Ethical Hacking

Assembly Language
Tutorial

Number Systems

- Memory in a computer consists of numbers
- Computer memory does not store these numbers in decimal (base 10)
- Because it greatly simplifies the hardware, computers store all information in a binary (base 2) format.

Base 10 System

- Base 10 numbers are composed of 10 possible digits (0-9)
- Each digit of a number has a power of 10 associated with it based on its position in the number
- For example:
  - 234 = 2 × 102 + 3 × 101 + 4 × 100

Base 2 System

- Base 2 numbers are composed of 2 possible digits (0 and 1)
- Each digit of a number has a power of 2 associated with it based on its position in the number. (A single binary digit is called a bit.)
- For example:
  - 110012 = 1 × 24 + 1 × 23 + 0 × 22 + 0 × 21 + 1 × 20

    = 16 + 8 + 1

    = 25

Decimal 0 to 15 in Binary

Decimal	Binary		Decimal	Binary
0	0000		8	1000
1	0001		9	1001
2	0010		10	1010
3	0011		11	1011
4	0100		12	1100
5	0101		13	1101
6	0110		14	1110
7	0111		15	1111

Binary Addition (C stands for Canary)

No previous carry				Previous carry			
0	0	1	1	0	0	1	1
+0	+1	+0	+1	+0	+1	+0	+1
0	1	1	0	1	0	0	1
			c		c	c	c

EC-Council

Hexadecimal Number

- Hexadecimal numbers use base 16. Hexadecimal (or hex for short) can be used as a shorthand for binary numbers.
- Hex has 16 possible digits. This creates a problem since there are no symbols to use for these extra digits after 9.
- By convention, letters are used for these extra digits. The 16 hex digits are 0-9 then A, B, C, D, E and F.
- The digit A is equivalent to 10 in decimal, B is 11, etc. Each digit of a hex number has a power of 16 associated with it.

Hex Example

- $2BD_{16} = 2 \times 16^2 + 11 \times 16^1 + 13 \times 16^0$
- $= 512 + 176 + 13$
- $= 701$

EC-Council

Hex Conversion

- To convert a hex number to binary, simply convert each hex digit to a 4-bit binary number.
- For example, 24D16 is converted to 0010 0100 11012.
- Note that the leading zeros of the 4-bits are important!
- If the leading zero for the middle digit of 24D16 is not used the result is wrong.
- Example:
- 110 0000 0101 1010 0111 11102 (Binary)
- 6 0 5 A 7 E (Base 16)

EC-Council

nibble

- A 4-bit number is called a nibble
- Thus each hex digit corresponds to a nibble
- Two nibbles make a byte and so a byte can be represented by a 2-digit hex number
- A byte's value ranges from 0 to 11111111 in binary, 0 to FF in hex and 0 to 255 in decimal

Computer memory

- The basic unit of memory is a byte
- A computer with 32 megabytes of memory can hold roughly 32 million bytes of information
- Each byte in memory is labeled by a unique number known as its address

Address	0	1	2	3	4	5	6	7
Memory	2A	45	B8	20	8F	CD	12	2E

Figure 1.4: Memory Addresses

Characters Coding

- All data in memory is numeric. Characters are stored by using a character code that maps numbers to characters
- One of the most common character codes is known as ASCII (American Standard Code for Information Interchange)
- A new, more complete code that is supplanting ASCII is Unicode
- One key difference between the two codes is that ASCII uses one byte to encode a character, but Unicode uses two bytes (or a word) per character
- For example, ASCII maps the byte 4116 (6510) to the character capital A; Unicode maps the word 004116

ASCII and UNICODE

- Since ASCII uses a byte, it is limited to only 256 different characters
- Unicode extends the ASCII values to words and allows many more characters to be represented
- This is important for representing characters for all the languages of the world

CPU

- The Central Processing Unit (CPU) is the physical device that performs instructions
- The instructions that CPUs perform are generally very simple
- Instructions may require the data they act on to be in special storage locations in the CPU itself called registers
- The CPU can access data in registers much faster than data in memory
- However, the number of registers in a CPU is limited, so the programmer must take care to keep only currently used data in registers

Machine Language

- The instructions a type of CPU executes make up the CPU's machine language
- Machine programs have a much more basic structure than higher level languages
- Machine language instructions are encoded as raw numbers, not in friendly text formats
- A CPU must be able to decode an instruction's purpose very quickly to run efficiently
- Programs written in other languages must be converted to the native machine language of the CPU to run on the computer

