

CMSC 313 Lecture 11

- **Project 3 Questions**
- **Recap Last Lecture**
- **Stack Instructions: PUSH, POP**
- **Subroutines (a.k.a. Functions) in Assembly**
- **How C functions pass parameters**

Project 3: External Records

Due: Tue 10/07/03, Section 0101 (Chang) & Section 0301 (Macneil)

Wed 10/08/03, Section 0201 (Patel & Bourner)

Objective

The objective of this programming project is to gain experience writing more complex assembly language programs and to use indexed addressing modes.

Assignment

Your assembly language program for this project will work with an externally defined array of records. This array is defined in a C program as follows:

```
struct {
    char  realname[32] ;
    char  nickname[16] ;
    char  alignment[20] ;
    char  role[20] ;
    int   points ;
    int   level ;
} records[10] ;

int num_records = 10 ;
```

The records in the array have pre-initialized values not shown here. The full text of the C program is available on the GL file system at: `/afs/umbc.edu/users/c/h/chang/pub/cs313/records.c`

Your assembly language program must search through the array and find the record with the least number of points and the record with the alphabetically first nickname. It must then print out the `realname` field of these two records. E.g.,

```
Lowest Points: James Pressman
First Nickname: Dan Gannett
```

Implementation Notes

- The sample data in `records.c` contains 10 records, but your program should work with any number of records. The number of records is stored in the `int` variable `num_records`.
- In order to access the externally defined array and integer variable, you must have the following declaration in your assembly language program:

```
extern    records, num_records
```

- You must also make your own test cases. The example in `records.c` does not fully exercise your program. Your program will be graded based upon other test cases.
- You will need to link your assembly language program with the data defined in the C program:

```
gcc -c records.c
nasm -f elf report.asm
ld records.o report.o
```

- An important part of this project is deciding how to use indexed addressing to access the data in the records. Think this through carefully. A clean and logical approach to this problem will yield clean and logical code that is easier to construct and, more importantly, easier to debug.

- Your program should be reasonably robust and report errors encountered (e.g., empty array) rather than crashing.
- Note that the strings stored in the array are C-style null-terminated strings.
- Nicknames should be compared using dictionary ordering. For example, any string starting with the letter 'a' comes before any string that starts with 'b'. In the case that one string is a prefix of another, the shorter string come first. E.g., "egg" comes before "egghead".
- To access each field of the record, you should use an offset from the address of the record. You should use `%define` constants instead of magic numbers. E.g.,

```
%define NickOffset 32
%define AlignOffset 48
%define RoleOffset 68
%define PointsOffset 88
%define LevelOffset 92
%define RecSize 96
```

- Project 4 will be based upon Project 3, so keep in mind that you will need to extend/modify this program.

Turning in your program

Use the UNIX `submit` command on the GL system to turn in your project. You should submit at least 4 files: your assembly language program, at least 2 of your own test cases and a typescript file of sample runs of your program. The class name for submit is `cs313_0101`, `cs313_0102` or `cs313_0103` for respectively sections 0101 (Chang), 0201 (Patel & Bourner) or 0301 (Macneil). The name of the assignment name is `proj3`. The UNIX command to do this should look something like:

```
submit cs313_0103 proj3 report.asm myrec1.c myrec2.c typescript
```

Last Time on Melrose Place

- **Compilers: mechanical process to turn high-level languages to assembly language**

- ◇ You can even write a compiler yourself with the help of lex & yacc
- ◇ Anything you can do in C you can also do in assembly language

- **Assembler**

- ◇ Needs two passes to back patch forward references
- ◇ Converts assembly language mnemonics to machine code
- ◇ You can even do this yourself by hand with the help of some tables from the Intel manual :-P

- **Linking & Loading**

- ◇ Linker resolves external references
- ◇ Loader assigns addresses to code and data sections.
- ◇ The loader must also patch instructions with real addresses

Stack Instructions

- **PUSH *op***

- ◇ the stack pointer ESP is decremented by the size of the operand
- ◇ the operand is copied to [ESP]

- **POP *op***

- ◇ the reverse of PUSH
- ◇ [ESP] is copied to the destination operand
- ◇ ESP is incremented by the size of the operand

