زبان ماشین و اسمبلی
(۱۳-۱۱-۰۰۵)

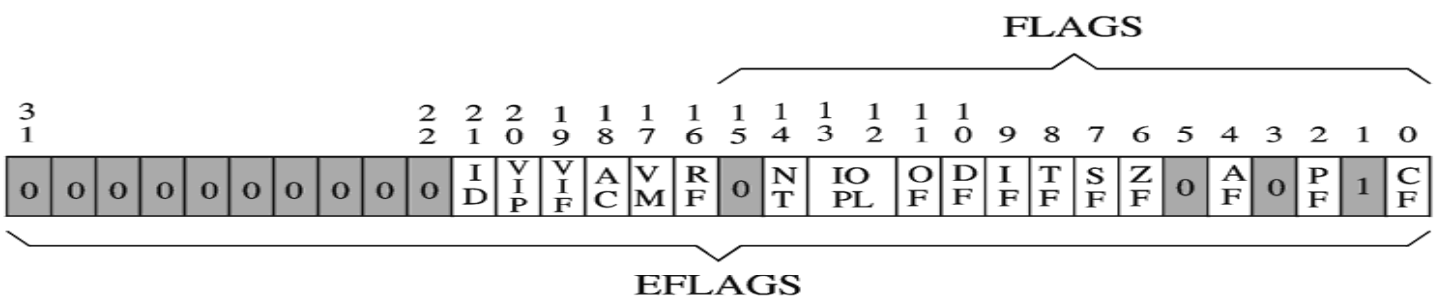
دستورهای شرطی
و محاسباتی

دانشگاه شهید بهشتی
دانشکده‌ی مهندسی برق و کامپیوتر
بهار ۱۳۹۴
احمد محمودی ازناوه

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

فهرست مطالب

نگاهی کلی به ثبات پرچم‌ها



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

یادآوری

پرچم‌های وضعیت

- EFLAGS
  - status flags
  - control flags
  - system flags

هر پرچم (flag) یک بیت است که به طور گسترده در دستورات پرش شرطی مورد استفاده قرار می‌گیرد

- Carry
  - unsigned arithmetic out of range
- Parity
  - sum of 1 bits is an even number
- Adjust Flag (Auxiliary Carry)
  - used for BCD numbers
- Zero
  - result is zero
- Sign
  - result is negative
- Overflow
  - signed arithmetic out of range

Status flags

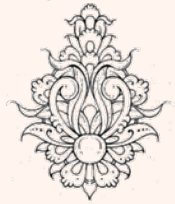
یادآوری

۱۵	۱۴	۱۳	۱۲	۱۱	۱۰	۹	۸	۷	۶	۵	۴	۳	۲	۱	۰
				OF				SF	ZF		AF		PF		CF

انشعابات شرطی

- برخلاف انشعاب‌های معمولی، انشعاب‌های شرطی همیشه صورت نمی‌گیرند. در این حالت پرش بستگی به مقدار پرچم‌های خاصی در ثبات EFLAGS دارد. پرچم‌های زیر در پرش‌های شرطی نقش دارند:

- Carry flag (CF) - bit 0 (lease significant bit)
- Overflow flag (OF) - bit 11
- Parity flag (PF) - bit 2
- Sign flag (SF) - bit 7
- Zero flag (ZF) - bit 6



پرش‌های شرطی

یک یا دو حرف است که شرط انتقال را نشان می‌دهد.

- ساختار پرش‌های شرطی به صورت زیر است:

- نتیجه‌ی عملیات محاسباتی(به عنوان مثال تفریق) قبل از دستور پرش ثبات‌های پرچم را تحت‌تاثیر قرار می‌دهند، دستور پرش شرطی بر اساس مقدار ثبات پرچم‌ها پرش را انجام می‌دهد.
- دستورات شرطی که از کلمات above و below استفاده می‌کنند، مربوط به اعداد بدون علامت هستند.
- دستورات شرطی که از کلمات greater و less استفاده می‌کنند، مربوط به اعداد علامت‌دار هستند.