Compilers

- A compiler is a program that translates programs written in a programming language into the machine language of a particular computer architecture
- In general, every type of CPU has its own unique machine language
- This is one reason why programs written for a Mac can not run on an IBM-type PC

Clock Cycle

- Computers use a clock to synchronize the execution of the instructions
- The clock pulses at a fixed frequency (known as the clock speed)
- When you buy a 1.5 GHz computer, 1.5 GHz is the frequency of this clock
- The clock does not keep track of minutes and seconds
- It simply beats at a constant rate. The electronics of the CPU uses the beats to perform their operations
- GHz stands for gigahertz or one billion cycles per second
- A 1.5 GHz CPU has 1.5 billion clock pulses per second

Original Registers

- General purpose registers. They are used in many of the data movement and arithmetic instructions
  - AX, BX, CX and DX
- Index registers. They are often used as pointers
  - SI and DI
- BP and SP registers are used to point to data in the machine language stack and are called the Base Pointer and Stack Pointer
- CS, DS, SS and ES registers are segment registers. They denote what memory is used for different parts of a program
- CS stands for Code Segment, DS for Data Segment, SS for Stack Segment and ES for Extra Segment
- ES is used as a temporary segment register

Instruction Pointer

- The Instruction Pointer (IP) register is used with the CS register to keep track of the address of the next instruction to be executed by the CPU.
- Normally, as an instruction is executed, IP is advanced to point to the next instruction in memory

EC-Council

Pentium Processor

- This CPU greatly enhanced the original registers
- First, it extends many of the registers to hold 32-bits (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP) and adds two new 16-bit registers FS and GS
- It also adds a new 32-bit protected mode
- In this mode, it can access up to 4 gigabytes
- Programs are again divided into segments, but now each segment can also be up to 4 gigabytes in size!

Interrupts

- Sometimes the ordinary flow of a program must be interrupted to process events that require prompt response
- The hardware of a computer provides a mechanism called interrupts to handle these events
- For example, when a mouse is moved, the mouse hardware interrupts the current program to handle the mouse movement (to move the mouse cursor, etc.)
- Interrupts cause control to be passed to an interrupt handler

Interrupt handler

- Interrupt handlers are routines that process the interrupt
- Each type of interrupt is assigned an integer number
- At the beginning of physical memory, a table of interrupt vectors resides that contain the segmented addresses of the interrupt handlers
- The number of interrupt is essentially an index into this table

External interrupts and Internal interrupts

- External interrupts are raised from outside the CPU. (The mouse is an example of this type.) Many I/O devices raise interrupts (e.g., keyboard, timer, disk drives, CD-ROM and sound cards).
- Internal interrupts are raised from within the CPU, either from an error or the interrupt instruction.
- Error interrupts are also called traps. Interrupts generated from the interrupt instruction are called software interrupts

Handlers

- Many interrupt handlers return control back to the interrupted program when they finish
- They restore all the registers to the same values they had before the interrupt occurred
- Thus, the interrupted program runs as if nothing happened (except that it lost some CPU cycles)
- Traps generally do not return. Often they abort the program.

Machine Language

- Every type of CPU understands its own machine language
- Instructions in machine language are numbers stored as bytes in memory
- Each instruction has its own unique numeric code called its operation code or opcode for short
- The 80x86 processor's instructions vary in size. The opcode is always at the beginning of the instruction
- Many instructions also include data (e.g., constants or addresses) used by the instruction

Machine Language

- Machine language is very difficult to program in directly
- Deciphering the meanings of the numerical-coded instructions is tedious for humans
- For example, the instruction that says to add the EAX and EBX registers together and store the result back into EAX is encoded by the following hex codes:
  - **`03 C3`**
- This is hardly obvious. Fortunately, a program called an assembler can do this tedious work for the programmer

Assembly Language

- An assembly language program is stored as text (just as a higher level language program)
- Each assembly instruction represents exactly one machine instruction. For example, the addition instruction would be represented in assembly language as:
  - **`add eax, ebx`**
- Here the meaning of the instruction is much clearer than in machine code
- The word add is a mnemonic for the addition instruction.
- The general form of an assembly instruction is:
  - mnemonic operand(s)

Assembler

- An assembler is a program that reads a text file with assembly instructions and converts the assembly into machine code
- Compilers are programs that do similar conversions for high-level programming languages
- An assembler is much simpler than a compiler
- Every assembly language statement directly represents a single machine instruction
- High-level language statements are much more complex and may require many machine instructions

