زبان ماشین و اسمبلی
(۱۳-۱۱-۰۰۵)

اولین برنامه به زبان اسمبلی x86

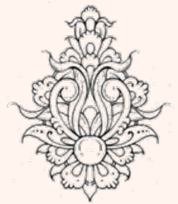
CPUID

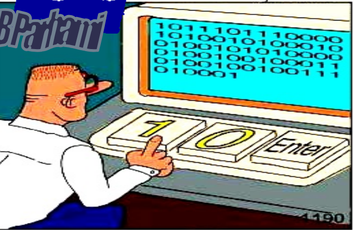
دانشگاه شهید بهشتی
دانشکده‌ی مهندسی برق و کامپیوتر
زمستان ۱۳۹۳
احمد محمودی ازناوه

http://faculties.sbu.ac.ir/~a_mahmoudi/

فهرست مطالب

- اولین برنامه به زبان اسمبلی x86
  - آشنایی با بخش‌های مختلف زبان اسمبلی
  - نماد دستورالعمل، راهنما و داده‌ها
  - آشنایی با قالب دستورها
  - دستورهای جابجایی

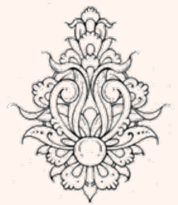




زبان اسمبلی

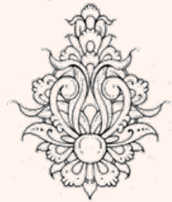
- زبان اسمبلی از سه بخش تشکیل می‌شود:
  - نماد دستورالعمل‌ها (*Opcode mnemonics*)
  - نمایش داده‌ها (*Data sections*)
  - راهنماها (*Directives*)

– اسمبلر گنو (GAS) برای نمایش دستورالعمل‌ها به جای شیوه‌ی نمایش Intel از قالب AT&T تبعیت می‌کند.



تفاوت‌های دو شیوه

- «داده‌های بی‌واسطه» در شیوه‌ی AT&T با پیشوند «$» همراه است در حالی که در فرمت Intel بدون پیشوند ظاهر می‌شود.
- همچنین در این فرمت برخلاف فرمت Intel برای نوشتن نام ثبات‌ها استفاده از پیشوند «%» لازم است.
- در شیوه‌ی AT&T عملگر مقصد در انتها قرار می‌گیرد در در حالی که Intel مقصد در ابتدا قرار می‌گیرد.
- در روش AT&T برای مشخص کردن طول عملوند از نمادهای متفاوت به عنوان دستورالعمل استفاده می‌شود، در حالی که در شیوه‌ی Intel طول عملوند، خود به عنوان عملوند دیگری ظاهر می‌شود.



movl $4, %eax

At&T

mov eax, 4

Intel

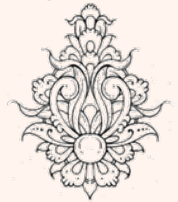
.intel_syntax directive

با این راهنما می‌توان با فرمت اینتل اسمبلی نوشت و با gas به‌زبان ماشین تبدیل کرد.

نماد دستورالعمل‌ها

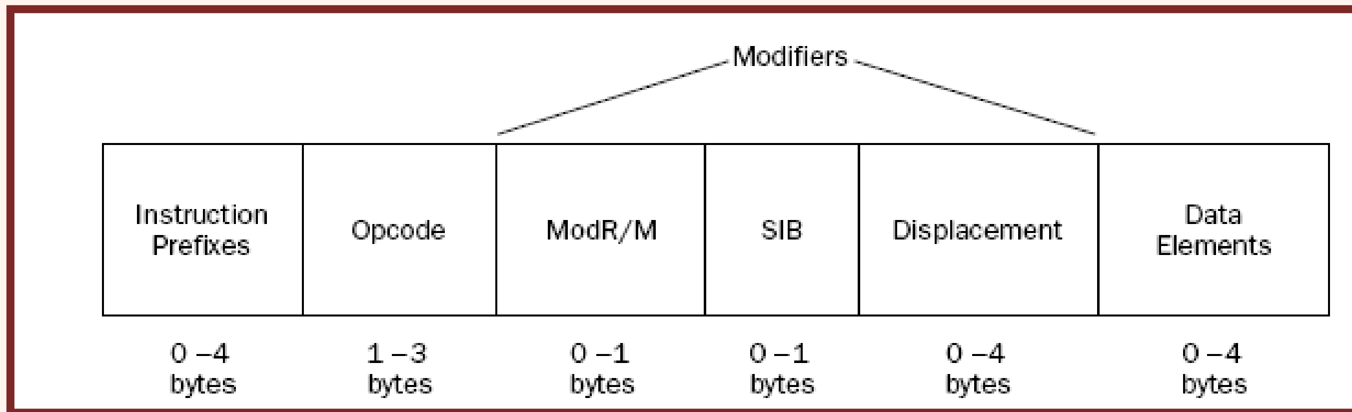
55	push %ebp
89 E5	mov %esp, %ebp
83 EC 08	sub $0x8, %esp
C7 45 FC 01 00 00 00	movl $0x1, -4(%ebp)
83 EC 0C	sub $0xc, %esp
6A 00	push $0x0
E8 D1 FE FF FF	call 8048348

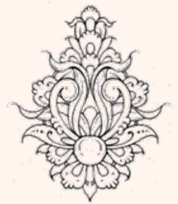
- دستورالعمل‌ها هسته‌ی مرکزی زبان اسمبلی را تشکیل می‌دهند.
- برای سادگی برنامه‌نویسی، هر دستور زبان ماشین معادل یک «نماد» در نظر گرفته می‌شود.
- اسمبلرهای مختلف، از نمادهای متفاوت استفاده می‌کنند. از این رو بین اسمبلرهای مختلف تفاوت‌هایی وجود دارد (نه تنها بین پردازنده‌های مختلف، حتی در مورد پردازنده‌های یک خانواده)



قالب دستورها در x86

- برخلاف MIPS، در خانواده‌ی x86 دستورها قالب پیچیده‌ای دارند. این قالب از چهار قسمت تشکیل شده است:
  - پیشوند دستورالعمل
  - بخش کد دستور (opcode)
  - بخش اصلاح‌کننده‌ها
  - بخش داده‌ها





برای آشنایی با جزییات این قالب به فصل اول کتاب مرجع مراجعه فرمایید.

