IA-32

Intel 32-bit Assembly Programming

Intel 32-bit Architecture: IA-32

Basic architectural plan for 32-bit Intel processors

- 1985 – today
- From 386 to Core, Xeon, and Centrino

Requirements

- Backward compatible with 8086, 80186, and 80286
- 32-bit integer
  - General registers extended to 32 bits
    - `EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP`
- 32-bit physical address
  - 2^{32} Bytes = 4 GB of addressable memory
- Expanded code set (32-bit instructions)
- Hardware support for modern operating system (Unix, Windows)

IA-32 introduced in 1985

- 386 was first IA-32 processor with full Unix implementation

Operating Modes

Real Mode
Start-up mode
8086 features

↓

Protected Mode
Full IA-32 features

All IA-32 processors initialize into real mode

- OS shifts processor into 32-bit protected mode

16-bit integers and address offsets
Sees `AX, BX, CX, DX, SI, DI, BP, SP, IP`
20-bit physical address (access to lowest 1 MB of RAM)
8086 interrupts

Windows/Linux/Unix/Mac OS run in protected mode

32-bit integers and physical addresses
Sees `EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP`
Hardware support for OS

- Task management
- Advanced segmentation model
- Virtual memory and paging
- Complex interrupt system

IA-32 General Purpose Registers

31 … 16	15 … 8	7 … 0	Register	Name
	AH	AL	EAX	Accumulator
	BH	BL	EBX	Base
	CH	CL	ECX	Counter
	DH	DL	EDX	Data

31 … 16	15 … 0	Register	Name
	SP	ESP	stack pointer
	BP	EBP	base pointer
	SI	ESI	source index
	DI	EDI	destination index
	IP	EIP	instruction pointer

Legal accesses

```
MOV AL,   12
MOV AH,   34
MOV AX,   3412
MOV EAX,  78563412
MOV SI,   3412
MOV ESI,  78563412
```

x86 Segment Registers

x86 segments are sections of physical memory

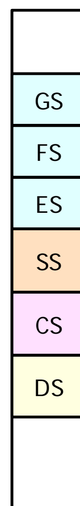
No one-to-one connection with program segments

Six defined memory pointers

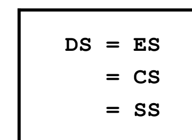
DS (name ~ Data Segment)
CS (name ~ Code Segment)
SS (name ~ Stack Segment)
ES (name ~ Extra Segment)
FS, GS — in IA-32 only

Six segment registers

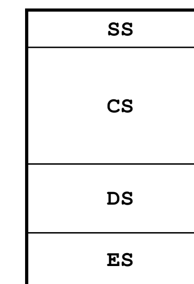
GS selector	GS
FS selector	FS
ES selector	ES
SS selector	SS
CS selector	CS
DS selector	DS
15 0	



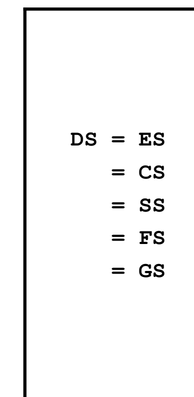
Typical Segment Register Usage



DOS *.com program
One 64 KB segment



DOS *.exe program
Four defined segments
Segment ≤ 64 KB

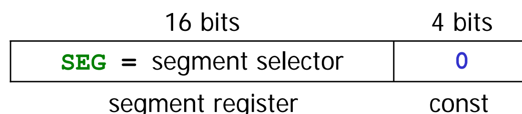


Linux software
One 4 GB segment
OS allocates memory to programs

Mapping Segment

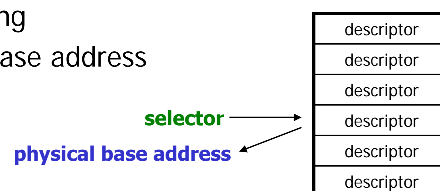
8086 segment mapping

SEG = 16-bit segment **SELECTOR** in segment register
SEG × **10h** = 20-bit physical base address