- **Where is the stack?**

- ◇ The stack has its own section
- ◇ Linux processes wake up with ESP initialized properly
- ◇ The stack grows “upward” – toward smaller addresses
- ◇ Memory available to the stack set using ‘limit’

PUSH—Push Word or Doubleword Onto the Stack

Opcode	Instruction	Description
FF /6	PUSH <i>r/m16</i>	Push <i>r/m16</i>
FF /6	PUSH <i>r/m32</i>	Push <i>r/m32</i>
50+ <i>rw</i>	PUSH <i>r16</i>	Push <i>r16</i>
50+ <i>rd</i>	PUSH <i>r32</i>	Push <i>r32</i>
6A	PUSH <i>imm8</i>	Push <i>imm8</i>
68	PUSH <i>imm16</i>	Push <i>imm16</i>
68	PUSH <i>imm32</i>	Push <i>imm32</i>
0E	PUSH CS	Push CS
16	PUSH SS	Push SS
1E	PUSH DS	Push DS
06	PUSH ES	Push ES
0F A0	PUSH FS	Push FS
0F A8	PUSH GS	Push GS

Description

Decrements the stack pointer and then stores the source operand on the top of the stack. The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits), and the operand-size attribute of the current code segment determines the amount the stack pointer is decremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is decremented by 4 and, if they are 16, the 16-bit SP register is decremented by 2. (The B flag in the stack segment's segment descriptor determines the stack's address-size attribute, and the D flag in the current code segment's segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the source operand.) Pushing a 16-bit operand when the stack address-size attribute is 32 can result in a misaligned the stack pointer (that is, the stack pointer is not aligned on a doubleword boundary).

The PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. Thus, if a PUSH instruction uses a memory operand in which the ESP register is used as a base register for computing the operand address, the effective address of the operand is computed before the ESP register is decremented.

In the real-address mode, if the ESP or SP register is 1 when the PUSH instruction is executed, the processor shuts down due to a lack of stack space. No exception is generated to indicate this condition.

IA-32 Architecture Compatibility

For IA-32 processors from the Intel 286 on, the PUSH ESP instruction pushes the value of the ESP register as it existed before the instruction was executed. (This is also true in the real-address and virtual-8086 modes.) For the Intel 8086 processor, the PUSH SP instruction pushes the new value of the SP register (that is the value after it has been decremented by 2).