Assembly Language Vs High-level Language

- Difference between assembly and high-level languages is that since every different type of CPU has its own machine language, it also has its own assembly language
- Porting assembly programs between different computer architectures is much more difficult than in a high-level language

Assembly Language Compilers

- Netwide Assembler or NASM (freely available off the Internet)
- Microsoft's Assembler (MASM)
- Borland's Assembler (TASM)
- There are some differences in the assembly syntax for MASM, TASM and NASM

Instruction operands

- Machine code instructions have varying number and type of operands; however, in general, each instruction itself will have a fixed number of oper-ands (0 to 3).
- Operands can have the following types:
  - **register**: These operands refer directly to the contents of the CPU's registers
  - **memory**: These refer to data in memory. The address of the data may be a constant hardcoded into the instruction or may be computed using
  - **values of registers**. Address are always offsets from the beginning of a segment.
  - **immediate**: These are fixed values that are listed in the instruction itself. They are stored in the instruction itself (in the code segment), not in the data segment.
  - **implied**: There operands are not explicitly shown. For example, the increment instruction adds one to a register or memory. The one is implied.

MOV instruction

- The most basic instruction is the MOV instruction
- It moves data from one location to another (like the assignment operator in a high-level language)
- It takes two operands:
  - **`mov dest, src`**
- The data specified by src is copied to dest
- One restriction is that both operands may not be memory operands
- The operands must also be the same size
- The value of AX can not be stored into BL

MOV instruction Example

- **`mov eax, 3`**
  - store 3 into EAX register (3 is immediate operand)
- **`mov bx, ax`**
  - store the value of AX into the BX register

ADD instruction

- The ADD instruction is used to add integers.
- **`add eax, 4`**
  - eax = eax + 4
- **`add al, ah`**
  - al = al + ah

SUB instruction

- The SUB instruction subtracts integers.
- **`sub bx, 10`**
  - bx = bx - 10
- **`sub ebx, edi`**
  - ebx = ebx - edi

INC and DEC instructions

- The INC and DEC instructions increment or decrement values by one
- **`inc ecx`**
  - ecx++
- **`dec dl`**
  - dl--

Directive

- Directive is an artifact of the assembler not the CPU
- They are generally used to either instruct the assembler to do something or inform the assembler of something
- They are not translated into machine code
- Common uses of directives are:
  - define constants
  - define memory to store data into
  - group memory into segments
  - conditionally include source code
  - include other files

preprocessor

- NASM code passes through a preprocessor just like C
- It has many of the same preprocessor commands as C
- NASM's preprocessor directives start with a % instead of a # as in C

equ directive

- The equ directive can be used to define a symbol
- Symbols are named constants that can be used in the assembly program
- The format is:
  - **`symbol equ value`**

%define directive

- This directive is similar to C's #define directive
- It is most commonly used to define constant macros just as in C
  - **`%define SIZE 100`**
  - **`mov eax, SIZE`**
- The above code defines a macro named SIZE and shows its use in a MOV instruction

Data directives

- Data directives are used in data segments to define room for memory.
- There are two ways memory can be reserved.
  - The first way only defines room for data
  - The second way defines room and an initial value
- The first method uses one of the RESX directives. The X is replaced with a letter that determines the size of the object (or objects) that will be stored
- The second method (that defines an initial value, too) uses one of the DX directives
- The X letters are the same as those in the RESX directives

Labels

- Labels allow one to easily refer to memory locations in code
- **Examples:**
  - `L1 db 0`
    - byte labeled L1 with initial value 0
  - `L2 dw 1000`
    - word labeled L2 with initial value 1000
  - `L3 db 110101b`
    - byte initialized to binary 110101 (53 in decimal)
  - `L4 db 12h`
    - byte initialized to hex 12 (18 in decimal)
  - `L5 db 17o`
    - byte initialized to octal 17 (15 in decimal)
  - `L6 dd 1A92h`
    - double word initialized to hex 1A92
  - `L7 resb 1`
    - 1 uninitialized byte
  - `L8 db "A"`
    - byte initialized to ASCII code for A (65)
  - `L9 db 0, 1, 2, 3`
    - defines 4 bytes
  - `L10 db "w", "o", "r", 'd', 0`
    - defines a C string = "word"
  - `L11 db 'word', 0`
    - same as L10

Label []