IA-32 segment mapping

SEG = 16-bit segment **SELECTOR** in segment register
Selector is index to descriptor table
Descriptor is table entry holding
32-bit or 64-bit physical base address
Segment size
Segment type
Segment access rights



descriptor
descriptor
descriptor
descriptor
descriptor
descriptor

IA-32 Offset

Offset

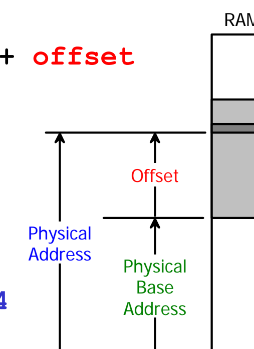
32-bit number
Combination of registers and immediate values
Offset ∈ {00000000, … , FFFFFFFF}
2^{32} possible offset values
Maximum segment = 2^{32} bytes = 4 GB

byte	FFFFFFFF
byte	…
byte	00000002
byte	00000001
Byte	00000000

Physical Address = **physical base address** + **offset**

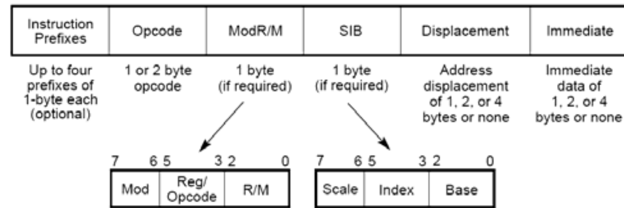
Example

Logical Address = **1234:11223344**
Segment selector = **1234** → descriptor table
Physical base address = **00000000**
Physical Address = **0** + **11223344** = **11223344**

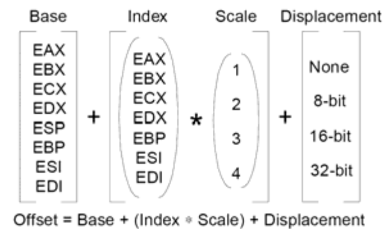


New Addressing Modes

New instruction encoding for IA-32



Flexible addressing in IA-32



Example of legal address on 386

```
mov eax,[eax+4*edi+11223344h]
```

On newer processors, `index = 1, 2, 3, 4, 8`

DOS Assembly Programming in IA-32

DOS *.exe and *.com programs run in real mode

- No protected mode features
- IA-32 processor can
  - Access 32-bit registers
  - Recognize IA-32 addressing modes

Forcing 32-bit instruction features

Prefix **66h** changes integer width to 32 bits

```
66B844332211          mov eax,0x11223344
B844332211
    B84433            mov ax,0x3344
    2211              and dl,[bx+di]
```

Prefix **67h** changes address to 32 bits

```
6667895D04            mov [ebp+0x4],ebx
```


Example 32-bit Code under DOS

```
      ORG 0x100
section .text
      mov eax,11223344h
      push eax
      pop ebx
      call disp32
      mov ebx,55667788h
      call disp32
      mov ax,4C00h
      int 21h

disp32:
      mov cx,08h   ; counter = 8
      mov ah,02h   ; DOS function is print byte
nibble:
      rol ebx,4    ; move most significant nibble to least
      mov dl,bl    ; load BL to print buffer
      and dl,0fh   ; zero upper nibble
      add dl,30h   ; ASCII digit range
      cmp dl,39h   ; is nibble in [A-F]
      jle go       ; if not > 9 print
      add dl,7h    ; if > 9 ASCII letter range
go:   int 21h      ; print the byte
      loop nibble  ; CX-- and continue
      mov dl, 0dh  ; CR
      int 21h
      mov dl, 0ah  ; LF
      int 21h
      ret
```


Disassemble

```
00000000  66B844332211  mov eax,0x11223344
00000006  6650          push eax
00000008  665B          pop ebx
0000000A  E80E00        call 0x1b
0000000D  66BB88776655  mov ebx,0x55667788
00000013  E80500        call 0x1b
00000016  B8004C        mov ax,0x4c00
00000019  CD21          int 0x21
0000001B  B90800        mov cx,0x8
0000001E  B402          mov ah,0x2
00000020  66C1C304      rol ebx,0x4
00000024  88DA          mov dl,bl
00000026  80E20F        and dl,0xf
00000029  80C230        add dl,0x30
0000002C  80FA39        cmp dl,0x39
0000002F  7E03          jng 0x34
00000031  80C207        add dl,0x7
00000034  CD21          int 0x21
00000036  E2E8          loop 0x20
00000038  B20D          mov dl,0xd
0000003A  CD21          int 0x21
0000003C  B20A          mov dl,0xa
0000003E  CD21          int 0x21
00000040  C3            ret
```

