Procedure

Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/10/27

with slides by Kip Irvine

Announcements

- Assignment #1 due today after the class.
- Q10, Q11
- Midterm examination will be held on the week of 11/10. It is an openbook exam.
- Scope: chapters 1-5

Chapter overview

- Linking to an External Library
- The Book's Link Library
- Stack Operations
- Defining and Using Procedures
- Program Design Using Procedures

The book’s link library

Link library overview

- A file containing procedures that have been compiled into machine code
  - constructed from one or more OBJ files
- To build a library, . . .
  - start with one or more ASM source files
  - assemble each into an OBJ file
  - create an empty library file (extension .LIB)
  - add the OBJ file(s) to the library file, using the Microsoft LIB utility

Take a quick look at Irvine32.asm by clicking on Examples at the bottom of this screen.

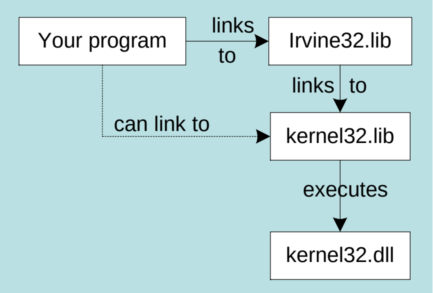
Calling a library procedure

- Call a library procedure using the CALL instruction. Some procedures require input arguments. The INCLUDE directive copies in the procedure prototypes (declarations).
- The following example displays "1234" on the console:

```
INCLUDE Irvine32.inc
.code
   mov eax,1234h        ; input argument
   call WriteHex        ; show hex number
   call Crlf            ; end of line
```

Linking to a library

- Your programs link to Irvine32.lib using the linker command inside a batch file named make32.bat.
- Notice the two LIB files: Irvine32.lib, and kernel32.lib
  - the latter is part of the Microsoft *Win32 Software Devlopment Kit*



Library procedures - overview (1 of 3)

Clrscr - Clears the console and locates the cursor at the upper left corner.

Crlf - Writes an end of line sequence to standard output.

Delay - Pauses the program execution for a specified *n* millisecond interval.

DumpMem - Writes a block of memory to standard output in hexadecimal.

DumpRegs - Displays the EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, and EIP registers in hexadecimal. Also displays the Carry, Sign, Zero, and Overflow flags.

GetCommandtail - Copies the program’s command-line arguments (called the *command tail*) into an array of bytes.

GetMseconds - Returns the number of milliseconds that have elapsed since midnight.

Library procedures - overview (2 of 3)

Gotoxy - Locates cursor at row and column on the console.

Random32 - Generates a 32-bit pseudorandom integer in the range 0 to FFFFFFFFh.

Randomize - Seeds the random number generator.

RandomRange - Generates a pseudorandom integer within a specified range.

ReadChar - Reads a single character from standard input.

ReadHex - Reads a 32-bit hexadecimal integer from standard input, terminated by the Enter key.

ReadInt - Reads a 32-bit signed decimal integer from standard input, terminated by the Enter key.

ReadString - Reads a string from standard input, terminated by the Enter key.

Library procedures - overview (3 of 3)

SetTextColor - Sets the foreground and background colors of all subsequent text output to the console.

WaitMsg - Displays message, waits for Enter key to be pressed.

WriteBin - Writes an unsigned 32-bit integer to standard output in ASCII binary format.

WriteChar - Writes a single character to standard output.

WriteDec - Writes an unsigned 32-bit integer to standard output in decimal format.

WriteHex - Writes an unsigned 32-bit integer to standard output in hexadecimal format.

WriteInt - Writes a signed 32-bit integer to standard output in decimal format.

WriteString - Writes a null-terminated string to standard output.