مثال

```
if(operand1==operand2)
 goto label1
```

```
sub operand1, operand2
je label1
```

ZF=1

```
if(operand1>=operand2)   //unsigned
 goto label1
```

```
sub operand2, operand1
jae  label1
```

CF=0

CF در تفریق اعداد بدون علامت نقش رقم قرضی را دارد.

```
if(operand1>=operand2)   //signed
 goto label1
```

```
sub operand2, operand1
jge  label1
```

SF=OF

SF=0(OF=0)

SF=1(OF=1)

مثال

```
if(operand1>operand2)   //unsigned
 goto label1
```

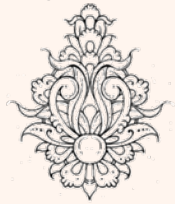
```
sub operand2, operand1
ja  label1
```

CF=0 and ZF=0

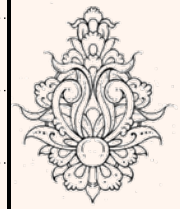
```
if(operand1>operand2)   //signed
 goto label1
```

```
sub operand2, operand1
jg  label1
```

SF=OF and ZF=0



Instruction	Description	EFLAGS
JA	Jump if above	CF=0 and ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX register is 0	
JECXZ	Jump if ECX register is 0	
JE	Jump if equal	ZF=1
JG	Jump if greater	ZF=0 and SF=OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 or SF<>OF
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=0 and ZF=0



Instruction	Description	EFLAGS
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater	ZF=1 or SF<>OF
JNGE	Jump if not greater or equal	SF<>OF
JNL	Jump if not less	SF=OF
JNLE	Jump if not less or equal	ZF=0 and SF=OF
JNO	Jump if not overflow	OF=0
JNP	Jump if not parity	PF=0
JNS	Jump if not sign	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if parity odd	PF=0
JS	Jump if sign	SF=1
JZ	Jump if zero	ZF=1

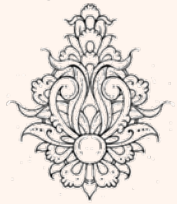
پرش‌های شرطی

- برای استفاده از دستورهای شرطی، باید به گونه‌ای پرچم‌ها را تغییر داد.
- برای این کار از دستور «مقایسه» استفاده می‌شود.

```
cmp operand1, operand2
```

(operand2-operand1)

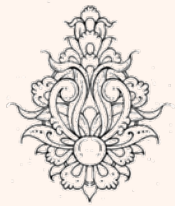
- با استفاده از این دستور نتیجه، بدون این که عملوندها را تحت تاثیر قرار دهند، محاسبه می‌شود.
- نتیجه‌ی اجرای این دستور تنها روی پرچم‌ها قابل مشاهده خواهد بود.



مثال

```
# cmptest.s - An example of using the CMP and JGE instructions
.section .text
.globl _start
_start:
   nop
   movl $15, %eax
   movl $10, %ebx
   cmp %eax, %ebx
   jge greater
   movl $1, %eax
   int $0x80
greater:
   movl $20, %ebx
   movl $1, %eax
   int $0x80
```

```
ahmad@ubuntu:~/MyData/courses/Asm$ echo $?
10
```

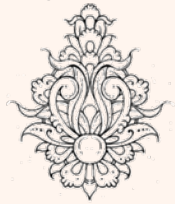


با استفاده از echo $? می‌توان وضعیت خروجی آخرین دستور را چک کرد

پرچم صفر

- پرچم صفر (zero flag) یکی از ساده‌ترین پرچم‌هاست که می‌توان مورد بررسی قرار داد.
- دستورهای «JZ» و «JE» این پرچم را مورد بررسی قرار می‌دهند و در صورتی که مقدار آن برابر با یک بود، پرش انجام می‌شود.
- این پرچم در اثر دستور مقایسه و یا هر عملیات ریاضی که منجر به صفر شود، مقدار یک خواهد گرفت.

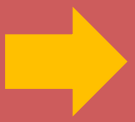
```
movl $30, %eax
subl $30, %eax
jz overthere
```



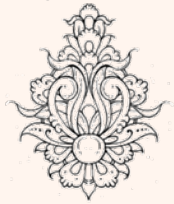
تمرین کلاسی

یک حلقه بنویسید که بدنه‌ی آن (Do Something) ده بار تکرار شود.

```
    movl $10, %edi
loop1:
    < Do somthing>
    dec %edi   #decrement
    jz out
    jmp loop1
out:
```



```
jnz loop1
```



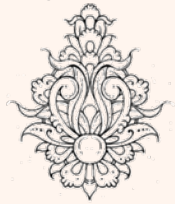
- دو دستورالعمل inc و dec دو دستور تک‌آدرسی هستند که موجب افزایش(کاهش) عملوند خود می‌شوند.
- عملوند آن می‌تواند ثبات یا حافظه باشد.