Example of 32-bit Address Overrides

```
                ORG 0x100
        section .data
                filename        db "test.txt",0
        section .bss
                handle          resw 1
        section .text
                mov eax,'abcd'
                mov ebx,'ABCD'
                mov ebp,2000h
                mov [ebp],eax
                mov [ebp+4],ebx
create:         mov dx,filename         ; point to file name
                mov cx,0h               ; default attributes
                mov ah,3ch              ; DOS create file
                int 21h                 ; DOS system call
                jc end                  ; stop on error
                mov [handle],ax         ; store file handle
write:          mov bx,[handle]         ; copy file handle to BX
                mov cx,8h               ; write 8 bytes to file
                mov edx,ebp             ; point EDX to buffer
                mov ah,40h              ; DOS write to file
                int 21h                 ; DOS system call
                jc end                  ; stop on error
close:          mov bx,[handle]         ; copy file handle to BX
                mov ah,3eh              ; DOS close file
                int 21h                 ; DOS system call
end:            mov ax,4C00h            ; return to DOS
                int 21h                 ; DOS system call
```


Assembler Output

```
00000100  66B861626364    mov eax,0x64636261
00000106  66BB41424344    mov ebx,0x44434241
0000010C  66BD00200000    mov ebp,0x2000
00000112  6667894500      mov [ebp+0x0],eax
00000117  66678955D04     mov [ebp+0x4],ebx
0000011C  BA4801          mov dx,0x148
0000011F  B90000          mov cx,0x0
00000122  B43C            mov ah,0x3c
00000124  CD21            int 0x21
00000126  721B            jc 0x43
00000128  A35401          mov [0x154],ax
0000012B  8B1E5401        mov bx,[0x154]
0000012F  B90800          mov cx,0x8
00000132  6689EA          mov edx,ebp
00000135  B440            mov ah,0x40
00000137  CD21            int 0x21
00000139  7208            jc 0x43
0000013B  8B1E5401        mov bx,[0x154]
0000013F  B43E            mov ah,0x3e
00000141  CD21            int 0x21
00000143  B8004C          mov ax,0x4c00
00000146  CD21            int 0x21
00000148  7465            jz 0xaf
0000014A  7374            jnc 0xc0
0000014C  2E7478          cs jz 0xc7
0000014F  7400            jz 0x51
```

```
C:\nasm\programs>type TEST.TXT
abcdABCD
```


NASM for Linux

NASM Package

nasm package available as source and executables

Typically **/usr/bin/nasm** and **/usr/bin/ndisasm**

Assembly

Linux uses ELF format for object and executable files

ELF = Executable and Linking Format

Typical header size = **180h - 3c0h** bytes

nasm –f elf [–o <output>] <filename>

Disassembly

View executable as 32-bit assembly code

ndisasm -b32 a.out | less

objdump -d a.out | less

Linking with gcc

Linking

Link object files with **gcc**

```
gcc [-options] <filename.o> [other_files.o]
```

Combine object files from various sources: assembly, C, C++, …

Produces **ELF32 i386 executable** file

Options

```
gcc -nostdlib <filename.o>
```

No Linux or C standard libraries linked

For pure assembly code without Linux or C function calls

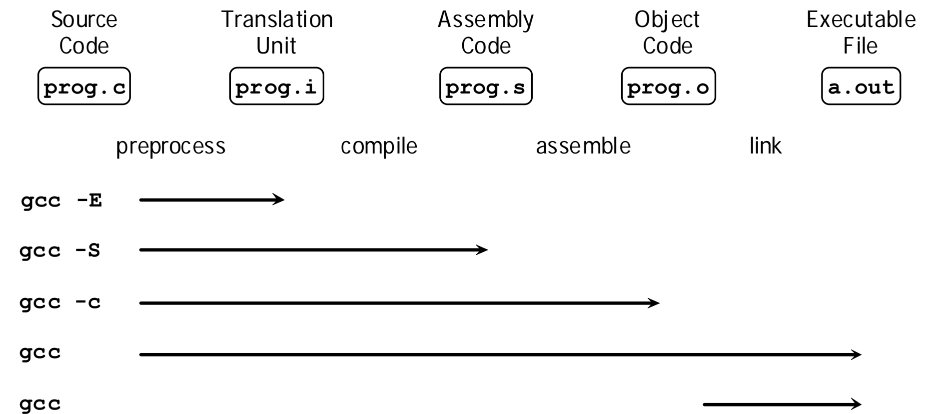
```
gcc -nostartfiles <filename.o>
```

No C standard libraries linked

For pure assembly code without standard C function calls

gcc Stages

Stages of Gnu C Compilation (gcc)