Example 1

Clear the screen, delay the program for 500 milliseconds, and dump the registers and flags.

```
.code
   call Clrscr
   mov eax,500
   call Delay
   call DumpRegs
```

Sample output:

```
EAX=00000613 EBX=00000000 ECX=000000FF EDX=00000000
ESI=00000000 EDI=00000100 EBP=0000091E ESP=000000F6
EIP=00401026 EFL=00000286 CF=0 SF=1 ZF=0 OF=0
```

Example 2

Display a null-terminated string and move the cursor to the beginning of the next screen line.

```
.data
str1 BYTE "Assembly language is easy!",0

.code
   mov edx,OFFSET str1
   call WriteString
   call Crlf
```

Example 3

Display the same unsigned integer in binary, decimal, and hexadecimal. Each number is displayed on a separate line.

```
IntVal = 35             ; constant
.code
   mov eax,IntVal
   call WriteBin        ; display binary
   call Crlf
   call WriteDec        ; display decimal
   call Crlf
   call WriteHex        ; display hexadecimal
   call Crlf
```

Sample output:

```
0000 0000 0000 0000 0000 0000 0010 0011
35
23
```

Example 4

Input a string from the user. EDX points to the string and ECX specifies the maximum number of characters the user is permitted to enter.

```
.data
fileName BYTE 80 DUP(0)

.code
   mov edx,OFFSET fileName
   mov ecx,SIZEOF fileName – 1
   call ReadString
```

Example 5

Generate and display ten pseudorandom signed integers in the range 0 – 99. Each integer is passed to WriteInt in EAX and displayed on a separate line.

```
.code
   mov ecx,10           ; loop counter

L1:
   mov  eax,100         ; ceiling value
   call RandomRange     ; generate random int
   call WriteInt        ; display signed int
   call Crlf            ; goto next display line
   loop L1              ; repeat loop
```

Example 6

Display a null-terminated string with yellow characters on a blue background.

```
.data
str1 BYTE "Color output is easy!",0

.code
   mov  eax,yellow + (blue * 16)
   call SetTextColor
   mov  edx,OFFSET str1
   call WriteString
   call Crlf
```

The background color must be multiplied by 16 before you add it to the foreground color.

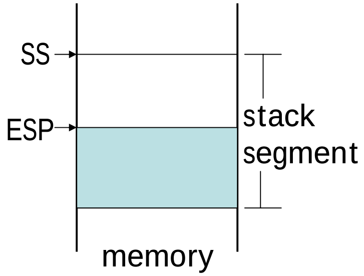
Stack operations

Stacks

- LIFO (Last-In, First-Out) data structure.
- push/pop operations
- You probably have had experiences on implementing it in high-level languages.
- Here, we concentrate on *runtime stack*, directly supported by hardware in the CPU. It is essential for calling and returning from procedures.

Runtime stack

- Managed by the CPU, using two registers
  - SS (stack segment)
  - ESP (stack pointer) * : point to the top of the stack usually modified by CALL, RET, PUSH and POP



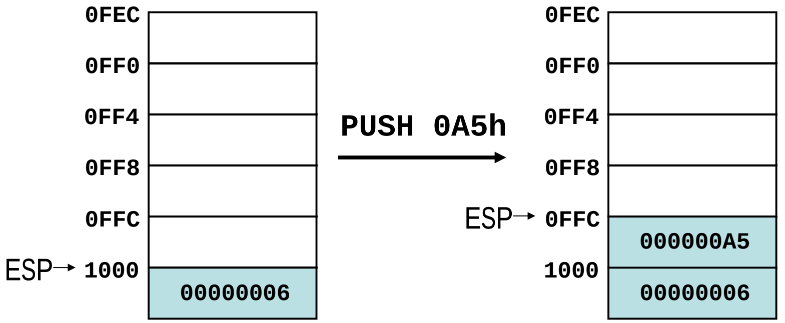
* SP in Real-address mode

PUSH and POP instructions

- PUSH syntax:
  - PUSH *r/m16*
  - PUSH *r/m32*
  - PUSH *imm32*
- POP syntax:
  - POP *r/m16*
  - POP *r/m32*