- There are two ways that a label can be used. If a plain label is used, it is interpreted as the address (or offset) of the data
- If the label is placed inside square brackets ([]), it is interpreted as the data at the address
- You should think of a label as a pointer to the data and the square brackets dereferences the pointer just as the asterisk does in C

Example

- **`mov al, [L1]`**
  - copy byte at L1 into AL
- **`mov eax, L1`**
  - EAX = address of byte at L1
- **`mov [L1], ah`**
  - copy AH into byte at L1
- **`mov eax, [L6]`**
  - copy double word at L6 into EAX
- **`add eax, [L6]`**
  - EAX = EAX + double word at L6
- **`add [L6], eax`**
  - double word at L6 += EAX
- **`mov al, [L6]`**
  - copy first byte of double word at L6 into AL

Input and output

- Input and output are very system dependent activities
- It involves interfacing with the system's hardware
- High level languages, like C, provide standard libraries of routines that provide a simple, uniform programming interface for I/O
- Assembly languages provide no standard libraries
- They must either directly access hardware (which is a privileged operation in pro-tected mode) or use whatever low level routines that the operating system provides

C Interface

- It is very common for assembly routines to be interfaced with C
- One advantage of this is that the assembly code can use the standard C library I/O routines
- To use these routines, you must include a file with information that the assembler needs to use them
- To include a file in NASM, use the %include preprocessor directive
- The following line includes the file needed:
  - **`%include "asm_io.inc"`**

Call

- To use one of the print routines, you load EAX with the correct value and use a CALL instruction to invoke it
- The CALL instruction is equivalent to a function call in a high level language
- It jumps execution to another section of code, but returns back to its origin after the routine is over

Creating a Program

- Today, it is unusual to create a stand alone program written completely in assembly language
- Assembly is usually used to key certain critical routines
- It is much easier to program in a higher level language than in assembly
- Using assembly makes a program very hard to port to other platforms
- In fact, it is rare to use assembly at all

Why should anyone learn assembly at all?

1. Sometimes code written in assembly can be faster and smaller than compiler generated code
2. Assembly allows access to direct hardware features of the system that might be difficult or impossible to use from a higher level language
3. Learning to program in assembly helps to gain a deeper understanding of how computers work
4. Learning to program in assembly helps you understand better how compilers and high level languages like C work

First.asm

first.asm

```
; file: first.asm
; First assembly program. This program asks for two integers as
; input and prints out their sum.
;
; To create executable using djgpp:
; nasm -f coff first.asm
; gcc -o first first.o driver.c asm_io.o

%include "asm_io.inc"
;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output
;
prompt1 db    "Enter a number: ", 0       ; don't forget null terminator
prompt2 db    "Enter another number: ", 0
outmsg1 db    "You entered ", 0
outmsg2 db    " and ", 0
outmsg3 db    ", the sum of these is ", 0

;
; uninitialized data is put in the .bss segment
;
segment .bss
;
; These labels refer to double words used to store the inputs
;
```

```
input1  resd 1
input2  resd 1

;
; code is put in the .text segment
;
segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; setup routine
        pusha

        mov     eax, prompt1      ; print out prompt
        call    print_string

        call    read_int          ; read integer
        mov     [input1], eax     ; store into input1

        mov     eax, prompt2      ; print out prompt
        call    print_string

        call    read_int          ; read integer
        mov     [input2], eax     ; store into input2

        mov     eax, [input1]     ; eax = dword at input1
        add     eax, [input2]     ; eax += dword at input2
        mov     ebx, eax          ; ebx = eax

        dump_regs 1                ; print out register values
        dump_mem  2, outmsg1, 1    ; print out memory
```

```
;
; next print out result message as series of steps
;
        mov     eax, outmsg1
        call    print_string        ; print out first message
        mov     eax, [input1]
        call    print_int           ; print out input1
        mov     eax, outmsg2
        call    print_string        ; print out second message
        mov     eax, [input2]
        call    print_int           ; print out input2
        mov     eax, outmsg3
```

EC-Council

```
        call    print_string        ; print out third message
        mov     eax, ebx
        call    print_int           ; print out sum (ebx)
        call    print_nl            ; print new-line

        popa
        mov     eax, 0              ; return back to C
        leave
        ret
```

first.asm

EC-Council

Assembling the code

- The first step is to assembly the code
- From the command line, type:
  - **`nasm -f object-format first.asm`**
- where object-format is either coff , elf , obj or win32 depending on what C compiler will be used