ELF Utilities

Object dump

```
objdump -d a.out | less
```

Displays disassembly of executable code (in **gas** format)

```
objdump -s a.out | less
```

Displays hexadecimal content of file with ASCII translation

Read ELF

```
readelf -h a.out
```

Displays summary of ELF header information

```
readelf -h a.out | grep address | awk '{print
    substr($4,7,3)}'
```

Displays 3-digit offset to start of executable code

Nasm disassembler

```
ndisasm -b32 -e`readelf -h a.out | grep address |
    awk '{print substr($4,7,3)}'`h a.out | less
```

Displays disassembly of executable code (in **nasm** format)

Disassembly Script

Edit file **nd**

Enter

```
ndisasm -b32 -e`readelf -h $1 | grep address | awk
    '{print substr($4,7,3)}'`h $1 | less
```

Run

```
~/nasm: nd a.out
```

General NASM Template for Linux

```
; uncomment next line to include C function calls
; extern standard_library_function_by_name
    section .data
; define initialized data structures here
    section .bss
; define uninitialized data structures here
; minimum allocation size is 1 dword = 32 bits
; cannot pass BSS pointers to all system calls
    section .text
    global _start          ; permits access to _start
_start:                    ; symbol for start EIP
    ; place code here
    ;
    mov eax, 1             ; Linux terminate code
    mov ebx, 0             ; or other exit code
    int 0x80               ; call Linux kernel
```


Simple Example — 1

```
section .data
        string db "this is a string",0
section .text
global _start
_start:
        mov eax, 0x11223344
        mov ebx, 0x55667788
        mov esi, string
        lodsb
;
mov eax, 1              ; Linux terminate code
mov ebx, 0              ; exit code
int 0x80                ; call Linux kernel
```


Simple Example — 2

Assemble

```
nasm -f elf ex1.asm
```

Link

```
gcc -nostdlib ex1.o
```

Rename

```
mv a.out ex1
```

Examine

```
nd ex1
```


Simple Example — 3

Dump — hex

```
objdump -s ex1

ex1:     file format elf32-i386

Contents of section .note.gnu.build-id:
 8048094 04000000 14000000 03000000 474e5500  ............GNU.
 80480a4 639f05fe 408b296b 771dec92 e3559231  c...@.)kw....U.1
 80480b4 91bc4e70                             ..Np
Contents of section .text:
 80480c0 b8443322 11bb8877 6655bedc 900408ac  .D3"...wfU......
 80480d0 b8010000 00bb0000 0000cd80           ............
Contents of section .data:
 80490dc 74686973 20697320 61207374 72696e67  this is a string
 80490ec 00
```

Simple Example — 4

Dump — gas disassemble

```
objdump -d ex1

ex1:     file format elf32-i386
Disassembly of section .text:

080480c0 <_start>:
 80480c0:       b8 44 33 22 11          mov    $0x11223344,%eax
 80480c5:       bb 88 77 66 55          mov    $0x55667788,%ebx
 80480ca:       be dc 90 04 08          mov    $0x80490dc,%esi
 80480cf:       ac                      lods   %ds:(%esi),%al
 80480d0:       b8 01 00 00 00          mov    $0x1,%eax
 80480d5:       bb 00 00 00 00          mov    $0x0,%ebx
 80480da:       cd 80                   int    $0x80
```


Simple Example — 5

nasm disassemble

```
ndisasm -b32 -e`readelf -h ex1 | grep address |
    awk '{print substr($4,7,3)}'`h ex1

00000000  B844332211        mov eax,0x11223344
00000005  BB88776655        mov ebx,0x55667788
0000000A  BEDC900408        mov esi,0x80490dc
0000000F  AC                lodsb
00000010  B801000000        mov eax,0x1
00000015  BB00000000        mov ebx,0x0
0000001A  CD80              int 0x80
0000001C  7468              jz 0x86
0000001E  69732069732061    imul esi,[ebx+0x20],dword
    0x61207369
```