PUSH—Push Word or Doubleword Onto the Stack (Continued)**Operation**

```

IF StackAddrSize 32
THEN
  IF OperandSize 32
  THEN
    ESP ESP □ 4;
    SS:ESP SRC; (* push doubleword *)
  ELSE (* OperandSize 16*)
    ESP ESP □ 2;
    SS:ESP SRC; (* push word *)
  FI;
ELSE (* StackAddrSize 16*)
  IF OperandSize 16
  THEN
    SP SP □ 2;
    SS:SP SRC; (* push word *)
  ELSE (* OperandSize 32*)
    SP SP □ 4;
    SS:SP SRC; (* push doubleword *)
  FI;
FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit. If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
-----	---

POP—Pop a Value from the Stack

Opcode	Instruction	Description
8F /0	POP <i>m16</i>	Pop top of stack into <i>m16</i> ; increment stack pointer
8F /0	POP <i>m32</i>	Pop top of stack into <i>m32</i> ; increment stack pointer
58+ <i>rw</i>	POP <i>r16</i>	Pop top of stack into <i>r16</i> ; increment stack pointer
58+ <i>rd</i>	POP <i>r32</i>	Pop top of stack into <i>r32</i> ; increment stack pointer
1F	POP DS	Pop top of stack into DS; increment stack pointer
07	POP ES	Pop top of stack into ES; increment stack pointer
17	POP SS	Pop top of stack into SS; increment stack pointer
0F A1	POP FS	Pop top of stack into FS; increment stack pointer
0F A9	POP GS	Pop top of stack into GS; increment stack pointer

Description

Loads the value from the top of the stack to the location specified with the destination operand and then increments the stack pointer. The destination operand can be a general-purpose register, memory location, or segment register.

The address-size attribute of the stack segment determines the stack pointer size (16 bits or 32 bits—the source address size), and the operand-size attribute of the current code segment determines the amount the stack pointer is incremented (2 bytes or 4 bytes). For example, if these address- and operand-size attributes are 32, the 32-bit ESP register (stack pointer) is incremented by 4 and, if they are 16, the 16-bit SP register is incremented by 2. (The B flag in the stack segment’s segment descriptor determines the stack’s address-size attribute, and the D flag in the current code segment’s segment descriptor, along with prefixes, determines the operand-size attribute and also the address-size attribute of the destination operand.)

If the destination operand is one of the segment registers DS, ES, FS, GS, or SS, the value loaded into the register must be a valid segment selector. In protected mode, popping a segment selector into a segment register automatically causes the descriptor information associated with that segment selector to be loaded into the hidden (shadow) part of the segment register and causes the selector and the descriptor information to be validated (see the “Operation” section below).

A null value (0000-0003) may be popped into the DS, ES, FS, or GS register without causing a general protection fault. However, any subsequent attempt to reference a segment whose corresponding segment register is loaded with a null value causes a general protection exception (#GP). In this situation, no memory reference occurs and the saved value of the segment register is null.

The POP instruction cannot pop a value into the CS register. To load the CS register from the stack, use the RET instruction.

If the ESP register is used as a base register for addressing a destination operand in memory, the POP instruction computes the effective address of the operand after it increments the ESP register. For the case of a 16-bit stack where ESP wraps to 0h as a result of the POP instruction, the resulting location of the memory write is processor-family-specific.

POP—Pop a Value from the Stack (Continued)

The POP ESP instruction increments the stack pointer (ESP) before data at the old top of stack is written into the destination.

A POP SS instruction inhibits all interrupts, including the NMI interrupt, until after execution of the next instruction. This action allows sequential execution of POP SS and MOV ESP, EBP instructions without the danger of having an invalid stack during an interrupt¹. However, use of the LSS instruction is the preferred method of loading the SS and ESP registers.

Operation

```

IF StackAddrSize 32
  THEN
    IF OperandSize 32
      THEN
        DEST SS:ESP; (* copy a doubleword *)
        ESP ESP + 4;
      ELSE (* OperandSize 16*)
        DEST SS:ESP; (* copy a word *)
        ESP ESP + 2;
      FI;
    ELSE (* StackAddrSize 16* )
      IF OperandSize 16
        THEN
          DEST SS:SP; (* copy a word *)
          SP SP + 2;
        ELSE (* OperandSize 32 *)
          DEST SS:SP; (* copy a doubleword *)
          SP SP + 4;
        FI;
      FI;
    FI;

```

Loading a segment register while in protected mode results in special checks and actions, as described in the following listing. These checks are performed on the segment selector and the segment descriptor it points to.

```

IF SS is loaded;
  THEN
    IF segment selector is null
      THEN #GP(0);

```

1. Note that in a sequence of instructions that individually delay interrupts past the following instruction, only the first instruction in the sequence is guaranteed to delay the interrupt, but subsequent interrupt-delaying instructions may not delay the interrupt. Thus, in the following instruction sequence:

```

STI
POP SS
POP ESP

```

interrupts may be recognized before the POP ESP executes, because STI also delays interrupts for one instruction.

POP—Pop a Value from the Stack (Continued)

```

    FI;
    IF segment selector index is outside descriptor table limits
      OR segment selector's RPL > CPL
      OR segment is not a writable data segment
      OR DPL > CPL
      THEN #GP(selector);
    FI;
    IF segment not marked present
      THEN #SS(selector);
  ELSE
    SS    segment selector;
    SS    segment descriptor;
  FI;
  FI;
  IF DS, ES, FS, or GS is loaded with non-null selector;
  THEN
    IF segment selector index is outside descriptor table limits
      OR segment is not a data or readable code segment
      OR ((segment is a data or nonconforming code segment)
          AND (both RPL and CPL > DPL))
      THEN #GP(selector);
    IF segment not marked present
      THEN #NP(selector);
  ELSE
    SegmentRegister    segment selector;
    SegmentRegister    segment descriptor;
  FI;
  FI;
  IF DS, ES, FS, or GS is loaded with a null selector;
  THEN
    SegmentRegister    segment selector;
    SegmentRegister    segment descriptor;
  FI;
  FI;