پرچم سرریز

- این پرچم در هنگام استفاده از اعداد علامت‌دار مورد استفاده قرار می‌گیرد.
- زمانی که یک عدد علامت‌دار، در محدوده‌ی مجاز عملوند مورد نظر قرار نگیرد، این پرچم یک می‌شود.

```
        movl $1,%eax
        movb $0x7f,%bl
        addb $0x10,%bl
        jo overhere
        int $0x80
overhere:
        movl $0,%ebx
        int $0x80
```

```
ahmad@ubuntu:~/MyData/courses/Asm$ echo $?
0
```



پرچم توازن

- این پرچم نشان‌دهنده‌ی تعداد بیت‌های یک، بعد از عملیات ریاضی می‌باشد.
- در صورتی که تعداد بیت‌های حاصل زوج باشند، این پرچم «یک» می‌شود.

```
# paritytest.s - An example of testing the
parity flag
.section .text
.globl _start
_start:
movl $1, %eax
movl $4, %ebx
subl $3, %ebx
jp overhere
int $0x80
overhere:
movl $100, %ebx
int $0x80
```

تنها بایت کم‌ارزش بر روی این پرچم اثر می‌گذارد

در عمل این دستور (jp) کاربردهای دیگری نیز دارد

```
ahmad@ubuntu:~/MyData/courses/Asm$ echo $?
1
```

نشانی‌دهی شاخص

```
values:
.int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
```

- آدرس‌دهی شاخص: هنگامی که از آرایه‌ها استفاده می‌کنیم، این شیوه‌ی نشانی‌دهی به کار می‌آید.
- در این شیوه آدرس یک خانه‌ی حافظه با کمک بخش‌های زیر به دست می‌آید:

آدرس شروع آرایه

– یک آدرس پایه (A base address)

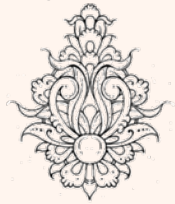
– ثبات آفست (offset) که به آدرس پایه افزوده می‌شود.

– اندازه‌ی داده‌ی مورد استفاده

– ثبات شاخص (index) که محل عنصر مورد نظر را نشان می‌دهد.

```
base_address(offset_address, index, size)
```

اختلاف آدرس عنصر اول از آدرس شروع

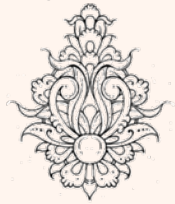


نشانی‌دهی شاخص (ادامه...)

```
base_address(offset_address, index, size)
```

```
base_address + offset_address + index * size
```

```
movl $2, %edi
movl values(, %edi, 4), %eax
```



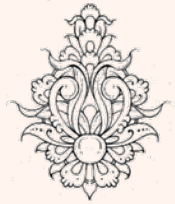
مثال

```
#movtest3.s
.section .data
output:
. asciz "The value is %d\n"
values:
. int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
.section .text
.globl _start
_start:
  nop
  movl $0, %edi
loop:
  movl values(, %edi, 4), %eax
  pushl %eax
  pushl $output
  call printf
  addl $8, %esp
  inc %edi
  cmpl $11, %edi
  jne loop
  movl $0, %ebx
  movl $1, %eax
  int $0x80
```

```
The value is 10
The value is 15
The value is 20
The value is 25
The value is 30
The value is 35
The value is 40
The value is 45
The value is 50
The value is 55
The value is 60
```

پرچم علامت

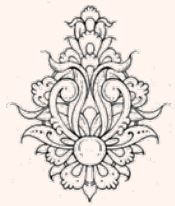
- در بسیاری موارد، مانند پویش عناصر (به صورت معکوس) آرایه که اندیس عنصر نخست صفر است، بررسی پرچم صفر مفید نیست.
- در این موارد از پرچم علامت استفاده می‌شود:



مثال

```
# signtest.s
.section .data
value:
    .int 21, 15, 34, 11, 6, 50, 32, 80, 10, 2
output:
    .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
    movl $9, %edi
loop:
    pushl value(, %edi, 4)
    pushl $output
    call printf
    add $8, %esp
    dec %edi
    jns loop
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

```
The value is: 2
The value is: 10
The value is: 80
The value is: 32
The value is: 50
The value is: 6
The value is: 11
The value is: 34
The value is: 15
The value is: 21
```



برای انجام حرهای تکرارشونده می‌توان از دستورهای پرش شرطی استفاده کرد، اما برای انجام چنین کاری، راه‌های ساده‌تری هم وجود دارد.

پرچم بیت نَقلی

- بیت نقلی، وقوع سرریز در اعداد بدون علامت را نشان می‌دهد.
- بر خلاف پرچم سرریز، inc و dec بر روی این پرچم اثری ندارند.
- همچنین هنگامی که در اعداد بدون علامت حاصل از صفر کم‌تر شود، این پرچم «یک» می‌شود.