Linux System Calls

Invoking standard C library call

Declare external library functions before data segment

Push parameters onto stack in proper order

Call function by name

Clean up stack after return

Invoking standard Linux system call

Similar to DOS system calls

Load parameters into **EAX, EBX, ECX, EDX**

Call Linux kernel with **INT 0x80**

References:

http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html

http://www.lxhp.in-berlin.de/

Exit

exit (terminate process)

```
EAX ← 1
EBX ← exit code
INT 0x80
```

Works like DOS version

```
MOV AH,4C
MOV AL,exit code
INT 0x21
```

Create File

```
creat
     EAX ← 8
     EBX ← pointer to ASCIIZ pathname
     ECX ← file permissions
       octal form (user/group/other)
       U/G/O = read + write + execute
         4 = read
         2 = write
         1 = ex
     INT 0x80

returns
     EAX ← integer file descriptor
```


Open File

```
open
     EAX ← 5
     EBX ← pointer to ASCIIZ pathname
     ECX ← file access mode
               0x00 = read only
               0x01 = write only
               0x02 = read/write
     EDX ← file permissions
     INT 0x80

returns
     EAX ← integer file descriptor
```


Write to File

```
write
     EAX ← 4
     EBX ← file descriptor
     ECX ← pointer to output buffer
     EDX ← number of bytes to write
     INT 0x80

returns
     EAX ← number of bytes actually written
```

Note

Write to screen using `stdout descriptor = 1`

Read from File

```
read
     EAX ← 3
     EBX ← file descriptor
     ECX ← pointer to input buffer
     EDX ← number of bytes to read
     INT 0x80

returns
     EAX ← number of bytes actually read
```

Note

Read from keyboard using `stdin descriptor = 0`

Close File

```
close
        EAX ← 6
        EBX ← file descriptor
        INT 0x80
```


Linux Call Example — 1

```
        section .data
path:   db "filename.txt",0
                                ; ASCIIZ pathname
str1:   db 'abcdefghijklmnopqrstuvwzyz',10,10
                                ; 10 = "\n"
len1    equ $-str1              ; len1 ← length of str1
str2:   db 'ABCDEFGHIJKLMNOPQRSTUVWZYZ',10,10
len2    equ $-str2              ; len2 ← length of str2
buff:   times 256 db 0          ; 256 zeros as buffer

        section .bss
desc:   resd 1                  ; d = dword = 32-bit integer
buff2:  resd 1                  ; minimum BSS allocation is dword
;buff: resb 256                 ; cannot pass BSS pointer to
                                ; read system call
```


Linux Call Example — 2

```
        section .text
        global _start
_start:
create: mov eax,8               ; create file system call
        mov ebx,path            ; pointer to pathname
        mov ecx,600o            ; access rights (o = octal)
        int 0x80                ; invoke Linux kernel
        mov [desc],eax          ; save file descriptor
        ;
write:  mov eax,4               ; write to file system call
        mov ebx,[desc]          ; file descriptor
        mov ecx,str1            ; pointer to string 1
        mov edx,len1            ; number of bytes to write
        int 0x80                ; invoke Linux kernel
        ;
        mov eax,4               ; write to file system call
        mov ebx,[desc]          ; file descriptor
        mov ecx,str2            ; pointer to string 2
        mov edx,len2            ; number of bytes to write
        int 0x80                ; invoke Linux kernel
```


Linux Call Example — 3

```
get:     mov eax,3         ; read from file system call
         mov ebx,0         ; file descriptor for stdin
         mov ecx,buff      ; pointer to input buffer (in data segment)
         mov edx,256       ; accept up to 265 bytes from stdin
         int 0x80          ; invoke Linux kernel
         ;
write2:  mov esi,buff      ; point ESI at input buffer from stdin
w2:      lodsb                     ; AL ← [ESI] , ESI ← ESI + 1
         and eax,0x000000ff        ; zero EAX except AL
         cmp al,0                  ; if AL = 0 then stop writing
         je close
         mov [buff2],eax           ; move EAX to memory at buff2 (BSS)
         mov eax,4                 ; write to file system call
         mov ebx,[desc]            ; file descriptor
         mov ecx,buff2             ; pointer to buffer
         mov edx,1                 ; number of bytes to write
         int 0x80                  ; invoke Linux kernel
         jmp w2                    ; continue
```