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If attempt is made to load SS register with null segment selector.
	If the destination operand is in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

Subroutine Instructions

- **CALL *label***

- ◇ Used to call a subroutine
- ◇ PUSHes the instruction pointer (EIP) on the stack
- ◇ jump to the label
- ◇ does NOTHING else

- **RET**

- ◇ reverse of CALL
- ◇ POPs the instruction pointer (EIP) off the stack
- ◇ execution proceeds from the instruction after the CALL instruction

- **Parameters?**

CALL—Call Procedure

Opcode	Instruction	Description
E8 <i>cw</i>	CALL <i>rel16</i>	Call near, relative, displacement relative to next instruction
E8 <i>cd</i>	CALL <i>rel32</i>	Call near, relative, displacement relative to next instruction
FF /2	CALL <i>r/m16</i>	Call near, absolute indirect, address given in <i>r/m16</i>
FF /2	CALL <i>r/m32</i>	Call near, absolute indirect, address given in <i>r/m32</i>
9A <i>cd</i>	CALL <i>ptr16:16</i>	Call far, absolute, address given in operand
9A <i>cp</i>	CALL <i>ptr16:32</i>	Call far, absolute, address given in operand
FF /3	CALL <i>m16:16</i>	Call far, absolute indirect, address given in <i>m16:16</i>
FF /3	CALL <i>m16:32</i>	Call far, absolute indirect, address given in <i>m16:32</i>

Description

Saves procedure linking information on the stack and branches to the procedure (called procedure) specified with the destination (target) operand. The target operand specifies the address of the first instruction in the called procedure. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of calls:

- Near call—A call to a procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment call.
- Far call—A call to a procedure located in a different segment than the current code segment, sometimes referred to as an intersegment call.
- Inter-privilege-level far call—A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- Task switch—A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for additional information on near, far, and inter-privilege-level calls. See Chapter 6, *Task Management*, in the *IA-32 Intel Architecture Software Developer’s Manual, Volume 3*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) onto the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified with the target operand. The target operand specifies either an absolute offset in the code segment (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register, which points to the instruction following the CALL instruction). The CS register is not changed on near calls.



CALL—Call Procedure (Continued)

For a near call, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s, resulting in a maximum instruction pointer size of 16 bits. (When accessing an absolute offset indirectly using the stack pointer [ESP] as a base register, the base value used is the value of the ESP before the instruction executes.)

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP register. As with absolute offsets, the operand-size attribute determines the size of the target operand (16 or 32 bits).

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real-address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers onto the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared to 0s.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following three types of far calls:

- Far call to the same privilege level.
- Far call to a different privilege level (inter-privilege level call).
- Task switch (far call to another task).

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register.

CALL—Call Procedure (Continued)

TASK-GATE:

```

    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector);
    FI;
    IF task gate not present
        THEN #NP(task gate selector);
    FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
        OR index not within GDT limits
        THEN #GP(TSS selector);
    FI;
    Access TSS descriptor in GDT;

    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector);
    FI;
    IF TSS not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;
```

TASK-STATE-SEGMENT:

```

    IF TSS DPL < CPL or RPL
    OR TSS descriptor indicates TSS not available
        THEN #GP(TSS selector);
    FI;
    IF TSS is not present
        THEN #NP(TSS selector);
    FI;
    SWITCH-TASKS (with nesting) to TSS
    IF EIP not within code segment limit
        THEN #GP(0);
    FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

RET—Return from Procedure

Opcode	Instruction	Description
C3	RET	Near return to calling procedure
CB	RET	Far return to calling procedure
C2 <i>iw</i>	RET <i>imm16</i>	Near return to calling procedure and pop <i>imm16</i> bytes from stack
CA <i>iw</i>	RET <i>imm16</i>	Far return to calling procedure and pop <i>imm16</i> bytes from stack

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns:

- Near return—A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- Far return—A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- Inter-privilege-level far return—A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled “Calling Procedures Using Call and RET” in Chapter 6 of the *IA-32 Intel Architecture Software Developer’s Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

RET—Return from Procedure (Continued)

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

Operation

(* Near return *)