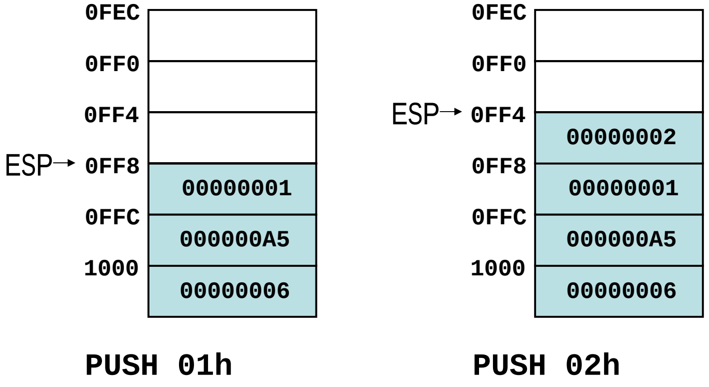
PUSH operation (1 of 2)

- A 32-bit push operation decrements the stack pointer by 4 and copies a value into the location pointed to by the stack pointer.



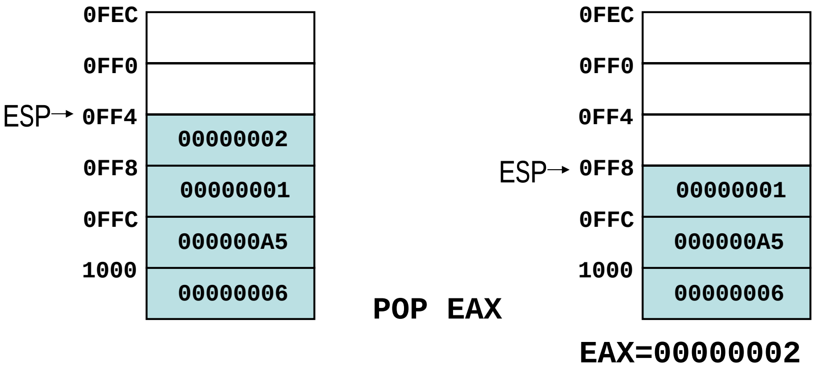
PUSH operation (2 of 2)

- This is the same stack, after pushing two more integers:



POP operation

- Copies value at stack[ESP] into a register or variable.
- Adds *n* to ESP, where *n* is either 2 or 4.
  - depends on the attribute of the operand receiving the data



When to use stacks

- Temporary save area for registers
- To save return address for CALL
- To pass arguments
- Local variables

Example of using stacks

Save and restore registers when they contain important values. Note that the PUSH and POP instructions are in the opposite order:

```
push esi                ; push registers
push ecx
push ebx

mov esi,OFFSET dwordVal    ; starting OFFSET
mov ecx,LENGTHOF dwordVal  ; number of units
mov ebx,TYPE dwordVal   ; size of a doubleword
call DumpMem            ; display memory

pop ebx                 ; opposite order
pop ecx
pop esi
```

Related instructions

- PUSHFD and POPFD
  - push and pop the EFLAGS register
- PUSHAD pushes the 32-bit general-purpose registers on the stack
  - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD pops the same registers off the stack in reverse order
  - PUSHA and POPA do the same for 16-bit registers

Example: reversing a string

```
.data
aName BYTE "Abraham Lincoln",0
nameSize = ($ - aName) – 1

.code
main PROC
; Push the name on the stack.
  mov ecx,nameSize
  mov esi,0
L1:
  movzx eax,aName[esi]   ; get character
  push eax               ; push on stack
  inc esi
  Loop L1
```

Example: reversing a string

```
; Pop the name from the stack, in reverse,
; and store in the aName array.
  mov ecx,nameSize
  mov esi,0
L2:
  pop eax           ; get character
  mov aName[esi],al ; store in string
  inc esi
  Loop L2

  exit
main ENDP
END main
```

Defining and using procedures

Creating Procedures

- Large problems can be divided into smaller tasks to make them more manageable
- A procedure is the ASM equivalent of a Java or C++ function
- Following is an assembly language procedure named sample:

```
sample PROC
   .
   .
   ret
sample ENDP
```

Documenting procedures

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- Receives: A list of input parameters; state their usage and requirements.
- Returns: A description of values returned by the procedure.
- Requires: Optional list of requirements called preconditions that must be satisfied before the procedure is called.