Directive

pseudo ops

راهنما(شبه‌دستور)

- کلماتی هستند، که توسط اسمبلر و یا لینکر مورد استفاده قرار می‌گیرند.
  - این کلمات جزء دستورات پردازنده نیستند.
    - بخش‌های مختلف برنامه را مشخص می‌کنند.
    - به عنوان مثال برای مشخص کردن نوع داده‌ها از راهنماها استفاده می‌شود.
  - یکی از این راهنماهای پراستفاده «.section» است که بخش‌های یک برنامه را مشخص می‌کند.

نمایش داده‌ها

- برای نوشتن یک برنامه افزون بر دستورالعمل‌ها، نیاز به استفاده از متغیرها و همچنین داده‌های ثابت وجود دارد.
- هنگامی که در یک زبان سطح بالا متغیری تعریف می‌کنیم، در عمل کامپایلر متناسب با نوع تعریف شده، بخشی از حافظه را رزرو می‌کند.
- مانند زبان‌های سطح بالا، زبان اسمبلی شما را قادر می‌سازد متغیرهایی تعریف کنید، که به بخشی از حافظه اشاره می‌کنند.

```
long testvalue = 150;
char message[22] = “This is a test message”;
float pi = 3.14159;
```

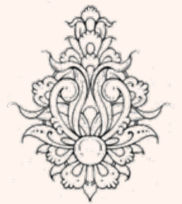
نمایش داده‌ها (ادامه...)

- تعریف یک متغیر از دو بخش تشکیل شده است:
  - برچسب (Label)

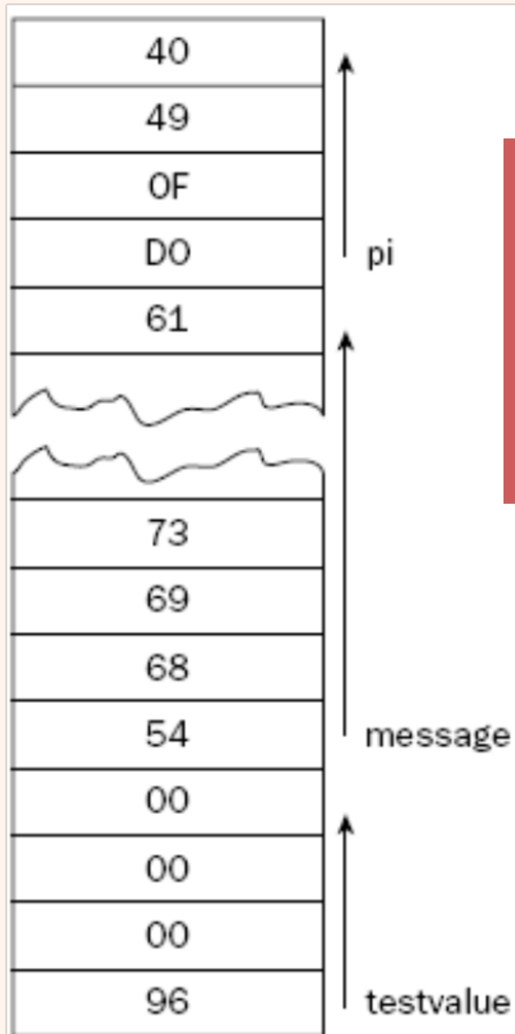
    برچسب به یک خانه‌ی حافظه اشاره می‌کند.

  - نوع داده (Data type)

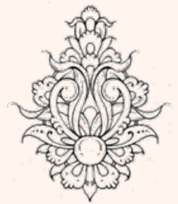
    نوع داده، میزان فضای مورد نیاز را نشان می‌دهد.



نمایش داده‌ها (ادامه...)



testvalue:
.long 150
message:
.ascii “This is a test message”
pi:
.float 3.14159



شبه‌دستورهای رزرو حافظه

Directive	Data Type
.ascii	Text string
.asciz	Null-terminated text string
.byte	Byte value
.double	Double-precision floating-point number
.float	Single-precision floating-point number
.int	32-bit integer number
.long	32-bit integer number (same as .int)
.octa	16-byte integer number
.quad	8-byte integer number
.short	16-bit integer number
.single	Single-precision floating-point number (same as .float)

- می‌توان با استفاده از یک شبه‌دستور چندین خانه‌ی حافظه را رزرو کرد.

```
label:
.long 100,150,200,250,300
```

ساختار برنامه

در اسمبلر GNU از راهنمای section. برای جدا کردن بخش‌های مختلف استفاده می‌شود.

.section.data

داده‌هایی که در برنامه مورد استفاده قرار می‌گیرد، در این بخش قرار می‌گیرد

.section.bss

فعلا با این قسمت کاری نداریم

.section.text

این بخش در تمامی برنامه‌های اسمبلی الزامی است. دستورالعمل ها در این بخش نوشته می‌شوند.

دستور جابجایی داده

- قالب کلی جابجایی داده به صورت زیر است:

movx source, destination

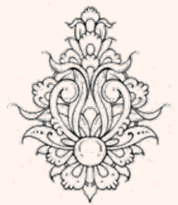
l for a 32-bit long word value
w for a 16-bit word value
b for an 8-bit byte value

- مثال:

movb %al, %bl

movw %ax, %bx

movl %eax, %ebx



دستور جابجایی داده (ادامه...)