Compiling the C code

- Compile the driver.c file using a C compiler
  - **`gcc -c driver.c`**
- The -c switch means to just compile, do not attempt to link yet
- This same switch works on Linux, Borland and Microsoft compilers as well

Linking the object files

- Linking is the process of combining the machine code and data in object files and library files together to create an executable file
- This process is complicated
- C code requires the standard C library and special startup code to run
- It is much easier to let the C compiler call the linker with the correct parameters, than to try to call the linker directly
  - `gcc -o first driver.o first.o asm io.o`
- This creates an executable called first.exe (or just first under Linux)

Understanding an assembly listing file

- The -l listing-file switch can be used to tell nasm to create a listing file of a given name
- This file shows how the code was assembled
- The first column in each line is the line number and the second is the offset (in hex) of the data in the segment
- The third column shows the raw hex values that will be stored

```
48 00000000 456E7465722061206E-     prompt1 db    "Enter a number: ", 0
49 00000009 756D6265723A2000
50 00000011 456E7465722061 6E6F-    prompt2 db    "Enter another number: ", 0
51 0000001A 74686572206E756D62-
52 00000023 65723A2000
```

Big and Little Endian Representation

- There are two popular methods of storing integers: big endian and little endian
- Big endian is the method that seems the most natural. The biggest (i.e. most significant) byte is stored first, then the next biggest, etc
- For example, the dword 00000004 would be stored as the four bytes 00 00 00 04
- IBM mainframes, most RISC processors and Motorola processors all use this big endian method
- Intel-based processors use the little endian method!
- Here the least significant byte is stored first
- 00000004 is stored in memory as 04 00 00 00
- This format is hardwired into the CPU and can not be changed

Skeleton File

skel.asm

```
%include "asm_io.inc"
segment .data
;
; initialized data is put in the data segment here
;

segment .bss
;
; uninitialized data is put in the bss segment
;

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; setup routine
        pusha

;
; code is put in the text segment. Do not modify the code before
; or after this comment.
;

        popa
        mov     eax, 0            ; return back to C
        leave
        ret
```

skel.asm

EC-Council

Working with Integers

- Integers come in two flavors: unsigned and signed
- Unsigned integers (which are non-negative) are represented in a very straightforward binary manner
- The number 200 as an one byte unsigned integer would be represented as by 11001000 (or C8 in hex)

Signed integers

- Signed integers (which may be positive or negative) are represented in a more complicated ways
- For example, consider –56. +56 as a byte would be represented by 00111000
- On paper, one could represent –56 as –111000, but how would this be represented in a byte in the computer's memory
- How would the minus sign be stored?
- There are three general techniques that have been used to represent signed integers in computer memory
- All of these methods use the most significant bit of the integer as a sign bit
- This bit is 0 if the number is positive and 1 if negative

Signed Magnitude

- The first method is the simplest and is called signed magnitude. It represents the integer as two parts
- The first part is the sign bit and the second is the magnitude of the integer
- So 56 would be represented as the byte 00111000 (the sign bit is underlined) and –56 would be 10111000

EC-Council

Two's Compliment

- Signed Magnitude methods described were used on early computers
- Modern computers use a method called two's complement representation
- The two's complement of a number is found by the following two steps:
  - 1. Find the one's complement of the number
  - 2. Add one to the result of step 1
- Here's an example using 00111000 (56)
  - First the one's complement is computed: 11000111
  - Then one is added:
- `11000111`
- `      + 1`
- `11001000`

If statements

- The following pseudo-code:
  - `if ( condition )`
    - `then block ;`
  - `else`
    - `else block ;`
- Could be implemented as:
  - `1 ; code to set FLAGS`
  - `2 jxx else_block ; select xx so that branches if condition false`
  - `3 ; code for then block`
  - `4 jmp endif`
  - `5 else_block:`
  - `6 ; code for else block`
  - `7 endif:`

Do while loops

- The do while loop is a bottom tested loop:
  - do
  - {
  - body of loop ;
  - } while (condition);
- This could be translated into:
  - 1 do:
  - 2 ; body of loop
  - 3 ; code to set FLAGS based on condition
  - 4 jxx do ; select xx so that branches if true

Example: Finding Prime Numbers

- This is a program that finds prime numbers
- Prime numbers are evenly divisible by only 1 and themselves
- There is no formula for doing this
- The basic method this program uses is to find the factors of all odd numbers3 below a given limit
- If no factor can be found for an odd number, it is prime