Linux Call Example — 4

```
close: mov eax,6
       mov ebx,[desc]
       int 0x80
       ;
exit:  mov eax,1
       mov ebx,0
       int 0x80
```


Linux Call Example — 5

```
~/nasm$ nasm -f elf ex2.asm
~/nasm$ gcc -nostdlib ex2.o
~/nasm$ mv a.out ex2
~/nasm$ ex2
I am writing this sentence.
~/nasm$ cat filename.txt
abcdefghijklmnopqrstuvwzyz

ABCDEFGHIJKLMNOPQRSTUVWZYZ

I am writing this sentence.
~/nasm$
```


Linux Call Example — 6

```
~/nasm$ ndisasm -e 180h -b32 ex2
```

```
00000000  B808000000    mov eax,0x8
00000005  BBA0950408    mov ebx,0x80495a0
0000000A  B900000000    mov ecx,0x0
0000000F  CD80          int 0x80
00000011  A3EC960408    mov [0x80496ec],eax
00000016  B804000000    mov eax,0x4
0000001B  8B1DEC960408  mov ebx,[0x80496ec]
00000021  B9AD950408    mov ecx,0x80495ad
00000026  BA1C000000    mov edx,0x1c
0000002B  CD80          int 0x80
0000002D  B804000000    mov eax,0x4
00000032  8B1DEC960408  mov ebx,[0x80496ec]
00000038  B9C9950408    mov ecx,0x80495c9
0000003D  BA1C000000    mov edx,0x1c
00000042  CD80          int 0x80
00000044  B803000000    mov eax,0x3
00000049  BB00000000    mov ebx,0x0
0000004E  B9E5950408    mov ecx,0x80495e5
00000053  BA00010000    mov edx,0x100
00000058  CD80          int 0x80
0000005A  BEE5950408    mov esi,0x80495e5
0000005F  AC            lodsb
00000060  25FF000000    and eax,0xff
00000065  3C00          cmp al,0x0
00000067  741E          jz 0x87
00000069  A3F0960408    mov [0x80496f0],eax
0000006E  B804000000    mov eax,0x4
00000073  8B1DEC960408  mov ebx,[0x80496ec]
00000079  B9F0960408    mov ecx,0x80496f0
0000007E  BA01000000    mov edx,0x1
00000083  CD80          int 0x80
00000085  EBD8          jmp short 0x5f
00000087  B806000000    mov eax,0x6
0000008C  8B1DEC960408  mov ebx,[0x80496ec]
00000092  CD80          int 0x80
00000094  B801000000    mov eax,0x1
00000099  BB00000000    mov ebx,0x0
0000009E  CD80          int 0x80
```


Using C Functions

```
      extern printf
      section .data
a:    dd    5
fmt:  db "a=%d, eax=%d", 10, 0   ; printf format string
                ; printf("a=%d, eax=%d\n", a, a+2)
      section .text
      global main              ; main replaces _start for C
main: mov eax, [a]             ; EAX ← value of a
      add eax, 2               ; EAX ← EAX + 2
      push eax                 ; value of a + 2
      push dword [a]           ; value of a
      push dword fmt           ; pointer to format string
      call printf              ; call C library function
      add esp, 12              ; clean up stack
                               ; (3 pushes of 4 bytes)
      mov eax,0                ; exit code to C environment
      ret                      ; return to C environment
```

Assembly and Linking

```
~/nasm$ nasm -f elf printf1.asm
~/nasm$ gcc printf1.o
~/nasm$ a.out
a=5, eax=7
~/nasm$
```


Another printf Example

```
        extern printf
        section .data
msg:    db      "Hello world: %c %s of length %d %d %X",10,0
char1:  db      'a'             ; character a
str1:   db      "string",0      ; ASCIIZ string
len:    equ     $-str1          ; len = length of str1
inta1:  dd      1234567         ; integer 1234567
hex1:   dd      0x6789ABCD      ; hex constant
        section .text
        global main
main:   push dword [hex1]       ; %X - hex constant
        push dword [inta1]      ; %d - integer data
        push dword len          ; %d - constant (equate)
        push dword str1         ; %s - pointer to "string"
        push dword [char1]      ; %c - the character 'a'
        push dword msg          ; pointer to format string
        call printf             ; call C library function
        add esp, 24             ; pop stack 6*4 = 24 bytes
        mov eax, 0              ; exit code
        ret
```