- جابجایی داده
  - داده‌ی بی‌واسطه به ثبات‌های همه‌منظوره یا حافظه
  - از ثبات (همه‌منظوره) به ثباتی دیگر
  - از حافظه به ثبات
  - از ثبات به حافظه
  - امکان جابجایی یک خانه‌ی حافظه با خانه‌ی حافظه دیگر وجود ندارد.

دستور جابجایی داده (ادامه...)

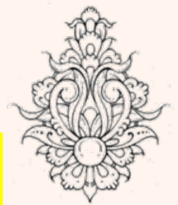
- جابجایی بین ثبات‌ها:

```
movl %eax, %ecx
movw %ax, %cx
```

- انتقال عدد ثابت به حافظه و یا ثبات:

```
movl $0, %eax
movl $0x80, %ebx
movl $100, height
```

در صورتی به طول عملوند توجه نشود، با خطا مواجه خواهیم شد



movb %al, %bx

```
.s: Assembler messages:
.s:19: Error: suffix or operands invalid for `mov'
@ubuntu:~$
```

دستور جابجایی داده (ادامه...)

- جابجایی بین ثبات‌ها و محتوای حافظه:

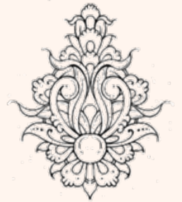
```
movw %cx, num
movl height, %eax
```

- انتقال آدرس حافظه به ثبات:

```
movl $height, %eax
```

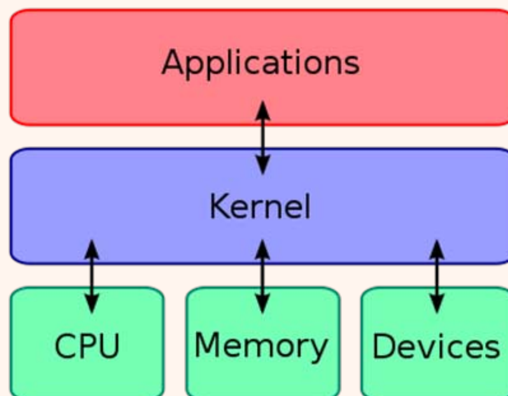
اولین برنامه به زبان اسمبلی

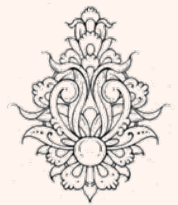
- همانند سایر زبان‌های برنامه‌نویسی، اولین برنامه‌ای که در اینجا خواهیم نوشت، hello world خواهد بود.
- چنین برنامه‌ای به نظر شما چگونه نوشته خواهد شد؟
- برای چاپ این عبارت ناچار به ارتباط با سخت‌افزار خواهیم بود!
- این کار از طریق «وقفه‌های سیستم‌عامل» انجام می‌شود.



فراخوانی سیستمی

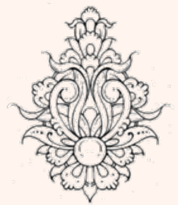
- بیشتر سیستم‌عامل‌ها توابعی دارند (core functions) که برنامه‌های کاربردی هم به آن دسترسی دارند.
- هسته‌ی سیستم‌عامل، وظیفه‌ی کنترل سخت‌افزار و نرم‌افزار را بر عهده دارد. کار کردن با سخت‌افزار و اجرای برنامه‌ها از وظایف سیستم‌عامل است.
- به طور کلی سیستم‌عامل وظیفه‌ی مدیریت منابع را به عهده دارد.





وقفه

- به حادثه‌ای با منشأ داخلی یا خارجی که روند عادی پردازنده را برای ارائه‌ی سرویس خاص متوقف می‌کند، «**وقفه**» می‌گویند.
- در مقابل وقفه مکانیزم Polling قرار دارد.
- انواع وقفه
  - خارجی
  - داخلی
    - نرم‌افزاری (فراخوانی سیستمی)
    - استثنائات



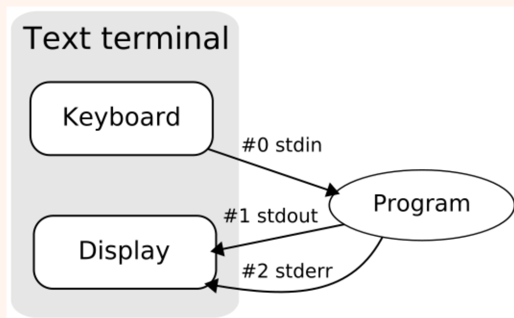
فراخوانی سیستمی (ادامه...)

- برای استفاده از توابع سیستمی linux باید از وقفه‌ی 0x80 استفاده کرد.
- در واقع برای سرویس‌های مختلف توابع مختلفی وجود دارد که از طریق شماره مشخص می‌شوند.
- برخلاف توابع C که پارامترها را از طریق پشته دریافت می‌کنند، در توابع سیستمی، پارامترها از طریق ثبات‌ها منتقل می‌شوند.
- ثبات eax حاوی شماره‌ی تابعی خواهد بود، که می‌خواهیم اجرا شود.

```
movl  $1, %eax
int   0x80
```

فراخوانی سیستمی (ادامه...)

- در این جا تنها در مورد دو تابع سیستمی به صورت مختصر صحبت می‌شود:
  - تابع شماره ۱: تابع خروج، اتمام اجرای فرآیند اجرا
    - مقدار بازگشتی در ebx قرار خواهد گرفت.
  - تابع شماره ۴: نوشتن داده در خروجی
    - با استفاده از ebx خروجی مشخص می‌شود. (عدد یک معادل خروجی استاندارد (STDOUT)است)
    - ecx حاوی آدرس رشته‌ایی است که بناست چاپ شود.
    - طول رشته در edx قرار می‌گیرد.