Finding Prime Numbers

```
unsigned guess;   /* current guess for prime     */
unsigned factor;  /* possible factor of guess    */
unsigned limit;   /* find primes up to this value */

 printf ("Find primes up to: ");
scanf("%u", &limit);
 printf ("2\n");   /* treat first two primes as  */
 printf ("3\n");   /* special case               */
guess = 5;         /* initial guess */
while ( guess <= limit ) {
   /* look for a factor of guess */
   factor = 3;
   while ( factor*factor < guess &&
             guess % factor != 0 )
     factor += 2;
   if ( guess % factor != 0 )
      printf ("%d\n", guess);
   guess += 2;   /* only look at odd numbers */
}
```

EC-Council

Code 1

prime.asm

```
%include "asm_io.inc"
segment .data
Message         db      "Find primes up to: ", 0

segment .bss
Limit           resd    1                   ; find primes up to this limit
Guess           resd    1                   ; the current guess for prime

segment .text
        global  _asm_main
_asm_main:
        enter   0,0                 ; setup routine
        pusha

        mov     eax, Message
        call    print_string
        call    read_int                ; scanf("%u", & limit );
        mov     [Limit], eax

```

EC-Council

Code 2

```
        mov     eax, 2                  ; printf("2\n");
        call    print_int
        call    print_nl
        mov     eax, 3                  ; printf("3\n");
        call    print_int
        call    print_nl

        mov     dword [Guess], 5        ; Guess = 5;
while_limit:                            ; while ( Guess <= Limit )
        mov     eax,[Guess]
        cmp     eax, [Limit]
        jnbe    end_while_limit         ; use jnbe since numbers are unsigned

        mov     ebx, 3                  ; ebx is factor = 3;
while_factor:
        mov     eax,ebx
        mul     eax                     ; edx:eax = eax*eax
        jo      end_while_factor        ; if answer won't fit in eax alone
        cmp     eax, [Guess]
        jnb     end_while_factor        ; if !(factor*factor < guess)
        mov     eax,[Guess]
        mov     edx,0
        div     ebx                     ; edx = edx:eax % ebx
        cmp     edx, 0
        je      end_while_factor        ; if !(guess % factor != 0)
```

Code 3

```
        add     ebx,2                   ; factor += 2;
        jmp     while_factor
end_while_factor:
        je      end_if                  ; if !(guess % factor != 0)
        mov     eax,[Guess]             ; printf("%u\n")
        call    print_int
        call    print_nl
end_if:
        add     dword [Guess], 2        ; guess += 2
        jmp     while_limit
end_while_limit:

        popa
        mov     eax, 0               ; return back to C
        leave
        ret
```

prime.asm

EC-Council

Indirect addressing

- Indirect addressing allows registers to act like pointer variables
- To indicate that a register is to be used indirectly as a pointer, it is enclosed in square brackets ([])
- For example:
  - `1 mov ax, [Data] ; normal direct memory addressing of a word`
  - `2 mov ebx, Data ; ebx = & Data`
  - `3 mov ax, [ebx] ; ax = *ebx`

Subprogram

- A subprogram is an independent unit of code that can be used from different parts of a program
- A subprogram is like a function in C
- A jump can be used to invoke the subprogram, but returning presents a problem
- If the subprogram is to be used by different parts of the program, it must return back to the section of code that invoked it
- The jump back from the subprogram can not be hard coded to a label

Simple Subprogram Example

sub1.asm

```
; file: sub1.asm
; Subprogram example program
%include "asm_io.inc"

segment .data
prompt1 db    "Enter a number: ", 0       ; don't forget null terminator
prompt2 db    "Enter another number: ", 0
outmsg1 db    "You entered ", 0
outmsg2 db    " and ", 0
outmsg3 db    ", the sum of these is ", 0

segment .bss
input1  resd 1
input2  resd 1

segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; setup routine
        pusha

        mov     eax, prompt1      ; print out prompt
        call    print_string

        mov     ebx, input1       ; store address of input1 into ebx
```

```
        mov     ecx, ret1         ; store return address into ec
        jmp     short get_int     ; read integer
ret1:
        mov     eax, prompt2      ; print out prompt
        call    print_string

        mov     ebx, input2
        mov     ecx, $ + 7        ; ecx = this address + 7
        jmp     short get_int

        mov     eax, [input1]     ; eax = dword at input1
        add     eax, [input2]     ; eax += dword at input2
        mov     ebx, eax          ; ebx = eax

        mov     eax, outmsg1
        call    print_string      ; print out first message
        mov     eax, [input1]
        call    print_int         ; print out input1
        mov     eax, outmsg2
        call    print_string      ; print out second message
        mov     eax, [input2]
        call    print_int         ; print out input2
        mov     eax, outmsg3
        call    print_string      ; print out third message
        mov     eax, ebx
        call    print_int         ; print out sum (ebx)
        call    print_nl          ; print new-line
```

```
        popa
        mov     eax, 0              ; return back to C
        leave
        ret
; subprogram get_int
; Parameters:
;   ebx - address of dword to store integer into
;   ecx - address of instruction to return to
; Notes:
;   value of eax is destroyed
get_int:
        call    read_int
        mov     [ebx], eax          ; store input into memory
        jmp     ecx                 ; jump back to caller
```

