CS 271
Computer Architecture &
Assembly Language

Lecture 9
The System Stack
More MASM Procedures
Intro to Parameter Passing
2/1/22, Tuesday
Odds and Ends

• Label names
  • Do not name them as L1, L2,... (our textbook give bad examples!)
    • Taking points off starting from programming assignment 4
  • Use meaningful names instead

• Indentation
  • Align in-line comments as well

• Midterm: 2/8 (Next Tuesday) during lecture time, same classroom
  • Review on Thursday
Lecture Topics:

• The System Stack
• More about MASM Procedures
• Documenting Procedures
• Register Management for Procedures
• Introduction to Parameter Passing
The System Stack
Stack

• Data structure (ADT)
• Last-in, first-out (LIFO or FILO)
• All operations reference the “top” of the stack
• Special names for operations
  • push, pop
• Applications:
  • Activation stack
  • Iterative implementation of recursive algorithms
  • Base conversion
  • Expression evaluation
  • Many others
The System Stack (Runtime Stack)

• The operating system maintains a stack
  • Implemented in memory
  • LIFO structure

• Managed by the CPU, using two registers
  • SS: address of stack segment
  • ESP: stack pointer (always points to “top” of stack)
    • i.e., ESP contains the address of the top of the stack
PUSH and POP Instructions (32-bit)

• **PUSH** syntax
  • PUSH $r/m32$
  • PUSH $immed$

• **POP** syntax
  • POP $r/m32$
PUSH Operation

• A push operation
  • **Decrements** the stack pointer by 4
  • Copies a value into the location pointed to by the stack pointer

• Actual decrement depends on the size of the operand
  • Note: it’s best to use 32-bit (DWORD, 4-byte) operands
Example PUSH

• Suppose that ECX contains 317 and ESP contains 0200h. In this case, [ESP] is 25.

• The next instruction is
  • push ecx

• Execute push ecx

• ESP: 01FCh

• [ESP]: 317

• Note: ESP is decremented, then 317 is stored in the stack

• Note: [ESP] means “content” of memory at the address in ESP

Stack Segment in Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01ECh</td>
<td>?</td>
</tr>
<tr>
<td>01F0h</td>
<td>?</td>
</tr>
<tr>
<td>01F4h</td>
<td>?</td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td></td>
</tr>
<tr>
<td>0200h</td>
<td>25</td>
</tr>
</tbody>
</table>
POP Operation

• A pop operation
  • Copies value at ESP into a register or variable.
  • Increments the stack pointer by 4

• Actual increment depends on the size of the operand
  • Note: it’s best to use 32-bit (DWORD, 4-byte) operands
Example POP

- Suppose that ESP contains 01FCh. In this case, [ESP] is 317
- The next instruction is `pop eax`
- Execute `pop eax`
- eax now contains 317
- ESP: 0200h
- [ESP]: 25
- Note: 317 is copied to EAX, then ESP is incremented. Memory contents unchanged.

### Stack Segment in Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>01ECh</td>
<td>?</td>
</tr>
<tr>
<td>01F0h</td>
<td>?</td>
</tr>
<tr>
<td>01F4h</td>
<td>?</td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>317</td>
</tr>
<tr>
<td>0200h</td>
<td>25</td>
</tr>
</tbody>
</table>

![Stack Segment in Memory Diagram]
Using PUSH and POP

• Save and restore registers when they contain important values. POP operands occur in the opposite of the order of PUSH operands

```assembly
push ecx ; save registers
push ebx

mov ecx,100h
mov ebx,0

; etc.

pop ebx ; restore registers
pop ecx
```
Example: Nested Loop

- Push the outer loop counter before entering the inner loop.
- Pop the outer loop counter when the inner loop terminates.

```asm
mov ecx,100 ; set outer loop count
L1:         ; begin the outer loop
    push ecx ; save outer loop count

mov ecx,20 ; set inner loop count
L2:         ; begin the inner loop
    ;
    ;
    loop L2 ; repeat the inner loop

    pop ecx ; restore outer loop count
loop L1    ; repeat the outer loop
```
When **not** to push

- Be sure that **PUSH** does not hide a return address
- Be sure that **POP** dose not lose a return address and/or replace needed values.
CALL and RET Instructions

• The **CALL** instruction calls a procedure
  • Pushes the offset of the next instruction onto the stack
  • Copies the address of the called procedure into EIP

• The **RET** instruction returns from a procedure
  • Pops top of stack into EIP
Procedure call/return Example (p1)

main PROC
...
    mov eax, 175
    mov ebx, 37
    mov edx, 25
    call Sum3
    mov result, eax
...
main ENDP

Sum3 PROC
    add eax, ebx
    add eax, edx
    ret
SumTwo ENDP

EAX ?
EBX ?
EDX ?
ESP 0200h
EIP 1202h (address of next instruction)