For example, a procedure of drawing lines could assume that display adapter is already in graphics mode,

Example: SumOf procedure

```
;------------------------------------------------
SumOf PROC
;
; Calculates and returns the sum of three 32-bit
;   integers.
; Receives: EAX, EBX, ECX, the three integers.
;           May be signed or unsigned.
; Returns: EAX = sum, and the status flags
;          (Carry, Overflow, etc.) are changed.
; Requires: nothing
;------------------------------------------------
   add eax,ebx
   add eax,ecx
   ret
SumOf ENDP
```

CALL and RET instructions

- The CALL instruction calls a procedure
  - pushes offset of next instruction on the stack
  - copies the address of the called procedure into EIP
- The RET instruction returns from a procedure
  - pops top of stack into EIP
- What do we need to do to have our toy computer to support CALL and RET?

CALL-RET example (1 of 2)

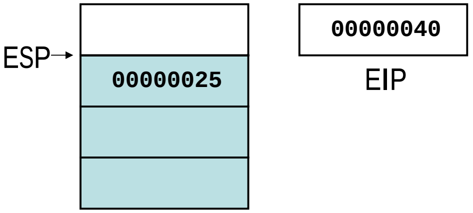
0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside MySub

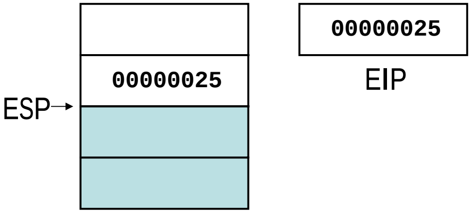
```
main PROC
   00000020 call MySub
   00000025 mov eax,ebx
   .
   .
main ENDP

MySub PROC
   00000040 mov eax,edx
   .
   .
   ret
MySub ENDP
```

CALL-RET example (2 of 2)

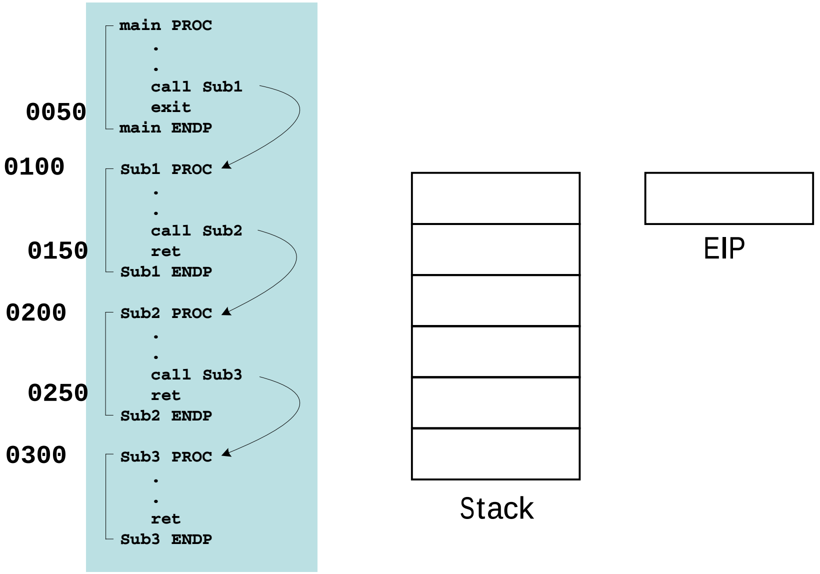
The CALL instruction pushes 00000025 onto the stack, and loads 00000040 into EIP



The RET instruction pops 00000025 from the stack into EIP



Nested procedure calls



```
     main PROC
        .
        .
        call Sub1
0050    exit
     main ENDP

0100 Sub1 PROC
        .
        .
        call Sub2
0150    ret
     Sub1 ENDP

0200 Sub2 PROC
        .
        .
        call Sub3
0250    ret
     Sub2 ENDP

0300 Sub3 PROC
        .
        .
        ret
     Sub3 ENDP
```