```
movl $0xffffffff, %ebx
inc %ebx
jc overflow
```

```
movl $2, %eax
subl $4, %eax
jc overflow
```

```
movl $0xffffffff, %ebx
addl $1, %ebx
jc overflow
```

- این پرچم، دارای دستورهای اختصاصی نیز می‌باشد.

Instruction	Description
CLC	Clear the carry flag (set it to zero)
CMC	Complement the carry flag (change it to the opposite of what is set)
STC	Set the carry flag (set it to one)

SET

Set byte to one on condition

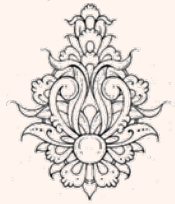
(386+)

```
Setxx reg8/mem8
```

```
(SETA, SETAE, SETB, SETBE, SETC,
SETE, SETG, SETGE, SETL, SETLE,
SETNA, SETNAE, SETNB, SETNBE,
SETNC, SETNE, SETNG, SETNGE,
SETNL, SETNLE, SETNO, SETNP,
SETNS, SETNZ, SETO, SETP, SETPE,
SETPO, SETS, SETZ)
```

Modifies flags: none

در صورت برقراری شرط محتوای مقصد را برابر ۱ قرار می‌دهند، در غیر این صورت آن را صفر می‌کند.



حلقه‌ها

- تنها با استفاده از یک دستور حلقه می‌توان یک فرآیند را چندین بار تکرار نمود.
- در دستورات حلقه مقدار موجود در ثبات ECX با هر بار اجرای دستور کاهش می‌یابد.
- دستور حلقه به صورت زیر مورد استفاده قرار می‌گیرد:

```
loop address
```

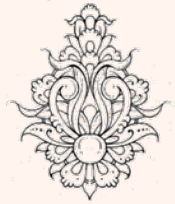
Instruction	Description
LOOP	Loop until the ECX register is zero
LOOPE/LOOPZ	Loop until either the ECX register is zero, or the ZF flag is not set
LOOPNE/LOOPNZ	Loop until either the ECX register is zero, or the ZF flag is set

شیوه‌ی نشانی‌دهی این دستور **pc relative** است.
این دستور تنها می‌تواند از آفست‌های هشت‌بیتی استفاده کند.

حلقه‌ها(ادامه...)

- ساختار حلقه به صورت زیر است:

```
< code before the loop >
movl $100, %ecx
loop1:
< code to loop through >
loop loop1
< code after the loop >
```



- در هنگام استفاده از حلقه‌ها می‌باید مراقب بود که مقدار ECX تغییر داده نشود، به ویژه هنگام استفاده از توابع
- کاهش محتوای ECX بر روی ثبات پرچم اثری نمی‌گذارد.

مثال

```
.section .data
output:
    .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
    movl $100, %ecx
    movl $0, %eax
loop1:
    addl %ecx, %eax
    loop loop1
    pushl %eax
    pushl $output
    call printf
    add $8, %esp
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

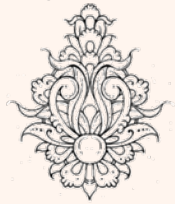
- برنامه‌ای بنویسید که بدون استفاده از فرمول تصاعد حسابی جمع اعداد از یک تا صد را حساب کرده و نتیجه را چاپ کند.

```
ahmad@ubuntu:~/Courses/Assembly/chapter3$ ./exp12
The value is: 5050
```

مثال(ادامه...)

```
.section .data
output:
    .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
    movl $0, %ecx
    movl $0, %eax
    jcxz done
loop1:
    addl %ecx, %eax
    loop loop1
done:
    pushl %eax
    pushl $output
    call printf
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

- با استفاده از دستور loop ابتدا یک واحد از ECX کاسته شده و سپس شرط چک می‌شود.
- در صورتی که ECX دارای مقدار صفر باشد، پس از اجرای این دستور چه می‌شود؟



مثال

- برنامه‌ای بنویسید که اعداد از صد تا یک به صورت نزولی چاپ کند.