```

IF instruction    near return
  THEN;
    IF OperandSize    32
      THEN
        IF top 12 bytes of stack not within stack limits THEN #SS(0); FI;
        EIP    Pop();
      ELSE (* OperandSize    16 *)
        IF top 6 bytes of stack not within stack limits
          THEN #SS(0)
        FI;
        tempEIP    Pop();
        tempEIP    tempEIP AND 0000FFFFH;
        IF tempEIP not within code segment limits THEN #GP(0); FI;
        EIP    tempEIP;
      FI;
    IF instruction has immediate operand
      THEN IF StackAddressSize=32
        THEN
          ESP    ESP + SRC; (* release parameters from stack *)
        ELSE (* StackAddressSize=16 *)
          SP    SP + SRC; (* release parameters from stack *)
        FI;
      FI;
    FI;

```

(* Real-address mode or virtual-8086 mode *)

```

IF ((PE    0) OR (PE    1 AND VM    1)) AND instruction    far return
  THEN;

```

RET—Return from Procedure (Continued)

```

ELSE (* OperandSize=16 *)
    EIP Pop();
    EIP EIP AND 0000FFFFH;
    CS Pop(); (* 16-bit pop; segment descriptor information also loaded *)
    CS(RPL) CPL;
    ESP ESP + SRC; (* release parameters from called procedure's stack *)
    tempESP Pop();
    tempSS Pop(); (* 16-bit pop; segment descriptor information also loaded *)
    (* segment descriptor information also loaded *)
    ESP tempESP;
    SS tempSS;
FI;
FOR each of segment register (ES, FS, GS, and DS)
    DO;
        IF segment register points to data or non-conforming code segment
        AND CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
        THEN (* segment register invalid *)
            SegmentSelector 0; (* null segment selector *)
        FI;
    OD;
For each of ES, FS, GS, and DS
    DO
        IF segment selector index is not within descriptor table limits
        OR segment descriptor indicates the segment is not a data or
        readable code segment
        OR if the segment is a data or non-conforming code segment and the segment
        descriptor's DPL < CPL or RPL of code segment's segment selector
        THEN
            segment selector register null selector;
    OD;
ESP ESP + SRC; (* release parameters from calling procedure's stack *)

```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector null.
	If the return instruction pointer is not within the return code segment limit
#GP(selector)	If the RPL of the return code segment selector is less than the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.

```
; File: subroutine.asm
;
; example of subroutines in assembly language.
```

```
%define STDOUT 1
%define SYSCALL_EXIT 1
%define SYSCALL_WRITE 4
```

```
SECTION .data ; initialized data section

msg1: db "Hello World", 10, 0 ; C-style \0 term. string

msg2: db "Good-bye, blue sky", 10, 0

SECTION .text ; Code section.
global _start ; let loader see entry point

_start: nop ; Entry point.
pstart: ; address for gdb

mov eax, msg1 ; print first string
call print

mov eax, msg2 ; print second string
call print

; final exit
;
pexit: mov eax, SYSCALL_EXIT ; exit function
mov ebx, 0 ; exit code, 0=normal
int 080h ; ask kernel to take over
```

```

; Subroutine print
; writes null-terminated string with address in eax
;
print:
    ; find \0 character and count length of string
    ;
    mov     edi, eax                ; use edi as index
    mov     edx, 0                 ; initialize count

count:  cmp     [edi], byte 0        ; null char?
        je     end_count
        inc   edx                 ; update index & count
        inc   edi
        jmp   short count

end_count:

    ; make syscall to write
    ; edx already has length of string
    ;
    mov     ecx, eax                ; Arg3: addr of message
    mov     eax, SYSCALL_WRITE     ; write function
    mov     ebx, STDOUT            ; Arg1: file descriptor
    int     080h                  ; ask kernel to write
    ret

; end of subroutine

```

```
linux3% gdb a.out
GNU gdb 19991004
Copyright 1998 Free Software Foundation, Inc.
```

