Misc 386, 486 Instructions

Sometimes we want to move a small-sized piece of data into a larger register. For example, we might want to move an 8 bit value into a 16 bit register. Generally this occurs with numbers. We might have a number that is being represented by a byte, but now we want to move it into a register and have the 16 bit register operate on the data. We can coerce the smaller data to the larger data size using the PTR operator:

```
.data
mynum db 1, 2
.code
mov ax, 0AAAAh
mov ax, word ptr mynum ; AX now contains 0201
```

But what if we only wanted to move one byte into the AX register? That is, we really wanted to just move the number 1 into AX. One solution is to use AL. But then we would also have to zero out AH. Rather than use two instructions, there is a special instruction to zero out the unfilled bits in the destination. This is the `movzx` instruction (move with zero extend):

```
mov ax, 0AAAAh
movzx ax, mynum ; AX now contains 0001
```

The `movzx` instruction requires the .386 or higher processor directive. We can do a similar thing for the extended registers:

```
mov eax, mynum ; EAX now contains 00000001
```

In a similar fashion, the `movsx` instruction (move with sign extend) can be used to extend negative numbers. Consider what happens if we use `movzx` on a negative value:

```
.data
mynum db −1
.code
mov ax, 0AAAAh
movzx ax, mynum ; AX now contains 00FF
```

Why did we get 00FF? Because −1 is represented as FF in two’s complement. To correctly get −1 into AX, we need to extend FF’s all the way to the most significant bit. The `movsx` instruction pads with 1’s instead of 0’s:

```
movsx ax, mynum ; AX now contains FFFF or −1
```
Using the Irvine Link Library

The Irvine link library contains several useful routines to input data, output data, and perform several tasks that one would normally have to use many operating system calls to complete.

This library is called irvine.lib and should have been installed when you installed the CD-ROM from the Irvine book. Appendix E contains a full list of the library routines. We will only cover a few basic ones here.

When the assembler finds a reference to a name that is not in the current source file, it cannot calculate the name’s effective address. The EXTRN directive tells the assembler that this name is in another source file, and that the linker will fill in the address later. The format for EXTRN is:

```plaintext
EXTRN name:type [, name:type]
```

Where name is the procedure or label you want, and type is the kind of data that is located externally. Here are some examples:

```plaintext
Extrn Writeint:proc ; Procedure named Writeint
Extrn bufsize:word, keybufptr:dword ; word, dword variables
Extrn true:abs, false:abs ; abs=constant, true and false
```

Each of these EXTRN directives should be located in the corresponding segment where they are to be used:

```plaintext
Extrn true:abs, false:abs
.data
extrn bufsize:word
extrn somechar:byte
.code
extrn Writeint:proc
```

Most of what you will use in the Irvine link library are various procedures. To invoke a procedure use the format:

```plaintext
call procedureName
```

The call will push the IP onto the stack, and when the procedure returns, it will be popped off the stack and continue executing where we left off, just like a normal procedure in C or C++. The procedures will handle saving and restoring any registers that might be used. It is important to keep in mind that these are all high-level procedures – they were written by Irvine. That is, an x86 machine does not come standard with the procedures available. Consequently, if you ever wish to use these routines in other settings, you might need to write your own library routines.
Parameters are passed to the procedures through the registers. Here are some of the procedures available:

**Clrscr**  
Clears the screen, moves the cursor to the upper-left corner

**Crlf**  
Writes a carriage return / linefeed to the display

**Gotoxy**  
Locates the cursor at the specified X/Y coordinates on the screen.  
\[ \text{DH} = \text{row (0-24)}, \ \text{DL} = \text{column (0-79)} \]

**Writechar**  
Writes a single character to the current cursor location  
\[ \text{AL} \text{ contains the ASCII character} \]

**Writechar_direct**  
Writes a single character to video memory  
\[ \text{AL} = \text{character} \]  
\[ \text{AH} = \text{attribute (foreground/background color)} \]  
\[ \text{DH} = \text{row (0-24)}, \ \text{DL} = \text{column (0-79)} \]