sub1.asm

EC-Council

The Stack

- Many CPU's have built-in support for a stack
- A stack is a Last-In First-Out (LIFO) list
- The stack is an area of memory that is organized in this fashion
- The PUSH instruction adds data to the stack and the POP instruction removes data
- The data removed is always the last data added

The SS segment

- The SS segment register specifies the segment that contains the stack (usually this is the same segment data is stored into)
- The ESP register contains the address of the data that would be removed from the stack
- This data is said to be at the top of the stack
- Data can only be added in double word units
- The PUSH instruction inserts a double word1 on the stack by subtracting 4 from ESP and then stores the double word at [ESP]
- The POP instruction reads the double word at [ESP] and then adds 4 to ESPESP is initially 1000H

ESP

```
push    dword 1     ; 1 stored at 0FFCh, ESP = 0FFCh
push    dword 2     ; 2 stored at 0FF8h, ESP = 0FF8h
push    dword 3     ; 3 stored at 0FF4h, ESP = 0FF4h
pop     eax         ; EAX = 3, ESP = 0FF8h
pop     ebx         ; EBX = 2, ESP = 0FFCh
pop     ecx         ; ECX = 1, ESP = 1000h
```

The Stack Usage

- The stack can be used as a convenient place to store data temporarily
- It is also used for making subprogram calls, passing parameters and local variables

The CALL and RET Instructions

- The 80x86 provides two instructions that use the stack to make calling subprograms quick and easy
- The CALL instruction makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack
- The RET instruction pops off an address and jumps to that address

Passing parameters on the stack

- Parameters to a subprogram may be passed on the stack
- They are pushed onto the stack before the CALL instruction
- Just as in C, if the parameter is to be changed by the subprogram, the address of the data must be passed, not the value
- If the parameter's size is less than a double word, it must be converted to a double word before being pushed
- The parameters on the stack are not popped off by the subprogram, instead they are accessed from the stack itself

Stack Data

- This is how the stack looks when a subprogram is called

ESP + 4	Parameter
ESP	Return address

ESP + 8	Parameter
ESP + 4	Return address
ESP	subprogram data

General subprogram form

```
subprogram_label:
        push    ebp             ; save original EBP value on stack
        mov     ebp, esp        ; new EBP = ESP
; subprogram code
        pop     ebp             ; restore original EBP value
        ret
```

ESP + 8	EBP + 8	Parameter
ESP + 4	EBP + 4	Return address
ESP	EBP	saved EBP

Sample subprogram call

```
      push   dword 1        ; pass 1 as parameter
      call   fun
      add    esp, 4         ; remove parameter from stack
```

Example

sub3.asm

```
%include "asm_io.inc"

segment .data
sum     dd   0

segment .bss
input   resd 1

;
; pseudo-code algorithm
; i = 1;
; sum = 0;
; while( get_int(i, &input), input != 0 ) {
;    sum += input;
;    i++;
; }
; print_sum(num);
segment .text
        global  _asm_main
_asm_main:
        enter   0,0               ; setup routine
        pusha

        mov     edx, 1            ; edx is 'i' in pseudo-code
while_loop:
        push    edx               ; save i on stack
        push    dword input       ; push address on input on stack
        call    get_int
        add     esp, 8            ; remove i and &input from stack
```

EC-Council

```

        mov     eax, [input]
        cmp     eax, 0
        je      end_while

        add     [sum], eax          ; sum += input

        inc     edx
        jmp     short while_loop

end_while:
        push    dword [sum]         ; push value of sum onto stack
        call    print_sum
        pop     ecx                 ; remove [sum] from stack

        popa
        leave
        ret
```