Assembly and Linking

```
~/nasm$ nasm -f elf printf2.asm
~/nasm$ gcc printf2.o
~/nasm$ a.out
Hello world: a string of length 7 1234567 6789ABCD
~/nasm$
```


Function Example — 1

```
    extern disp
    section .data
str1: db 'abcdefghijklmnopqrstuvwzyz',10,10

    section .text
    global _start
_start:
    push str1-1    ; points to char before string
    call disp

exit: mov eax,1
    mov ebx,0
    int 0x80
```

Function Example — 2

```
    section .text
    global disp
disp:                   ; esp = parameter + 4 (call pushes eip)
    push ebp            ; esp = parameter + 8
    push edi            ; esp = parameter + 12
    push esi            ; esp = parameter + 16
    mov ebp,esp         ; new data frame
    mov edi,[ebp+16]      ; edi <-- pointer to parameter
L1: inc edi               ; edi points to character in string
    mov eax,4
    mov ebx,1
    mov ecx,edi
    mov edx,1
    int 0x80
    cmp byte [edi],10
    jne L1
```

```
clean_up:
        mov esp,ebp
        pop esi
        pop edi
        pop ebp
        ret
```


Function Example — 3

```
~/nasm$ nasm -f elf ex3a.asm
~/nasm$ nasm -f elf ex3b.asm
~/nasm$ gcc -nostartfiles ex3a.o ex3b.o
~/nasm$ mv a.out ex3
~/nasm$ ex3
abcdefghijklmnopqrstuvwzyz
~/nasm$
```


Function Example — 4

```
00000000  68FF9F0408       push dword 0x8049fff
00000005  E816000000       call dword 0x20
0000000A  B801000000       mov eax,0x1
0000000F  BB00000000       mov ebx,0x0
00000014  CD80             int 0x80
0000001F  90               nop
00000020  55               push ebp
00000021  57               push edi
00000022  56               push esi
00000023  89E5             mov ebp,esp
00000025  8B7D10           mov edi,[ebp+0x10]
00000028  47               inc edi
00000029  B804000000       mov eax,0x4
0000002E  BB01000000       mov ebx,0x1
00000033  89F9             mov ecx,edi
00000035  BA01000000       mov edx,0x1
0000003A  CD80             int 0x80
0000003C  803F0A           cmp byte [edi],0xa
0000003F  75E7             jnz 0x28
00000041  89EC             mov esp,ebp
00000043  5E               pop esi
00000044  5F               pop edi
00000045  5D               pop ebp
00000046  C3               ret
```


Combining Assembly with C — 1

factorial.c

```
#include <math.h>
#include <stdio.h>
main()
{
     int times;
     int i , j = 12;
     for (times = 0 ; times < 10000000 ; ++times){
        i = factorial(j);
     }
     printf("%d\n",i);
}
int factorial(n)
     int n;
{
     if (n == 0)
        return 1;
     else
        return  n * factorial(n-1);
}
```

main
- sets j = 12
- calls factorial 10,000,000 times

factorial calculates n! by recursion

Combining Assembly with C — 2

```
~/nasm$ gcc factorial.c
```
produces executable `a.out`
```
~/nasm$ time a.out
479001600

real    0m9.281s
user    0m8.339s
sys     0m0.008s
```

Program `a.out` runs in 8.339 seconds on 300 MHz Pentium II

Combining Assembly with C — 3

Compile program as separate files

factorial-a.c
```
main()
{
  int times;
  int i,j=12;
  for (times = 0 ; times < 10000000 ; times++){
    i = factorial(j);
  }
  printf("%d\n",i);
}
```

factorial-b.c
```
#include <math.h>
#include <stdio.h>
int factorial(n)
   int n;
{
   if (n == 0)
       return 1;
   else
     return  n * factorial(n-1);
}
```


Combining Assembly with C — 4

```
~/nasm$ gcc -c factorial-a.c
```
produces linkable object file `factorial-a.o`
```
~/nasm$ gcc -c factorial-b.c
```
produces linkable object file `factorial-b.o`

```
~/nasm$ gcc factorial-a.o factorial-b.o
```
produces executable `a.out`