Local and global labels

A local label is visible only to statements inside the same procedure. A global label is visible everywhere.

```
main PROC
  jmp L2              ; error!
L1::                  ; global label
  exit
main ENDP

sub2 PROC
L2:                   ; local label
  jmp L1              ; ok
  ret
sub2 ENDP
```

Procedure parameters (1 of 3)

- A good procedure might be usable in many different programs
- Parameters help to make procedures flexible because parameter values can change at runtime
- General registers can be used to pass parameters

Procedure parameters (2 of 3)

The ArraySum procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
   mov esi,0            ; array index
   mov eax,0            ; set the sum to zero

L1:
   add eax,myArray[esi] ; add each integer to sum
   add esi,4            ; point to next integer
   loop L1              ; repeat for array size

   mov theSum,eax       ; store the sum
   ret
ArraySum ENDP
```

Procedure parameters (3 of 3)

This version of ArraySum returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Recevies: ESI points to an array of doublewords,
;           ECX = number of array elements.
; Returns:  EAX = sum
;-------------------------------------------------
   mov eax,0            ; set the sum to zero

L1:
   add eax,[esi]        ; add each integer to sum
   add esi,4            ; point to next integer
   loop L1              ; repeat for array size

   ret
ArraySum ENDP
```

Calling ArraySum

```
.data
array DWORD 10000h, 20000h, 30000h, 40000h
theSum DWORD ?
.code
main PROC
  mov     esi, OFFSET array
  mov     ecx, LENGTHOF array
  call    ArraySum
  mov     theSum, eax
```

USES operator

- Lists the registers that will be saved (to avoid side effects) (return register shouldn't be saved)

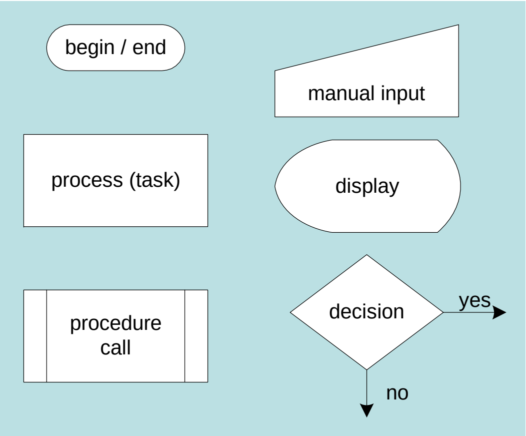
```
ArraySum PROC USES esi ecx
   mov eax,0  ; set the sum to zero
   ...
```

MASM generates the following code:

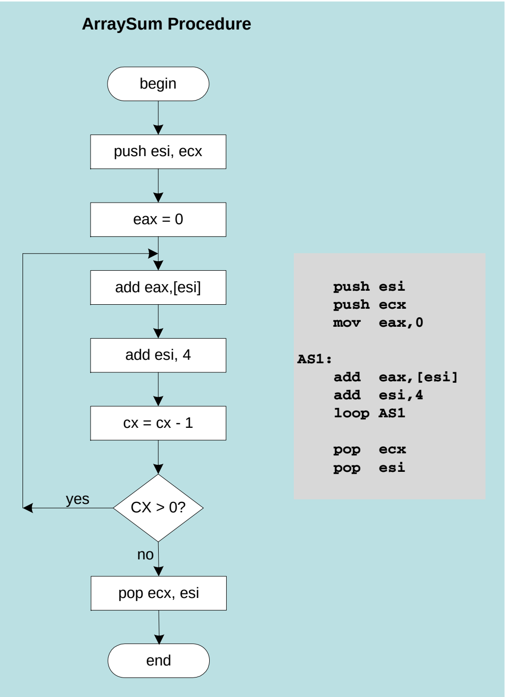
```
ArraySum PROC
   push esi
   push ecx
   .
   .
   pop ecx
   pop esi
   ret
ArraySum ENDP
```

Flowchart symbols

- The following symbols are the basic building blocks of flowcharts:



Flowchart for the ArraySum Procedure



```
    push esi
    push ecx
    mov  eax,0
AS1:
    add  eax,[esi]
    add  esi,4
    loop AS1

    pop  ecx
    pop  esi
```

Program design using procedures

- Top-Down Design (functional decomposition) involves the following:
  - – design your program before starting to code
  - – break large tasks into smaller ones
  - – use a hierarchical structure based on procedure calls
  - – test individual procedures separately

Integer summation program (1 of 4)

Spec.: Write a program that prompts the user for multiple 32-bit integers, stores them in an array, calculates the sum of the array, and displays the sum on the screen.