اولین برنامه

```
# example 1, hello, world!
.section .data
output:
    .ascii "hello, world!\n"
.section .text
.globl _start
_start:
    movl $4, %eax
    movl $1, %ebx
    movl $output, %ecx
    movl $14, %edx
    int $0x80

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

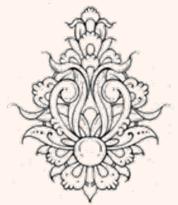
بخش داده

بعد از علامت #
می‌توان توضیحات را
نوشت

با این کار نماد استفاده شده برای
لینکر و سایر برنامه‌های خارجی قابل
مشاهده باشد

باید به نحوی به لینکر اطلاع داد
که شروع برنامه از کجاست. این
کار با برچسب _start انجام
می‌شود

بخش code



می‌توان به جای _start از کلمه‌ی دیگری نیز استفاده کرد، در این
صورت نام انتخاب شده با پارامتر e- به لینکر معرفی خواهد شد

اولین برنامه (ادامه...)

```
# example 1, hello, world!
.section .data
output:
    .ascii "hello, world!\n"
.section .text
.globl _start
_start:
    movl $4, %eax
    movl $1, %ebx
    movl $output, %ecx
    movl $14, %edx
    int $0x80

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

با استفاده از این وقفه رشته‌ای که آدرس آن مشخص شده است چاپ می‌شود

در اینجا علامت $ بیانگر آدرسی است که برچسب نشان می‌دهد.

exit(0)

اولین برنامه به زبان اسمبلی (ادامه...)

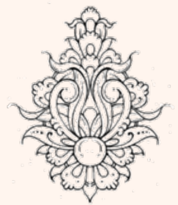
برای تبدیل فایل اسمبلی به زبان ماشین و لینک کردن آن

```
$ as -o example1.o example1.s
$ ld -o example1  example1.o
```

بدین ترتیب می‌توان این برنامه را اجرا کرد

```
$ ./example1
```

```
ahmad@ubuntu:~/Courses/Assembly/Sample$ ./example1
hello, world!
```



جابجایی داده بین حافظه و ثبات

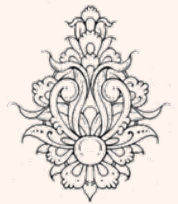
- در اولین قدم باید مشخص کرد، نحوه‌ی دستیابی به حافظه چگونه است.
  – دستیابی مستقیم:

movl value, %eax

movl %ecx, value

```
# movtest1.s
.section .data
value:
.int 1
.section .text
.globl _start
_start:
movl value, %ecx
movl $1, %eax
movl $0, %ebx
int $0x80
```

```
# movtest2.s
.section .data
value:
.int 1
.section .text
.globl _start
_start:
movl $100, %eax
movl %eax, value
movl $1, %eax
movl $0, %ebx
int $0x80
```



آدرس‌دهی غیرمستقیم ثبات

- ثبات‌ها علاوه بر نگهداری داده، برای نگهداری آدرس خانه‌های حافظه به کار می‌آیند.
- هنگامی که یک ثبات، آدرس یک خانه‌ی حافظه را در خود دارد، در عمل نقش یک «اشاره‌گر» را ایفا می‌کند.
- در صورتی که برچسب label مرجع داده‌ای در حافظه باشد، با دستور movl $values, %edi می‌توان آدرسی که توسط این برچسب مشخص شده است را به ثبات edi منتقل نمود.

این علامت اسمبلر را راهنمایی می‌کند:
منظور آدرسی است که برچسب نشان می‌دهد نه محتوای این خانه‌ی حافظه

آدرس‌دهی غیرمستقیم ثبات (ادامه...)

```
movl $values, %edi
```

- تا بدین‌جای کار یک آدرس به ثبات انتقال یافت:

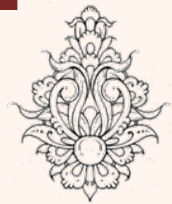
```
movl %ebx, (%edi)
```

بدون این پرانتزها محتوای ebx در edi نوشته می‌شد، اما با وجود این پرانتزها محتوای ebx در خانه‌ای از حافظه که edi به آن اشاره می‌کند، نوشته خواهد شد.

- امکان افزودن یک مقدار به این آدرس مشخص شده وجود دارد، برای این کار مقدار مورد نظر پیش از پرانتزها نوشته خواهد شد.

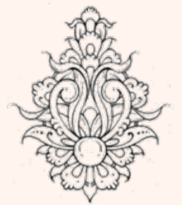
```
movl %edx, 4(%edi)
```

```
movl %edx, -4(%edi)
```



مثال

```
 #movtest4.s – An example of indirect addressing
.section .data
values:
.   int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
.section .text
.globl _start
_start:
   movl values, %eax
   movl $values, %edi
   movl $100, 4(%edi)
   movl  4(%edi),%ebx
    movl $1, %eax
   int $0x80
```



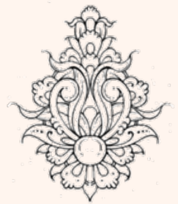
CPUID

برنامه‌ای بنویسید که سازنده‌ی پردازنده را در خروجی چاپ کند

- اجرای چنین دستورالعملی، به سادگی در زبان‌های سطح بالا امکان‌پذیر نمی‌باشد.
- این دستورالعمل بسته به خواسته‌ی کاربر، اطلاعاتی در مورد سازنده و مدل پردازنده ارائه می‌کند.
- نوع اطلاعات مورد نظر با استفاده از ثبات EAX اعلام می‌شود:

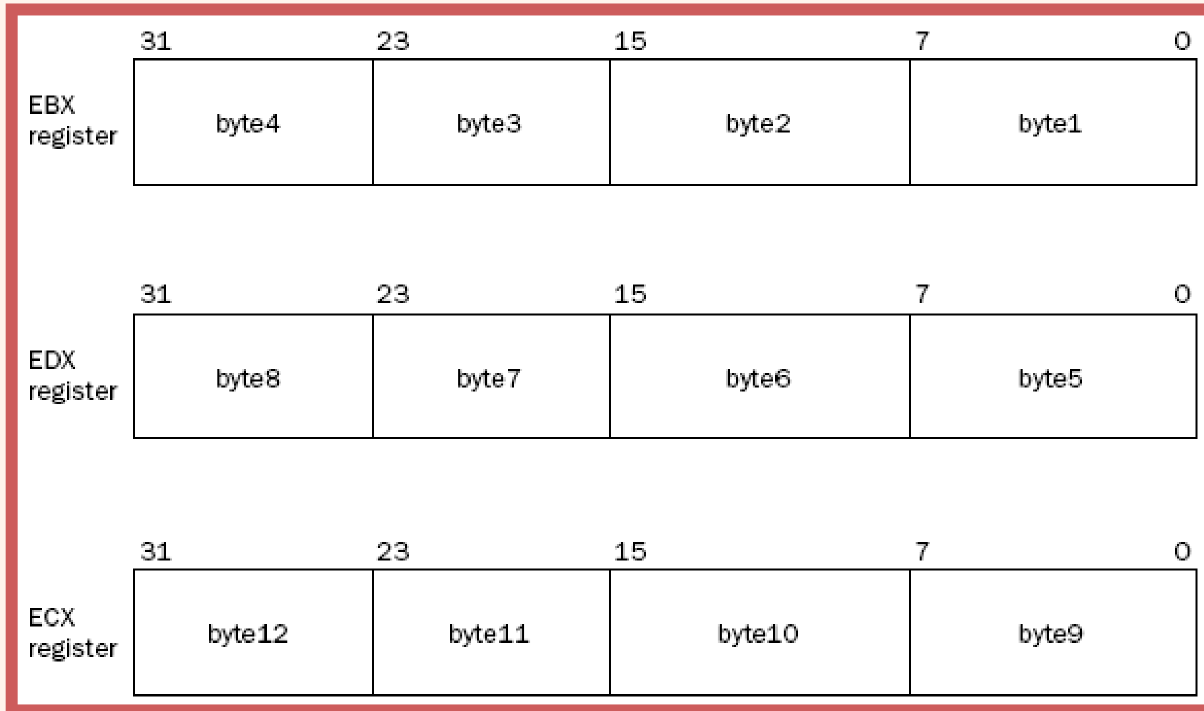
EAX Value	CPUID Output
0	Vendor ID string, and the maximum CPUID option value supported
1	Processor type, family, model, and stepping information
2	Processor cache configuration
3	Processor serial number
4	Cache configuration (number of threads, number of cores, and physical properties)
5	Monitor information
80000000h	Extended vendor ID string and supported levels
80000001h	Extended processor type, family, model, and stepping information
80000002h - 80000004h	Extended processor name string

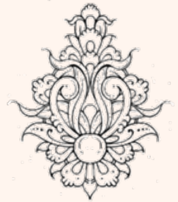
در عمل ثبات EAX مانند آرگومان ورودی عمل می کند



CPUID (ادامه...)

- هنگامی که در EAX عدد 0 قرار گیرد و این دستور اجرا شود، یک رشته که شناسه‌ی سازنده را نشان می‌دهد به ترتیب در ثبات‌های EBX، EDX و ECX قرار خواهد گرفت.





CPUID

```
#cpuid.s Sample program to extract the processor Vendor ID
.section .data
output:
.ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"

.section .text
.globl _start
_start:
  movl $0, %eax
  cpuid
  movl $output, %edi
  movl %ebx, 28(%edi)
  movl %edx, 32(%edi)
  movl %ecx, 36(%edi)
  movl $4, %eax
  ...
  movl $42, %edx
  int $0x80
  movl $1, %eax
  movl $0, %ebx
  int $0x80
```

بخش داده

بخش code

default label (identifier)

باید به نحوی به لینکر اطلاع داد که شروع برنامه از کجاست، این کار با برچسب _start انجام می‌شود

در صورت نبود این برچسب خطای زیر رخ می‌دهد:

```
ahmad@ubuntu:~$ ld -o cpuid cpuid.o
ld: warning: cannot find entry symbol _start; defaulting to 0000000008048074
```

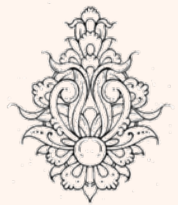
CPUID (ادامه...)

```
#cpuid.s Sample program to extract the processor Vendor ID
.section .data
output:
.ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"

.section .text
.globl _start
_start:
   movl $0, %eax
   cpuid
   movl $output, %edi
   movl %ebx, 28(%edi)
   movl %edx, 32(%edi)
   movl %ecx, 36(%edi)
   movl $4, %eax
   movl $1, %ebx
   movl $output, %ecx
   movl $42, %edx
   int $0x80
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

بخش code

cpuid

با این دستور می‌توانید سازنده و مدل پردازنده را دریافت کرده و نمایش دهید



CPUID (ادامه...)