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...etc</td>
<td></td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>?</td>
</tr>
<tr>
<td>0200h</td>
<td>456</td>
</tr>
</tbody>
</table>
Procedure call/return Example (p2)

```
main  PROC
...
    mov  eax,175
    mov  ebx,37
    mov  edx,25
    call Sum3
    mov  result,eax
...
main  ENDP

Sum3  PROC
    add  eax, ebx
    add  eax, edx
    ret
SumTwo ENDP
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...etc</td>
<td></td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>?</td>
</tr>
<tr>
<td>0200h</td>
<td>456</td>
</tr>
</tbody>
</table>
Procedure call/return Example (p3)

main PROC
...
    mov    eax, 175
    mov    ebx, 37
    mov    edx, 25
    call   Sum3
    mov    result, eax
...
main ENDP

Sum3 PROC
    add    eax, ebx
    add    eax, edx
    ret
SumTwo ENDP

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...etc</td>
<td></td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>1216h (return address)</td>
</tr>
<tr>
<td>0200h</td>
<td>456</td>
</tr>
</tbody>
</table>
Procedure call/return Example (p4)

main PROC
...
  mov  eax,175
  mov  ebx,37
  mov  edx,25
  call Sum3
  mov  result,eax

main ENDP

Sum3 PROC
  add  eax, ebx
  add  eax, edx
  ret
SumTwo ENDP

EAX  237
EBX  37
EDX  25
ESP  01FCh
EIP  2C7Ah (address of ret instruction)

Stack Segment in Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...etc</td>
<td></td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>1216h</td>
</tr>
<tr>
<td>0200h</td>
<td>456</td>
</tr>
</tbody>
</table>
Procedure call/return Example (p5)

main PROC
...
    mov       eax,175
    mov       ebx,37
    mov       edx,25
    call      Sum3
    mov       result,eax
...
main ENDP

Sum3 PROC
    add       eax, ebx
    add       eax, edx
    ret
SumTwo ENDP

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>...etc</td>
<td></td>
</tr>
<tr>
<td>01F8h</td>
<td>?</td>
</tr>
<tr>
<td>01FCh</td>
<td>1216h</td>
</tr>
<tr>
<td>0200h</td>
<td>456</td>
</tr>
</tbody>
</table>

EAX   237
EBX   37
EDX   25
ESP   0200h
EIP   1216h (address of mov instruction)
The System Stack

• There is much more to learn about the system stack
  • Parameter passing
  • Activation records
  • Etc.

• Be sure that you understand:
  • How the stack works
  • Push decrements, Pop increments
  • The importance of keeping the stack aligned
More about MASM Procedures
Documenting Procedures
Register Management for Procedures
In MASM Procedures ... Beware!

• Avoid duplicate labels
  • Labels inside a procedure are only visible within that procedure
  • Don’t use the same label names in different procedures

• **Preconditions**: Be sure to set required registers before calling library procedures.

• Be aware of registers changed in procedures.
Local and Global Labels

- Procedures should be invoked by executing a `call` statement
  - Bad style (and a **very bad idea**) to jump into a procedure from outside the procedure
- Procedures should terminate by executing a `ret` statement
  - Bad style (and a **very bad idea**) to jump to a label outside a procedure
- Assembly language permits implementing some **very bad ideas** and **very bad styles**
  - However, good programmers don’t use them
Nested Procedure calls

• Any procedure might call another procedure
• Return addresses are “stacked” (LIFO)
• **RET** instructions must follow the order on the stack
  • This is one very good reason not to jump into or out of a procedure!

• It is essential that the stack be properly aligned when the **RET** instruction is executed!!
Documenting Procedures

• Documentation for each procedure:
  • Description: A description of the task accomplished by the procedure
  • Receives: A list of input parameters; state usage and requirements
  • Returns: A description of values returns by the procedure
  • Preconditions: List of requirements that must be satisfied before the procedure is called
  • Register changed: List of registers that may have different values than they had when the procedure was called

• If a procedure is called without satisfying the preconditions, the procedure’s creator makes no promise that it will work.
; Procedure to calculate the summation
; of integers from a to b.
; receives: a and b are global variables
; returns: global sum = a+(a+1)+ ... +b
; preconditions: a <= b
; registers changed: eax, ebx, ecx

calculate PROC
  ...
  ret
calculate ENDP
Saving Registers

• If a procedure changes any registers, the calling procedure might lose important data

• Two ways to save data:
  • By the calling procedure
    • Registers may be saved before call, and restored after return
  • By the called procedure
    • Registers may be saved at the beginning of the procedure, and restored before the return
Saving / Restoring Registers

• Methods:

1. Move register contents to named memory locations, then restore after procedure returns.

2. Use `pushad` and `popad`
   • Option 1: calling procedure pushes before call, pops after return
   • Option 2: called procedure pushes at beginning, and pops before the return

3. Save selected registers on the system stack
   • Option 1: calling procedure pushes before call, pops after return
   • Option 2: called procedure pushes at beginning, and pops before the return
Method 1:
Save Register Contents in Memory

- Example (in main ... aReg, bReg declared in .data)

