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:
/usr/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 HP-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
```