**Scroll**  
Scroll a window on the screen with a chosen color  
\[ \text{CH} = \text{upper left corner row} \]  
\[ \text{CL} = \text{upper left corner column} \]  
\[ \text{DH} = \text{lower right row} \]  
\[ \text{DL} = \text{lower right column} \]  
\[ \text{BH} = \text{attribute (color) of scrolled lines} \]

**Readkey**  
Waits for a keypress.  
\[ \text{AH} = \text{key scan code} \]  
\[ \text{AL} = \text{ASCII code} \]

Let’s see these in action:

```
.code
main proc

    mov ax, @data
    mov ds, ax

    extrn Clrscr:proc, Crlf:proc, Gotoxy:proc, Writechar:proc,
            Writechar_direct:proc, Scroll:proc, Readkey:proc
```
; Illustrate ClrScr and Scroll
call Clrscr
mov ch, 10
mov cl, 10
mov dh, 60
mov dl, 60 ; 50 x 50 window starting at coordinate 10, 10 to 60,60
mov bh, 0AAh; AA = red
call Scroll ; Scroll
call Readkey ; Get a keypress

;Move the cursor around
mov dh, 24
mov dl, 79 ; bottom-right corner
call Gotoxy ; Move cursor there
mov al, '*'
call Writechar ; Write '*' in bottom right
call Readkey ; Get a keypress

;Directly output a row of '&'s to the screen
mov ah, 0E3h ; Set color attribute
mov al, '&
mov ex, 79
mov dh, 5 ; row 5
L1: mov dl, cl
call Writechar_direct
loop L1

mov ax, 4c00h
int 21h
main endp

end main

Here are some more:

**Randomize** Initialize random number seed

**Random_range** Generate a pseudo-random number Value in range from 0 … EAX-1

**Readchar** Reads a key if one is available without echo If char waiting, ZF = 0, AL = char Else, ZF = 1
Waitchar  
Waits for a key to be pressed without echo
AL contains the character.

Readlong  
Waits for/reads a ASCII string and interprets as a
a long 32 bit value. Stored in EAX.

Readint  
Waits for/reads a ASCII string and interprets as a
a 16 bit value. Stored in AX.

Readstring  
Waits for/reads a ASCII string.
Input: DX contains the offset to store the string
CX contains max character count
Output: AX contains number of chars input

Writeint  
Outputs AX as an unsigned integer
BX is the radix: 16=hex, 10=dec, 8=oct, 2=bin

Writeint_signed  
Outputs AX as a signed integer
Decimal format only.

Writestring  
Write a null-terminated string.
Input: DX points to the strings offset

Delay_seconds  
Waits EAX number of seconds.
This is better than the active-wait in the book
at 4.7.3

Here is another code sample:

.model small
.486
.stack 100h
.data
myint dw ?
mychar db ?
mystr db 30 dup(0)
myprompt db "Enter a string:","0
myprompt2 db "Enter a number:","0

.code
main proc
extrn Readlong:proc, Readstring:proc, Randomize:proc, Random_range:proc, \ 
   Readchar:proc, Writeint:proc, Writeint_signed:proc, Writestring:proc, \ 
   Delay_seconds:proc, Crlf:proc, Readint:proc

   mov ax, @data
   mov ds, ax


; Output 3 random numbers
call Randomize
mov eax, 5       ; Get number from 0-5
call Random_range
mov bx, 10
mov eax, 5       ; output AX as int in base 10
call Writeint
call Crlf
mov eax, 5       ; Get number from 0-5
call Random_range
call Writeint
Call Crlf
mov eax, 5       ; Get number from 0-5
call Random_range
call Writeint
Call Crlf
; Output last value again in binary
mov bx, 2
call Writeint
call Crlf

; Get and display a string
mov dx, offset myprompt
call Writestring
mov cx, 30       ; Max length of 30
call Readstring
; Output what was typed
call Writestring
Call Crlf

; Get a number and display it
mov dx, offset myprompt2
call Writestring
call Readint       ; Int stored in AX
call Crlf
mov bx, 10
call Writeint

mov ax, 4c00h
int 21h
main endp
end main

There are other functions for time, string comparisons, and disk operations.