```
(gdb) disas *pstart
```

```
Dump of assembler code for function pstart:
```

```
0x8048081 <pstart>:      mov     %eax,0x80490c0
0x8048086 <pstart+5>:    call   0x80480a1 <print>
0x804808b <pstart+10>:   mov     %eax,0x80490cd
0x8048090 <pstart+15>:  call   0x80480a1 <print>
0x8048095 <pexit>:      mov     %eax,0x1
0x804809a <pexit+5>:    mov     %ebx,0x0
0x804809f <pexit+10>:   int    0x80
```

```
End of assembler dump.
```

```
(gdb) break *pstart
```

```
Breakpoint 1 at 0x8048081
```

```
(gdb) break *print
```

```
Breakpoint 2 at 0x80480a1
```

```
(gdb) run
```

```
Starting program: /afs/umbc.edu/users/c/h/chang/home/asm/sub/a.out
```

```
Breakpoint 1, 0x8048081 in pstart ()
```

```
(gdb) print/x $esp
```

```
$1 = 0x7ffffb90
```

```
(gdb) cont
```

```
Continuing.
```

```
Breakpoint 2, 0x80480a1 in print ()
```

```
(gdb) print/x $esp
```

```
$2 = 0x7ffffb8c
```

```
(gdb) x/1wx $esp
```

```
0x7ffffb8c:      0x0804808b
```

```
(gdb) cont
Continuing.
Hello World
```

```
Breakpoint 2, 0x80480a1 in print ()
```

```
(gdb) print/x $eax
```

```
$3 = 0x80490cd
```

```
(gdb) x/20cb &msg2
```

```
0x80490cd <msg2>:          71 'G'  111 'o'  111 'o'  100 'd'  45 '-'  98
'b'  121 'y'  101 'e'
```

```
0x80490d5 <msg2+8>:       44 ','  32 ' '  98 'b'  108 'l'  117 'u'  101
'e'  32 ' '  115 's'
```

```
0x80490dd <msg2+16>:     107 'k'  121 'y'  10 '\n'  0 '\000'
```

```
(gdb) x/1wx $esp
```

```
0x7fffffb8c:          0x08048095
```

```
(gdb) cont
```

```
Continuing.
```

```
Good-bye, blue sky
```

```
Program exited normally.
```

```
(gdb) quit
```

```
linux3% exit
```

```

; File: recursive.asm
;
; example of subroutines in assembly language.

%define STDOUT 1
%define SYSCALL_EXIT 1
%define SYSCALL_WRITE 4

SECTION .data ; initialized data section

msg1: db "Hello World", 10, 0 ; C-style \0 terminated
string

msg2: db 10, "Good-bye, blue sky", 10, 0

char: db 0, 0 ; single char followed by \0

SECTION .text ; Code section.
global _start ; let loader see entry point

_start: nop ; Entry point.
pstart: ; address for gdb

mov eax, msg1 ; print first string
call print

mov al, '5'
call recurse

mov eax, msg2 ; print second string
call print

; final exit
;
pexit: mov eax, SYSCALL_EXIT ; exit function
mov ebx, 0 ; exit code, 0=normal
int 080h ; ask kernel to take over

```

```

; A recursive subroutine
; counts down to '0'
; parameter stored in register al

recurse:
    cmp     al, '0'           ; don't go below '0'
    jae    rcont             ; go back
    ret

rcont:  push    ax           ; save al
        dec     al           ; param for recursive call
        call   recurse      ; recursively count down
        pop     ax          ; restore count
        mov    [char], al   ; prepare string for printing
        mov    eax, char    ; param for print subroutine
        call   print
        ret

; Subroutine print
; writes null-terminated string with address in eax
;
print:
    ; find \0 character and count length of string
    ;
    mov     edi, eax         ; use edi as index
    mov     edx, 0          ; initialize count

count:  cmp     [edi], byte 0  ; null char?
        je     end_count
        inc   edx           ; update index & count
        inc   edi
        jmp   short count

end_count:
    ; make syscall to write
    ; edx already has length of string
    ;
    mov     ecx, eax        ; Arg3: addr of message
    mov     eax, SYSCALL_WRITE ; write function
    mov     ebx, STDOUT    ; Arg1: file descriptor
    int     080h          ; ask kernel to write
    ret