```
#cpuid.s Sample program to extract the processor V
.section .data
output:
 .ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"

.section .text
.globl _start
_start:
   movl $0, %eax
   cpuid
   movl $output, %edi
   movl %ebx, 28(%edi)
   movl %edx, 32(%edi)
   movl %ecx, 36(%edi)
   movl $4, %eax
   movl $1, %ebx
   movl $output, %ecx
   movl $42, %edx
   int $0x80
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

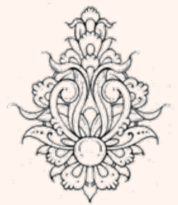
بخش داده

این قسمت برای در نظر گرفتن جای خالی در حافظه استفاده می‌شود

این برچسب در عمل یک آدرس از حافظه را نشان می‌دهد

این شبه‌دستور نشان می‌دهد که داده‌ی که در حافظه ذخیره می‌شود، رشته‌ی متنی است که با کدهای اسکی نمایش داده می‌شوند

با استفاده از این شبه دستور اعلام می‌شود که این برچسب global است، یعنی از برنامه‌های دیگر نیز قابل دسترسی است



CPUID (ادامه...)

```
#cpuid.s Sample program to extract the processor Vendor ID
.section .data
output:
 .ascii "The processor Vendor ID is 'xxxxxxx
.section .text
.globl _start
_start:
   movl $0, %eax
   cpuid
   movl $output, %edi
   movl %ebx, 28(%edi)
   movl %edx, 32(%edi)
   movl %ecx, 36(%edi)
   movl $4, %eax
   movl $1, %ebx
   movl $output, %ecx
   movl $42, %edx
   int $0x80
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

بدین ترتیب، اولین قدم، قرار دادن پارامتر مورد نظر در ثبات EAX می‌باشد.

در این بخش عملا یک اشاره‌گر درست کرده و با کمک آن محتوای ثبات‌ها را به حافظه منتقل می‌کند

در قدم بعدی، رشته‌ای که در output قرار دارد، به واسط خط فرمان فرستاده شود. برای این کار از توابع سیتمی لینوکس استفاده می‌کنیم. برای این کار از دستور int $0x80 استفاده می‌شود. با این کار یک وقفه‌ی نرم‌افزاری تولید می‌شود.

حال نوبت به خاتمه دادن به اجرای این برنامه می‌رسد برای این کار نیز از وقفه نرم‌افزاری استفاده کنیم، بازگرداندن مقدار 0 بدین معناست که خطایی رخ نداده است.

CPUID (ادامه...)

```
$ as -o cpuid.o cpuid.s
$ ld -o cpuid cpuid.o
```

برای تبدیل فایل اسمبلی به زبان ماشین و لینک کردن آن

بدین ترتیب می‌توان این برنامه را اجرا کرد

```
$ ./cpuid
```

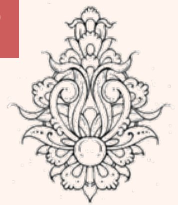
```
The processor Vendor ID is 'GenuineIntel'
```

دقت داشته باشید، می‌توان از gcc نیز استفاده نمود

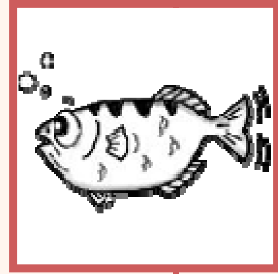
```
$ gcc -o cpuid cpuid.s
$ ./cpuid
```

البته با اعمال تغییرات زیر

```
.section .text
.globl main
main:
```



استفاده از gdb



```
ahmad@ubuntu:~$ ls -l cpuid
-rwxr-xr-x 1 ahmad ahmad 663 2011-10-28 02:54 cpuid
ahmad@ubuntu:~$
```

- برای این که بتوانیم از gdb استفاده کنیم، می‌باید برنامه را دوباره و با استفاده از پارامتر –gstabs اسمبل کنیم.

```
File  Edit  View  Terminal  Help
ahmad@ubuntu:~$ as -gstabs -o cpuid.o cpuid.s
ahmad@ubuntu:~$ ld -o cpuid cpuid.o
ahmad@ubuntu:~$
```

```
ahmad@ubuntu:~$ ls -l cpuid
-rwxr-xr-x 1 ahmad ahmad 991 2011-10-28 02:40 cpuid
ahmad@ubuntu:~$
```

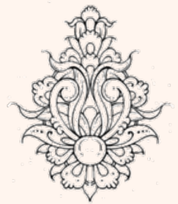
با این پارامتر اطلاعاتی به برنامه‌ی اسمبل شده افزوده می‌شود، تا gdb بتواند برنامه‌را خط به خط دنبال کند.

استفاده از gdb (ادامه...)

```
ahmad@ubuntu:~$ gdb cpuid
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/ahmad/cpuid...done.
(gdb)
```

- بدین‌ترتیب دیباگر گنو شروع به کار می‌کند، برنامه با دستور run شروع به اجرا شدن می‌کند.

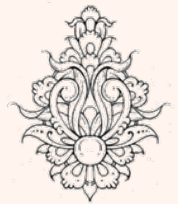
```
(gdb) run
Starting program: /home/ahmad/cpuid
The processor Vendor ID is 'GenuineIntel'

Program exited normally.
(gdb)
```



استفاده از gdb (ادامه...)

- برای این که بتوانیم، برنامه را خط به خط اجرا کنیم، باید از breakpoint استفاده کنیم.
  - با مشخص کردن برچسب
  - با مشخص کردن یک خط از برنامه
  - بدین‌ترتیب که اجرای برنامه ادامه یابد تا زمانی که یک متغیر به مقدار خاصی برسد.
  - بعد از این که یک تابع تعداد دفعات مشخصی تکرار شد.

```
break *label+offset
```



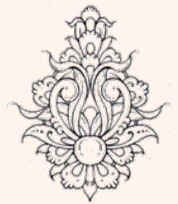
استفاده از gdb (ادامه...)

- با قرار دادن breakpoint را در خط اول، (برای اولین دستور) اجرای برنامه متوقف نمی‌شود.

```
break *_start
```

- برای رفع این مشکل می‌توان به صورت زیر عمل کرد:

```
break *_start+5
```



و یا این که یک nop در اولین سطر برنامه وارد کرده و breakpoint را برای آن سطر قرار داد.