```assembly
mov   aReg, eax ;save registers
mov   bReg, ebx
mov   eax, count ;set parameters
mov   ebx, OFFSET val
call  someProc
mov   eax, aReg ;restore registers
mov   ebx, bReg
```
Method 2: Save all Registers on the System Stack

- **pushad** pushes the 32-bit general-purpose registers onto the stack
  - Order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI

- **popad** pops the same registers off the stack in reverse order
  - Note: it’s best to use 32-bit (DWORD) operands
Method 2:
Save all Registers on the System Stack

• Example (Option 1: in calling procedure):

```assembly
pushad      ;save registers
call        someProc
popad       ;restore registers
...```
Method 2: Save all Registers on the System Stack

• Example (Option 2: in the called procedure):

```assembly
    calcSum    PROC
    pushad     ;save registers
    ...
    ;procedure body
    ...
    popad      ;restore registers
    ret
    calcSum    ENDP
```
Method 3:
Save Selected Registers on the System Stack

• Example:
  • **push eax**
    • pushes the contents of eax onto the system stack
  • **pop eax**
    • Pops the top of the system stack into eax
Methods 2 and 3: 
Save Registers on the System Stack

• **Warnings:**
  • Be sure that values don’t get lost
  • Be sure that the system stack is properly aligned
    • The return address must be on the top of the stack when the `ret` statement is executed!!
• Experiment with MASM

• Try several ways to do some simple tasks
• Use DEBUG to see what happens
Introduction to Parameter Passing
Parameters

• Definitions:
  • **Argument** *(actual parameters)* is a value or reference **passed to** a procedure
  • **Parameter** *(formal parameters)* is a value or reference **received by** a procedure
  • **Return value** is a value determined by the procedure, and **communicated back** to the calling procedure.

• No theoretical limit, but **practicality** and readability rule.
Parameters Classifications

• An **input parameter** is data passed by a calling program to a procedure.
  • The called procedure is not expected to modify the corresponding argument variable, and even if it does, the modification is confined to the procedure itself.

• An **output parameter** is created by passing the **address** of an argument variable when a procedure is called.
  • The “address of” a variable is the same thing as a “**pointer to**” or a “**reference to**” the variable. In MASM, we use **OFFSET**.
  • The procedure does not use any existing data from the variable, but it fills in new contents before it returns.

• An **input-output parameter** is the **address** of an argument variable which contains input that will be both **used** and **modified** by the procedure.
  • The content is modified at the memory address passed by the calling procedure.
Passing Values/Addresses to/from Procedures

• Methods:
  1. Use shared memory (global variables)
  2. Pass parameters in registers
  3. Pass parameters on the system stack
1. Use Shared Memory (Global Variables)

• Set up memory contents before call and/or before return
• Generally ... it’s a **bad idea** to use global variables
  • Procedure might change memory contents needed by other procedures (unwanted side-effect)

• **For Program #1 - #4 ... we use globals**
  • Later we will pass parameters on the system stack.
2. Pass Parameters in Registers

• Set up registers before call and/or before return
• Generally ... it’s a not a good idea to pass parameters in registers
  • Procedure might change register contents

• However
  • Some Irvine library procedures require values in registers (e.g., “Receives” and “Preconditions” for ReadString)
  • Some Irvine library procedures return values in registers (e.g., “Returns” for ReadInt)
3. Pass Parameters on the System Stack

• Push parameters onto the system stack before the call

• Two ways to use the parameters:
  • Procedure moves parameters from the stack into registers/variables
  • Set up a “stack frame”, and reference parameters directly on the stack

• Remove parameters and return to the calling program

• Much more later on this method

• This is the method used by high-level languages
Register vs. Stack Parameters

- Register parameters require dedicating a register to each parameter.
- Stack parameters make better use of system resources.
- Example:
  - Two ways of calling Summation procedure.

**Method 1** (parameters in registers):
```
pushad  ;save registers
mov    ebx,low
mov    ecx,high
call   Summation
mov    sum, eax
popad  ;restore registers
```

**Method 2** (parameters on stack):
```
push   low
push   high
push   OFFSET sum
call   Summation
```
Register vs. Stack Parameters

• Of course, methods of calling a procedure and passing parameters depend on the procedure implementation ... and vice-versa.
  • Some “setup” is involved (in the calling procedure)
  • Some “bookkeeping” is involved (in the called procedure)

• Parameters in registers require register management
• Parameters on the system stack require stack management
Saving Registers

• Remember!

• There’s only one set of registers.
• If a called procedure changes any registers, the calling procedure might lose important data
• In call cases, when a procedure is called:
  • Be aware of preconditions
    • What conditions must be true before the procedure can perform its task?
  • Be aware of what registers are changed (document!)
  • Save and restore registers if necessary