous. The program assembles and runs but you put the wrong thing into D0.

$1006 is the contents of a memory location

#1006 is a simple decimal number

Kern Lab I/O Routines

In order to use these routines, the line:

```
include io.s
```

must be the first line of the user assembly code, and you must make the debugger load the file "lab3.com" by typing

```
db68k -c lab3.com <your_file_name>
```

HexIn
gets a hex number from the keyboard and stores its word-length value in D0.

Example:

```
          include   io.s
_Main     jsr       HexIn
          move.w    D0,num   ;stores input word
                             at num
          ...
```

HexOut
the inverse of HexIn; it outputs the word in D0 as a hex number to the debugger screen, followed by a line feed.

Example:

```
          include     io.s
_Main     move.w      #32,d0
          jsr         HexOut    ;will print 20 (20
                                decimal = $32)
```

HexOutLong
same as HexOut except that it prints the long word in D0

PrintString
This routine requires the value in A0 to point to a string, a sequence of non-zero bytes (usually ASCII characters) followed by a zero byte that identifies the end. When it is called, this routine will print to the screen the string pointed to by A0.

The string can have special ASCII values such as $0A for line feed or $08 for backspace.

Example:

```
            include   io.s
_Main       lea       str,a0
            jsr       PrintString          ;prints the below
                                           string
* does not add a line feed after the print
            ...                            ;rest of program
str         DC.B      “This is a string”,0
```

An example of a string with special characters might be:

```
str         DC.B      $0A, $0A, “he”, $08,”i”,$0A,00
```

This string can be interpreted as

linefeed	$0A
linefeed	$0A
ASCII text	"he"
backspace	$08
ASCII text	"i"
linefeed	$0A
terminating character	0

This string would print two blank lines followed by “he”. The $08 after “he” is a backspace so the e would be erased and replace by the “i” and a linefeed. The end result would be two blank lines, and “hi” on the next line. Anything else printed would go on the line after “hi”.

Using Polled I/O to Input From the Keyboard

I/O status register	$10040	0, no input yet 1, byte in the receive register
I/O receive register	$10042	contains the input byte, automatically clears the I/O status register

These routines require the ACIA.com file to be loaded into the debugger which is automatically done by lab3.com

Example:

```
            include     io.s            ;required for
                                        PrintString
status      equ         $10040
receive     equ         $10042
_Main       lea         str,a0
            jsr         PrintString     ;prompt used for
                                        input
test        tst.b       status
            beq         test            ;if 0, no byte is
                                        ready so continue
                                        to poll keyboard
            move.b      receive,D0      ;if not zero, get byte
                                        and do something
                                        with it
            ...                         ;rest of your
                                        program

str         DC.B        “Type a letter:”,0
```