```
break *_start+1
```

استفاده از gdb (ادامه...)

```
(gdb) next
11          movl $output, %edi
(gdb) info reg
eax            0xa      10
ecx            0x6c65746e       1818588270
edx            0x49656e69       1231384169
ebx            0x756e6547       1970169159
esp            0xbffff4f0       0xbffff4f0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804807c        0x804807c <_start+8>
eflags         0x212    [ AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

Data Command	Description
info registers	Display the values of all registers
print	Display the value of a specific register or variable from the program
x	Display the contents of a specific memory location

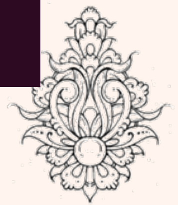
در کلاس حل تمرین با سایر دستورات gdb آشنا خواهید شد

مثال

```c
#include <stdio.h>
int main(){
    char *S="sBU Assembly class!";
    *S='S';
    printf("%s\n",S);
    return 0;
}
```

```
File  Edit  View  Terminal  Help
ahmad@ubuntu:~/Courses/Assembly$ gcc -Wall test.c -o test
ahmad@ubuntu:~/Courses/Assembly$
```

```
ahmad@ubuntu:~/Courses/Assembly$ ./test
Segmentation fault
```



تعریف انواع داده‌ای

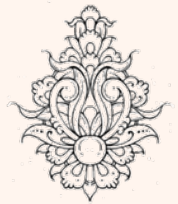
- برای تعریف انواع داده‌ها می‌توان از دو بخش **داده** و **پشته** استفاده نمود.
- یکی از معمول‌ترین روش‌ها استفاده از بخش **داده** است.
- برای تمام داده‌هایی که در این بخش مورد استفاده قرار می‌گیرند، جایی در حافظه در نظر گرفته می‌شود، که از تمامی بخش‌ها می‌توان به آن دسترسی داشت، طول عمر چنین داده‌ای، برابر با **طول عمر برنامه** است.
- داده‌هایی که در این بخش تعریف می‌شوند، در فایل اجرایی خواهند بود.
- همان‌طور که پیش از این دیدیم، برای تعریف داده به دو بخش نیاز داریم:
  - برچسب
  - شبه‌دستور

section.data

برچسب هیچ معنایی برای اسمبلر ندارد، تنها به‌عنوان یک نقطه‌ی مرجع برای دستیابی به حافظه مورد استفاده قرار می‌گیرد.

طول فضایی که باید ذخیره شود، توسط شبه‌دستورها مشخص می‌شود

می‌توان بخش دیگری با نام rodata. تعریف کرد، داده‌های این بخش فقط خواندنی هستند و قابل تغییر دادن نیستند



معادل اسمبلی مثال قبل

```
ahmad@ubuntu:~/Courses/Assembly$ gcc test.c -S
ahmad@ubuntu:~/Courses/Assembly$
```

```
ahmad@ubuntu:~/Courses/Assembly$ cat test.s
        .file   "test.c"
        .section        .rodata
.LC0:
        .string "sBU Assembly class!"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        ...
        movl    $.LC0, 28(%esp)
```

با دستکاری فایل اسمبلی تولید شده، می‌توانید خروجی این برنامه‌را ببینید

شبه‌دستورهای رزرو حافظه (ادامه...)

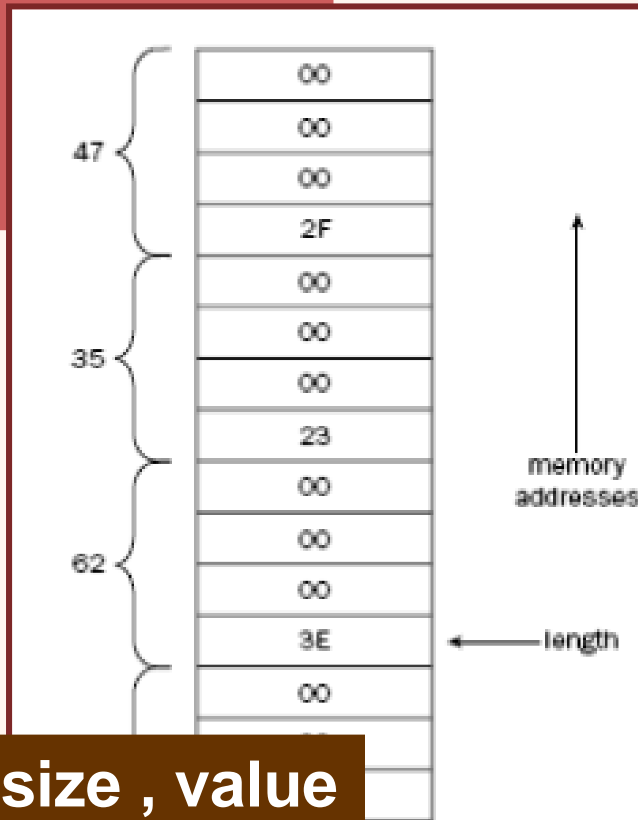
Defining static symbols

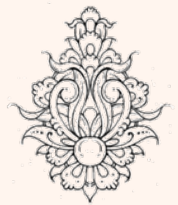
- می‌توان نمادهایی را به صورت ثابت تعریف نمود:

```
.equ factor, 3
.equ LINUX_SYS_CALL, 0x80
```

```
movl $LINUX_SYS_CALL, %eax
```

```
.section .data
msg:
.ascii "This is a test message"
factors:
.double 37.45, 45.33, 12.30
height:
.int 54
length:
.int 62, 35, 47
```





.fill repeat , size , value

- با استفاده از این راهنما می‌توان به‌تعداد repeat داده‌ای با اندازه‌ی size و مقدار اولیه‌ی value در بخش .data قرار داد.

