Computer Architecture and System Software

Lecture 07: Assembly Language Programming

Instructor:
Rob Bergen
Applied Computer Science
University of Winnipeg

Announcements

- Assignment 3 posted
- Midterm marks posted
- You can pick up your midterm at the end of class

Midterm

- Observations:
  - Confusion between ‘exp’ and ‘E’ in FP numbers
  - Some students lost marks for not showing work while working with FPs

Branching and Loop Instructions

- JMP used to make the program execution jump to a specific label or address
  - Called an unconditional jump since jump always occurs
- jmp label
  - Label identifies the next instruction to be executed
- Example

```
                mov   ax, 1
inc_again:      inc   ax
jmp             inc_again
mov             bx, ax
```

Branching and Loop Instructions

- J<cond> used to make the program execution jump to a specific label or address **if the condition is true**
- j<cond> label
- Example

```
read_char:
                mov dl, 0
                …               ;Code for reading a character into al
                cmp al, 0Dh     ;Compare the character to ODh
                je CR_received  ;if equal, jump to CR_received
                inc cl          ;otherwise, increment cl and
                jmp read_char   ;go back to read another
CR_received:                    ;character from keyboard
…
```

Branching and Loop Instructions

- Note, while the result is not saved anywhere, the operation sets the zero flag (ZF = 1) if the two operands are the same
- je jump if equal
- jg jump if greater
- jl jump if less
- jge jump if greater than or equal
- jle jump if less than or equal
- jne jump if not equal
- jz jump if zero
- jnz jump if not zero
- jc jump if carry
- jnc jump if not carry

Branching Example

Consider the following code for different conditionals:

```
go_back:
   inc AL
   cmp AL,BL
   statement_1
   mov BL,77H
```

statement_1	AL	BL	Action
je go_back	66h	66h	inc AL
jg go_back	66h	65h	inc AL
jg go_back jl go_back	66h	66h	mov BL, 77h
jle go_back jge go_back	66h	66h	inc AL
jne go_back jg go_back jge go_back	67h	66h	inc AL

Branching and Loop Instructions

- How is iteration performed?

```
                mov cl, 50
repeat1:
                <loop body>
                dec cl          ;decrement cl, update flag
                jnz repeat1             ;jumps back to
                                        ;repeat1 if cl is not 0
```

Alternate loop instruction

loop operand (Uses cx register by default)

Example:

```
mov cx, 50              ;cx = 50 (there will be 50 loops)
loop_start:
        inc al
        loop loop_start ;decrements cx until cx = 0
                        ;if cx is not 0, jump to label
```

Assembly Examples

A note on DOSBOX

- If you have a file name that is over 6 characters, DOSBOX will shorten it like this: LongFileName.asm → LONGFI~1.asm
- If this has happened, you must type in the name as it appears in your directory

DOSBOX instructions (updated)

- To compile your program:

  masm NameOfFile.asm

  → masm NAMEOF~1.asm

  (depending on how long your filename is)

- To link your file:

  link NameOfFile

  → link NAMEOF~1

- To run your file:

  NameOfFile.exe

  → NAMEOF~1.exe

Debug and Tracing

- After compiling your exe file, you can type 'debug *ProgramName*.exe' in DOSBOX
- Typing t (for trace), you can trace the contents of registers one line at time.
- Note that trace traces though interrupt calls as well, so you will see more instructions than appear in your source code.
- 't *num*' traces through *num* lines at a time (ie. t 3)

- When you're done debugging, type q to quit

Debug and Tracing

- Let's try debugging AdditionExample2.asm (from the course website) in class

Procedures

```
push ax                    ;push contents of register ax to stack
push dx                    ;push contents of register dx to stack
call myproc                ;call (procedure name)
…                          ;instruction executed after procedure


; procedure
myproc proc                ;(procedure name) proc
    mov ah, 02h
    mov dl, 'S'            ;code for displaying 'S'
    int 21 h
    ret                    ;return to instruction after call
myproc endp                ;(procedure name) endp
```

Procedures (Background)

- The Pentium stack is defined to grow towards smaller addresses
- Stack Pointer (SP) points to the top of the stack
- Stack Segment (SS) points to the start of the stack segment in memory
- Call
  - Return address (IP) is pushed to stack
  - Jump to effective address given by operand
  - Return address is the address of the instruction following the call

Procedures (Background)

- Return
  - Pop return address from stack
  - Jump to that address

- Note: Procedures are defined after they are called, (after the final int 21h command) but they must be defined before the end of the code segment.

Parameter Passing

- There is little/no support for passing parameters to procedures
- When it comes to implementation
  - Convention
  - Its up to the programmer to follow conventions

Parameter Passing

- Passing parameters means getting data into a place set aside for the parameters
- Both the calling program and the procedure need to know where the parameters are
  - Calling program places them
    - Possibly uses values returned by the procedure
  - Procedure uses the parameters

Parameter Passing

- Simplest mechanism: use registers
  - Calling program puts the parameters into specific registers
  - Procedure uses them
    - Int 21-DOS function dispatcher

Parameter Passing

- Intel architecture suffers from not having enough registers
  - Very soon one runs out of registers to use
- Parameter passing by registers is not used as much with this method
  - Used on more modern architectures
- Solution: pass parameters on stack

Parameter Passing

- One last problem: What happens if procedure needs to use registers?
- Solution: store registers on stack before using them
- Two ways of implementing this:
  - Callee saved: a procedure clears out some registers for its own use
  - Caller saved: The calling program saves the registers and local variables that it does not want procedure to overwrite
- Either way you may need to use pusha, pushf, popa, popf

Parameter Passing

- Either way you may need to use pusha, pushf, popa, popf
- Note: MASM generates an error message if using the PUSHA or POPA
- By default the MASM generates code for the 8086 processor
  - These instructions are implemented only for the 80186, 80286, 80386 processors.
- To solve problem use a .186, .286, or .386 directive in the first line of your code
- See "Error A2105 with PUSHA and POPA instructions“
  - http://support.microsoft.com/kb/40192

Summary

- Before any procedure call
  - Caller gets parameters into correct location
  - Control is transferred to procedure
- Before procedure return
  - Put return values into correct location
  - Restore anything that needs to be restored
    - Return address, callee saved registers, frame pointer
  - Jump to return location

****Include extensive comments in procedure to indicate input and return values***

Reading large numbers

- Recall that we were able to read, display and perform arithmetic with one digit numbers last lecture

- We concluded that multi-digit numbers would have to be decoded one digit at a time
  - Like how we pushed one character at a time onto the stack while printing

- Let's look at the general procedure for the reading of a multi digit number

Reading numbers

- Steps:
  1. Read string into buffered input
  2. Point to last character in string
  3. Decode number, multiply by a power of 10
  4. Store result, check if done
  5. Otherwise, point to next character and repeat steps 3 and 4

  You will need a procedure like this for Assignment 3!

Assignment 3

- Strategy:
  - First write a procedure that can read a multi-digit number
  - Construct a loop that calculates x_k, x_{k+1}… You will need to test whether $x_k = x_{k+1}$ at each iteration and exit the loop if this is true.
  - Write a procedure that prints a multi-digit number to display results. (Opposite steps as reading)
  - You may need to clear registers from time to time in order to add/subtract/multiply/divide properly.
  - Use ‘mov ah, 0’ for example to clear ah.
  - Use unsigned integer division (div) (ie. 9/4 = 2)

Lab 06

- FP multiplication/division review
- Tracing register contents through assembly programs (either by hand or debug)
- Know how to use interrupt subroutines (read/display)
- You will be e-mailed a copy of an assembly program (or ask Ryan)