CS 271
Computer Architecture & Assembly Language

Lecture 17

Macros

Recursion

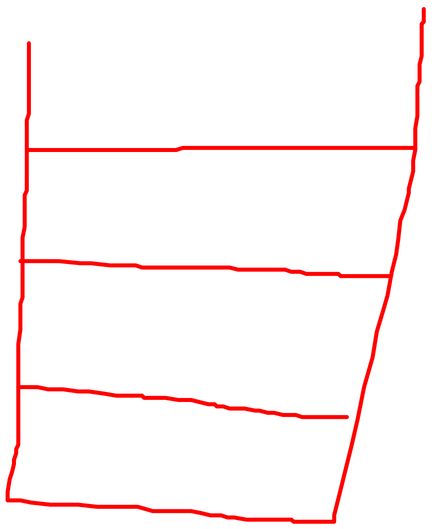
3/1/22, Tuesday

Lecture Topics:

- Macros
- Recursion

Macros

Procedure (Review)



- Separate, named sections of code
  - May have parameters
  - Calling mechanism
  - Return mechanism
- During assembly, procedure code is translated once
- During execution, control is transferred to the procedure at each call (activation record, etc.). May be called many times.
- All labels, etc. are local to the activation record.

Macro

- Separate, named section of code
  - May have parameters
- Once defined, it can be invoked (called) one or more times
  - Use name only (don't use CALL)
- During assembly, entire macro code is substituted for each call (expansion)
  - Similar to a constant
  - Invisible to the programmer

Defining Macros

- A macro must be defined before it can be invoked (i.e., in the program file, the definition must precede any invocations).
- Parameters are optional.
- Each parameter follows the rules for identifiers.
- Syntax:

```
macroname MACRO [param-1, param-2,...]
 statement-list
ENDM
```

Invoking Macros

- To invoke a macro, just give the name and the arguments (if any).
    - Each argument matches a declared parameter
    - Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro expands, it generates assembly language source code

Example Macro Definition and Call

- Sets up registers and uses Irvine' library *WriteString*

```
mWriteStr MACRO buffer
   push edx
   mov  edx,OFFSET buffer
   call WriteString
   pop  edx
ENDM
.data
str1 BYTE "Welcome!",10,13,0
str2 BYTE "Please tell me your name ",0
.code
   . . .
   mWriteStr str1
   mWriteStr str2
   . . .
```

Example Macro Expansion

- The expanded code shows how the str1 argument replaced the parameter named buffer:

```
mWriteStr MACRO buffer
    push edx
    mov  edx,OFFSET buffer
    call WriteString
    pop  edx
ENDM
```

```
1   push edx
1   mov  edx,OFFSET str1
1   call WriteString
1   pop  edx
```

Example Macro Definition and Call

- The *mReadStr* macro provides a convenient wrapper around *ReadString* procedure calls.

```
mReadStr MACRO varName
    push ecx
    push edx
    mov edx,OFFSET varName
    mov ecx, SIZEOF varName
    call ReadString
    pop edx
    pop ecx
ENDM
.data
firstName BYTE 30 DUP(?)
.code
    . . .
    mReadStr firstName
    . . .
```

A More Complex Macro

```
seq   macro      a, b         ; Print a sequence
      mov   eax,a             ;       from a to b
      mov   ebx,b
test:
      cmp   eax,ebx           ; if a <= b
      jg    quit              ; print a and repeat
      call  WriteDec          ; otherwise quit
      inc   eax
      jmp   test
quit:
endm
```

What’s the Problem?