Identical to previous program version

Combining Assembly with C — 5

AT&T gas assembly from `gcc -S factorial-a.c`

```
        .file    "factorial2a.c"
        .section .rodata
.LC0:
        .string  "%d\n"
        .text
.globl main
        .type    main, @function
main:
        pushl    %ebp
        movl     %esp, %ebp
        subl     $24, %esp
        andl     $-16, %esp
        movl     $0, %eax
        addl     $15, %eax
        addl     $15, %eax
        shrl     $4, %eax
        sall     $4, %eax
        subl     %eax, %esp
        movl     $12, -4(%ebp)
        movl     $0, -12(%ebp)
        jmp      .L2
.L3:
        subl     $12, %esp
        pushl    -4(%ebp)
        call     factorial
        addl     $16, %esp
        movl     %eax, -8(%ebp)
        leal     -12(%ebp), %eax
        incl     (%eax)
.L2:
        cmpl     $9999999, -12(%ebp)
        jle      .L3
        subl     $8, %esp
        pushl    -8(%ebp)
        pushl    $.LC0
        call     printf
        addl     $16, %esp
        leave
        ret
```

Output from `intel2gas`

```
 ;FILE "factorial2a.c"
        SECTION    .rodata
.LC0:
        db       '%d',10,''
        SECTION .text
GLOBAL main
        GLOBAL main:function
main:
        push     ebp
        mov      ebp,esp
        sub      esp,24
        and      esp,-16
        mov      eax,0
        add      eax,15
        add      eax,15
        shr      eax,4
        sal      eax,4
        sub      esp,eax
        mov      dword [ebp-4],12
        mov      dword [ebp-12],0
        jmp L2
```

```
L3:
        sub      esp,12
        push     dword [ebp-4]
        call     factorial
        add      esp,16
        mov      [ebp-8],eax
        lea      eax, [ebp-12]
        inc      dword [eax]
L2:
        cmp      dword [ebp-12],9999999
        jle L3
        sub      esp,8
        push     dword [ebp-8]
        push     dword .LC0
        call     printf
        add      esp,16
        leave
        ret
```

intel2gas converts
- **gas** to **nasm** (Intel)
- **nasm** to **gas**

Combining Assembly with C — 6

Assembly version of **factorial** function written for **nasm**

- Uses "register variables" with no memory accesses
- Exploits advantages of Intel **imul** and **loop** instructions

```
section .text2
global factorial
factorial:
    push ebp              ; set up data frame
    mov ebp,esp
    mov ecx,[ebp+8]       ; ecx ← passed parameter
    mov eax,1             ; init eax
L1: imul ecx              ; eax ← eax * ecx
    loop L1               ; ecx ← ecx – 1
                          ; jmp L1 if ecx > 0
    mov esp,ebp           ; restore esp
    pop ebp               ; restore ebp
    ret                   ; return to calling function
```


Combining Assembly with C — 7

~/nasm$ `nasm -f elf -o factorial-c.o factorial-c.asm`

produces linkable object file **factorial-c.o**

~/nasm$ `gcc factorial-a.o factorial-c.o`

produces executable file **a.out**

```
~/nasm$ time a.out
479001600
real    0m4.964s
user    0m4.287s
sys     0m0.009s
```

C only version of **a.out** runs in 8.339 seconds on 300 MHz Pentium II

C + assembly version runs in 4.287 seconds on 300 MHz Pentium II

Speed-up of 8.339 / 4.287 ~ 2

Assembly Language Debugger

Assembly Language Debugger

- Similar to **debug.exe** for DOS
  - Help, disassemble, run, step, display memory & registers
- Source package **ald-0.1.7** available for Linux

```
~/nasm$ ald ex3
Assembly Language Debugger 0.1.7
Copyright (C) 2000-2004 Patrick Alken
ex3: ELF Intel 80386 (32 bit), LSB - little endian,
    Executable, Version 1 (Current)
Loading debugging symbols...(24 symbols loaded)
ald> disassemble
08048180:<_start>    68FF9F0408   push 0x8049fff
08048185             E816000000   call near +0x16 (0x80481a0:disp)
0804818A:<exit>      B801000000   mov eax, 0x1
0804818F             BB00000000   mov ebx, 0x0
08048194             CD80         int 0x80
```