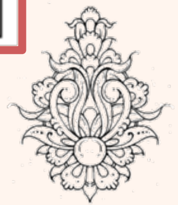
بخش bss

- با استفاده از bss میزان مشخصی از حافظه برای استفاده‌های بعدی رزرو می‌شود.
- اسمبلر گنو از دو شبه دستور برای این منظور استفاده می‌کند.

Directive	Description
.comm	Declares a common memory area for data that is not initialized
.lcomm	Declares a local common memory area for data that is not initialized

در خارج از فضای محلی کد قابل دسترسی نیستند

```
.comm symbol, length
```

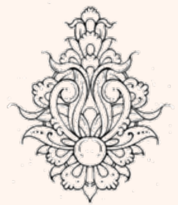


```
.section .bss
.lcomm buffer, 10000
```

بدین ترتیب ۱۰۰۰۰ بایت فضا برای استفاده‌های بعدی در نظر گرفته می‌شود

بخش bss (ادامه...)

- یکی از محاسن استفاده از این بخش این است که در object file فضایی اشغال نمی‌کند.
- تنها میزان فضای مورد نیاز در object file قرار داده می‌شود.
- در واقع فضای مورد نیاز در هنگام اجرای برنامه و توسط loader تخصیص داده می‌شود.
- آیا می‌توان متغیرهایی که مقداردهی اولیه دارند را در این بخش قرار داد؟



بخش bss (ادامه...)

```
#sizetest1.s – A sample program to view
      the executable size
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

```
#sizetest2.s - A sample program to view
      the executable size
.section .bss
.   lcomm buffer, 10000
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

```
# sizetest3.s - A sample program to
      view the executable size
.section .data
buffer:
   .fill 10000
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

```
File  Edit  View  Terminal  Help
ahmad@ubuntu:~$ as -o sizetest1.o sizetest1.s
ahmad@ubuntu:~$ ld -o sizetest1   sizetest1.o
ahmad@ubuntu:~$ ls -l sizetest1
-rwxr-xr-x 1 ahmad ahmad 460 2011-10-28 10:06 sizetest1
ahmad@ubuntu:~$ as -o sizetest2.o sizetest2.s
ahmad@ubuntu:~$ ld -o sizetest2   sizetest2.o
ahmad@ubuntu:~$ ls -l sizetest2
-rwxr-xr-x 1 ahmad ahmad 575 2011-10-28 10:07 sizetest2
ahmad@ubuntu:~$ as -o sizetest3.o sizetest3.s
ahmad@ubuntu:~$ ld -o sizetest3   sizetest3.o
ahmad@ubuntu:~$ ls -l sizetest3
-rwxr-xr-x 1 ahmad ahmad 10575 2011-10-28 10:09 sizetest3
ahmad@ubuntu:~$
```

پشته

existing
stack
data

00
00
00
2F
00
00
00
23
00
00
00
45
00
00
00
12

memory
addresses

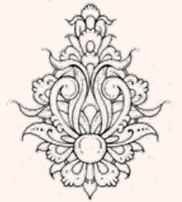
current ESP

new ESP

ECX

00
00
00
12

push1 % ecx



پشته (ادامه...)

pushx source

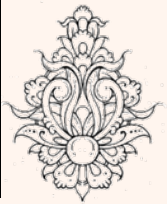
- 16-bit register values
- 32-bit register values
- 16-bit memory values
- 32-bit memory values
- 16-bit segment registers
- 16-bit immediate data values
- 32-bit immediate data values

popx destination

- 16-bit registers
- 16-bit segment registers
- 32-bit registers
- 16-bit memory locations
- 32-bit memory locations

سایر شکل‌های دستورهای مرتبط با پشته

Instruction	Description
PUSHA/POPA	Push or pop all of the 16-bit general-purpose registers
PUSHAL/POPAL	Push or pop all of the 32-bit general-purpose registers
PUSHF/POPF	Push or pop the lower 16 bits of the EFLAGS register
PUSHFL/POPFL	Push or pop the entire 32 bits of the EFLAGS register



توجه داشته باشید با استفاده از دستور mov هم می‌توان داده‌ای به پشته منتقل کرد و یا از روی پشته داده‌ای را خواند.

قالب دستورات

- در خانواده‌ی x86 دستورهای محاسباتی دو عملوند دارند:

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

عملیات حسابی صحیح- جمع

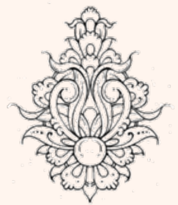
```
addx source, destination
```

immediate value
memory location
register

memory location
register

هر دو عملوند نمی‌توانند حافظه باشند

```
addb $10, %al
addw %bx, %cx
addl data, %eax
addl %eax, %eax
```

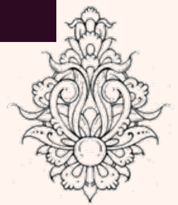


مثال

addtest1.s

```
.section .data
data:
    .int 40
.section .text
.globl _start
_start:
    nop
    movl $0, %eax
    movl $0, %ebx
    movl $0, %ecx
    movb $20, %al
    addb $10, %al
    movw $100, %cx
    addw %cx, %bx
    movl $100, %edx
    addl %edx, %edx
    addl data, %eax
    addl %eax, data
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

```
(gdb) print $eax
$1 = 70
(gdb) print $ebx
$2 = 100
(gdb) print $ecx
$3 = 100
(gdb) print $edx
$4 = 200
(gdb) x &data
0x80490b0 <data>:       0x0000006e
(gdb) x /d &data
0x80490b0 <data>:       110
```



عملیات حسابی صحیح- تفریق

```
subx source, destination
```

source:
immediate value
memory location
register

destination:
memory location
register

هر دو عملوند نمی‌توانند حافظه باشند