; end of subroutine
linux3% nasm -f elf recurse.asm

```

```
linux3% nasm -f elf recurse.asm  
linux3% ld recurse.o  
linux3%
```

```
linux3% a.out  
Hello World  
012345  
Good-bye, blue sky  
linux3%
```

Linux/gcc/i386 Function Call Convention

- **Parameters pushed right to left on the stack**
 - ◇ first parameter on top of the stack
- **Caller saves EAX, ECX, EDX if needed**
 - ◇ these registers will probably be used by the callee
- **Callee saves EBX, ESI, EDI**
 - ◇ there is a good chance that the callee does not need these
- **EBP used as index register for parameters, local variables, and temporary storage**
- **Callee must restore caller's ESP and EBP**
- **Return value placed in EAX**

A typical stack frame for the function call:

```
int foo (int arg1, int arg2, int arg3) ;
```

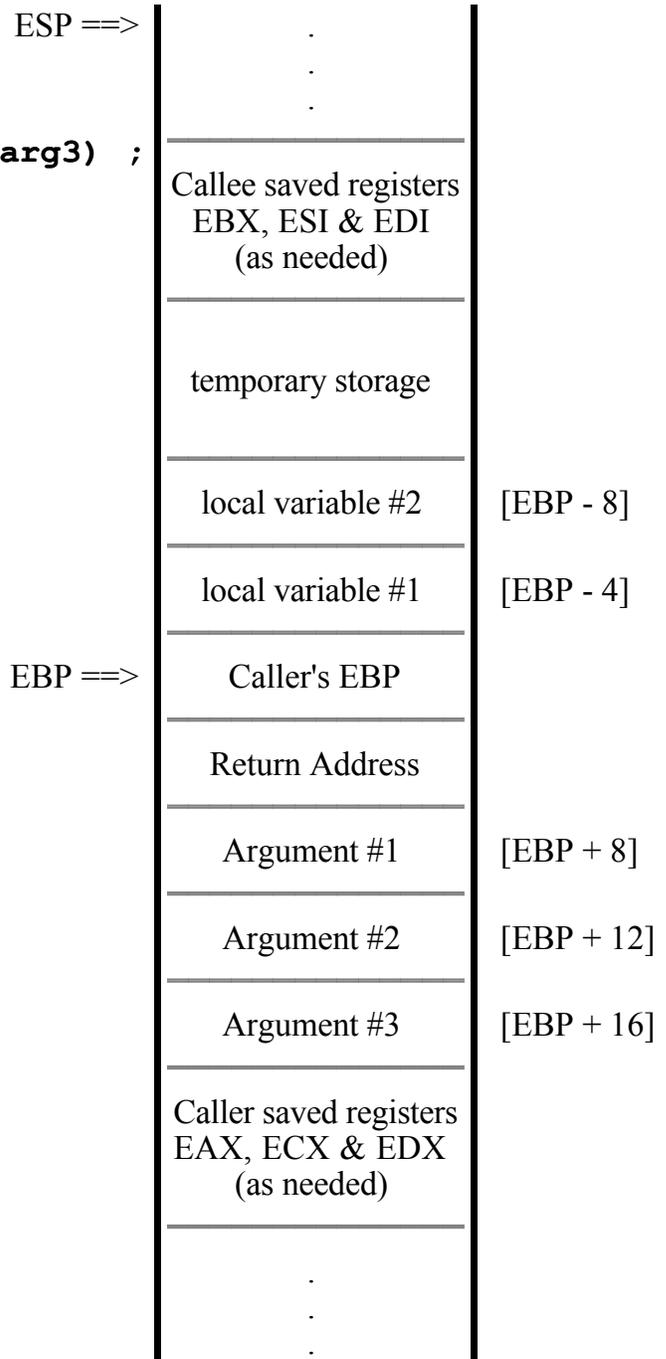


Fig. 1

Next Time

- **More C function call examples**
- **Project 4**