EC-Council

```
; subprogram get_int
; Parameters (in order pushed on stack)
;   number of input (at [ebp + 12])
;   address of word to store input into (at [ebp + 8])
; Notes:
;   values of eax and ebx are destroyed
segment .data
prompt  db      ") Enter an integer number (0 to quit): ", 0

segment .text
get_int:
        push    ebp
        mov     ebp, esp

        mov     eax, [ebp + 12]
        call    print_int

        mov     eax, prompt
        call    print_string

        call    read_int
        mov     ebx, [ebp + 8]
        mov     [ebx], eax          ; store input into memory
```

```
        pop     ebp
        ret                             ; jump back to caller

; subprogram print_sum
; prints out the sum
; Parameter:
;   sum to print out (at [ebp+8])
; Note: destroys value of eax
;
segment .data
result  db      "The sum is ", 0

segment .text
print_sum:
        push    ebp
        mov     ebp, esp

        mov     eax, result
        call    print_string

        mov     eax, [ebp+8]
        call    print_int
        call    print_nl

        pop     ebp
        ret
```

sub3.asm

Local variables on the stack

- The stack can be used as a convenient location for local variables
- This is exactly where C stores normal (or automatic in C lingo) variables
- Using the stack for variables is important if you wish subprograms to be reentrant

EC-Council

General subprogram form with local variables

```
subprogram_label:
      push   ebp                ; save original EBP value on stack
      mov    ebp, esp           ; new EBP = ESP
      sub    esp, LOCAL_BYTES   ; = # bytes needed by locals
; subprogram code
      mov    esp, ebp           ; deallocate locals
      pop    ebp                ; restore original EBP value
      ret
```

Example: C version of sum

```
void calc_sum( int n, int * sump )
{
  int i, sum = 0;

  for( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

Example: Assembly version of sum

```
cal_sum:
        push    ebp
        mov     ebp, esp
        sub     esp, 4                  ; make room for local sum

        mov     dword [ebp - 4], 0      ; sum = 0
        mov     ebx, 1                  ; ebx (i) = 1
for_loop:
        cmp     ebx, [ebp+12]           ; is i <= n?
        jnle    end_for

        add     [ebp-4], ebx            ; sum += i
        inc     ebx
        jmp     short for_loop

end_for:
        mov     ebx, [ebp+8]            ; ebx = sump
        mov     eax, [ebp-4]            ; eax = sum
        mov     [ebx], eax              ; *sump = sum;

        mov     esp, ebp
        pop     ebp
        ret
```

Multi-module program

- Multi-module program is one composed of more than one object file.
- They consisted of the C driver object file and the assembly object file (plus the C library object files)
- The linker combines the object files into a single executable program
- The linker must match up references made to each label in one module (i.e. object file) to its definition in another module
- In order for module A to use a label defined in module B, the extern directive must be used
- After the extern directive comes a comma delimited list of labels
- The directive tells the assembler to treat these labels as external to the module

Saving registers

- First, C assumes that a subroutine maintains the values of the following registers: EBX, ESI, EDI, EBP, CS, DS, SS, ES
- This does not mean that the subroutine can not change them internally
- It means that if it does change their values, it must restore their original values before the subroutine returns
- The EBX, ESI and EDI values must be unmodified because C uses these registers for register variables
- Usually the stack is used to save the original values of these registers

Stack inside printf Statement

EBP + 12	value of **x**
EBP + 8	address of format string
EBP + 4	Return address
EBP	saved EBP

Labels of functions

- Most C compilers prepend a single underscore ( ) character at the beginning of the names of functions and global/ static variables
- For example, a function named f will be assigned the label f
- If this is to be an assembly routine, it must be labelled f, not f
- The Linux gcc compiler does not prepend any character
- Under Linux ELF executables, one simply would use the label f for the C function f

Calculating addresses of local variables

- Consider the case of passing the address of a variable (let's call it x) to a function (let's call it foo)
- If x is located at EBP – 8 on the stack, one cannot just
  - use: `mov eax, ebp - 8`
- Why? The value that MOV stores into EAX must be computed by the assembler (that is, it must in the end be a constant)
- There is an instruction that does the desired calculation. It is called LEA (for Load Effective Address)
- The following would calculate the address of x and store it into EAX:
  - `lea eax, [ebp - 8]`

EC-Council

- End of Slides