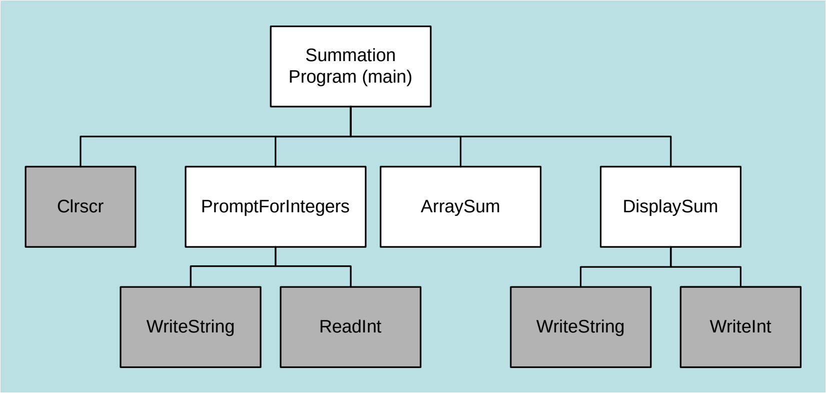
Main steps:

- Prompt user for multiple integers
- Calculate the sum of the array
- Display the sum

Procedure design (2 of 4)

```
Main
   Clrscr                          ; clear screen
   PromptForIntegers
        WriteString                ; display string
        ReadInt                    ; input integer
   ArraySum                        ; sum the integers
   DisplaySum
        WriteString                ; display string
        WriteInt                   ; display integer
```

Structure chart (3 of 4)



Summation Program (main)

Clrscr

PromptForIntegers

ArraySum

DisplaySum

WriteString

ReadInt

WriteString

WriteInt

PromptForIntegers

```
;---------------------------------------------------
PromptForIntegers PROC
;
; Prompts the user for an array of integers, and
; fills the array with the user's input.
; Receives: ESI points to the array,
;           ECX = array size
; Returns:  nothing
;--------------------------------------------------
  pushad          ; save all registers

  mov  edx,OFFSET prompt1 ; address of the prompt
  cmp  ecx,0              ; array size <= 0?
  jle  L2                 ; yes: quit
```

PromptForIntegers

```
L1:
  call WriteString      ; display string
  call ReadInt          ; read integer into EAX
  call Crlf             ; go to next output line
  mov  [esi],eax        ; store in array
  add  esi,4            ; next integer
  loop L1

L2:
  popad           ; restore all registers
  ret
PromptForIntegers ENDP
```

PromptForIntegers

```
;---------------------------------------------------
DisplaySum PROC
; Displays the sum on the screen
; Receives: EAX = the sum
; Returns:  nothing
;---------------------------------------------------
  push edx
  mov  edx,OFFSET prompt2  ; display message
  call WriteString
  call WriteInt            ; display EAX
  call Crlf
  pop  edx
  ret
DisplaySum ENDP
```

Code fragment

```
IntegerCount = 3              ; array size
.data
prompt1 BYTE  "Enter a signed integer: ",0
prompt2 BYTE  "The sum of the integers is: ",0
array   DWORD  IntegerCount DUP(?)
.code
main PROC
  call Clrscr
  mov  esi,OFFSET array
  mov  ecx,IntegerCount
  call PromptForIntegers
  call ArraySum
  call DisplaySum
  exit
main ENDP
```

Sample output (4 of 4)

```
Enter a signed integer: 550
Enter a signed integer: -23
Enter a signed integer: -96
The sum of the integers is: +431
```