- Code is expanded for each call
- If the macro is called more than once …

Duplicate labels

A More Complex Macro

```
seq   macro      a, b          ; Print a sequence
      mov   eax,a              ;        from a to b
      mov   ebx,b
test:
      cmp   eax,ebx            ; if a <= b
      jg    quit               ; print a and repeat
      call  WriteDec           ; otherwise quit
      inc   eax
      jmp   test
quit:
endm
```

Duplicate Labels

- You can specify that a label is LOCAL
- MASM handles the problem by appending a unique number to the label

```
Seq         macro          a, b
   LOCAL    test
   LOCAL    quit
                        ; Print a sequence
   mov      eax,a       ;      from a to b
   mov      ebx,b
test:
   cmp      eax,ebx     ; if a <= b
   jg       quit
   . . .
```

Parameters

- Arguments are substituted exactly as entered, so any valid argument can be used
- There is no checking for memory, registers, or literals
- Example calls to seq:

```
seq     x,y         ;memory
seq     ecx,edx     ;registers
seq     1,20        ;literals
```

Another Problem

ebx eax

```
seq     macro           a, b            ; Print a sequence
        mov     eax,a                   ;       from a to b
        mov     ebx,b
test:
        cmp     eax,ebx                 ; if a <= b
        jg      quit                    ; print a and repeat
        call    WriteDec                ; otherwise quit
        inc     eax
        jmp     test
quit:
endm
```

- What if macro is called with conflicting register parameters:
- E.g., **`seq     ebx, eax`**
- This macro would always print one number.

Macro vs. Procedure

- Macros are very convenient, easy to understand
- Macros actually execute faster than procedures
  - No return address, stack manipulation, etc.
- Macros are invoked by name
  - Parameter are “in-line”
  - Macro does not have a `ret` statement (why?)
- Why would you ever use a procedure instead of macro?
- If the macro is called many times, the assembler produces “fat code”
  - Invisible to the programmer
  - Each macro call expands the program code by the length of the macro code

Macro vs. Procedure

- Use a macro for short code that is called “a few” times, and uses only a few registers.
- Use a procedure for more complex tasks or code that is called “many” times.
  - The terms “few” and “many” are relative to the size of the whole program
- For both: Save registers!

- Is it OK to invoke a macro inside of a loop that executes 100 times?
- Is it OK to invoke a procedure inside of a loop that executes 100 times?

Demo

- Shows macros, macro calls, and macro parameters

Recursion

Recursion

- Many processes are defined by using previous results of the same process
- Example: summation (a, b) when a <= b
- Iterative definition:
  - Summation(a, b) = a + (a+1) + (a+2) + … + b
  - Recursive definition:

$$\sum_{i=a}^{a} i = a \qquad \sum_{i=a}^{b} i = a + \sum_{i=a+1}^{b} i$$

Recursion

- Note that the definition has two parts

$$\sum_{i=a}^{b} i = a + 1 + \sum_{i=a+2}^{b} i$$

Base case

$$\sum_{i=a}^{a} i = a$$

Recursive part

$$\sum_{i=a}^{b} i = a + \sum_{i=a+1}^{b} i$$

Recursive in Computer Programs

- Recursion occurs in programs when:
  - A procedure calls itself
  - Procedure A calls procedure B, which in turn calls procedure A
  - Calls are repeated in a cycle of procedure calls
- Recursion in programs mirrors recursive definitions

Example (pseudo-code)

```
function summation (a,b) returns sum of
   integers from a to b.
preconditions: a <= b

function summation (int a, int b):
if a == b
     return a
else return a + summation(a+1,b)
```

$$\sum_{i=a}^{a} i = a \qquad \sum_{i=a}^{b} i = a + \sum_{i=a+1}^{b} i$$

Demo

- Recursive version of summation problem

- Issues:
  - Using stack frames* for recursion is essential.
    - Why?
  - What causes stack overflow?
  - Why pass all 3 parameters (since 2 of them never change)?

*stack frame, activation frame, activation record

Recursion Warnings

$F(0) = 1$
$F(1) = 1$

- A good mathematical recursive definition does not necessarily imply a recursive procedure.
  - Example: Fibonacci sequence

$F(n) = F(n-1) + F(n-2)$

$F(5) = F(4) + F(3)$
$= F(3) + F(2) + F(2) + F(1)$

- Be sure that
  - The base case is defined
  - The base case is reachable
  - The recursive calls approach the base case
- Infinite (or too much) recursion results in “stack overflow”
- What would happen with a “recursive” macro?