```
.section .data
output:
    .asciz "The value is:
   %d\n"
.section .text
.globl _start
_start:
    nop
    movl $100, %ecx
loop1:
    pushl %ecx
    pushl $output
    call printf
    add $8, %esp
    loop loop1
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

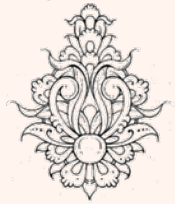
```
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673
The value is: -1082058673^C
```

مثال

- برنامه‌ای بنویسید که اعداد از صد تا یک به صورت نزولی چاپ کند.

```
.section .data
output:
    .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
    nop
    movl $100, %ecx
loop1:
    pushl %ecx
    pushl $output
    call printf
    add $4, %esp
    popl %ecx
    loop loop1
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

```
The value is: 25
The value is: 24
The value is: 23
The value is: 22
The value is: 21
The value is: 20
The value is: 19
The value is: 18
The value is: 17
The value is: 16
The value is: 15
The value is: 14
The value is: 13
The value is: 12
The value is: 11
The value is: 10
The value is: 9
The value is: 8
The value is: 7
The value is: 6
The value is: 5
The value is: 4
The value is: 3
The value is: 2
The value is: 1
```

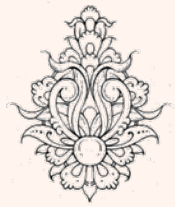


ساختارهای کنترلی در زبان‌های سطح بالا

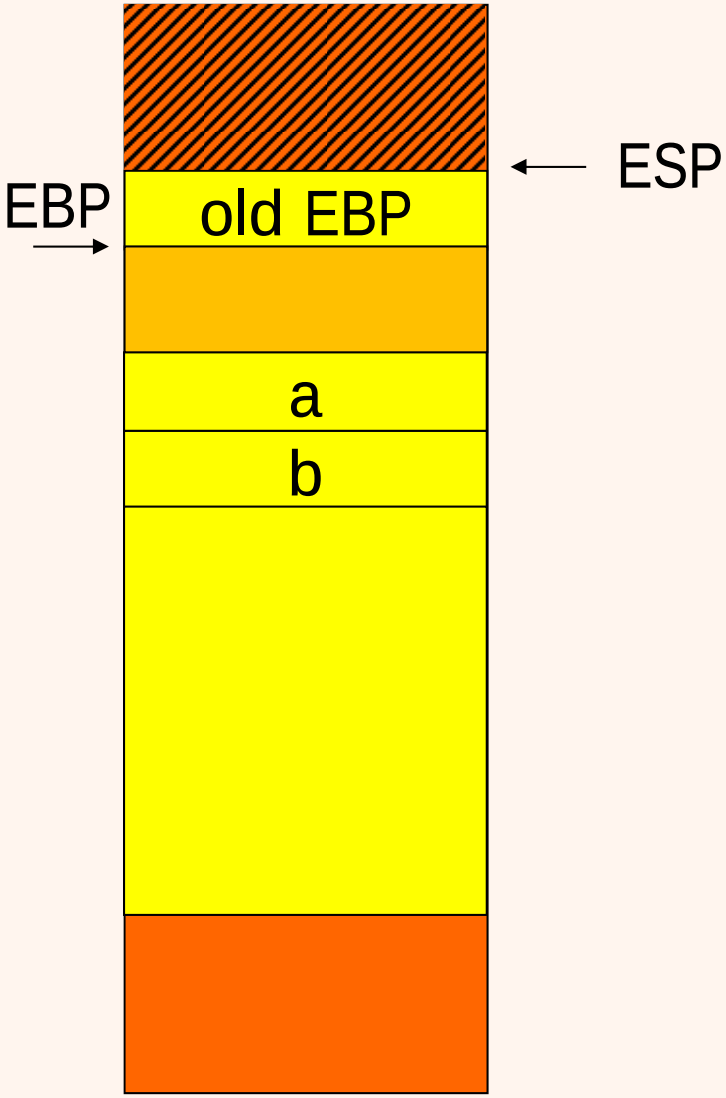
دستور if

```c
#include <stdio.h>
int main()
{
   int a = 100;
   int b = 25;
   if (a > b)
   {
      printf("The higher value is %d\n", a);
   } else
      printf("The higher value is %d\n", b);
   return 0;
}
```

```
        .file   "ifthen.c"
        .section        .rodata
.LC0:
        .string         "The higher value is %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
```



متغیرهای محلی




```
pushl    %ebp
movl %esp, %ebp
```

```
andl $-16, %esp
```

با این دستور شروع حافظه برای متغیرهای محلی مضربی از شانزده خواهد شد.

```
subl $32, %esp
```

بدین ترتیب سی‌ودو بایت فضا از روی پشته برای متغیرهای محلی گرفته می‌شود.

هم‌ترازی

Byte alignment

	Odd Bank	Even Bank	
F	90	87	E
D	E9	11	C
B	F1	24	A
9	01	46	8
7	76	DE	6
5	14	33	4
3	55	12	2
1	AB	FF	0

Data Bus (15:8)	Data Bus (7:0)

Address 0

data(15:8)=AB,data(7:0)=FF

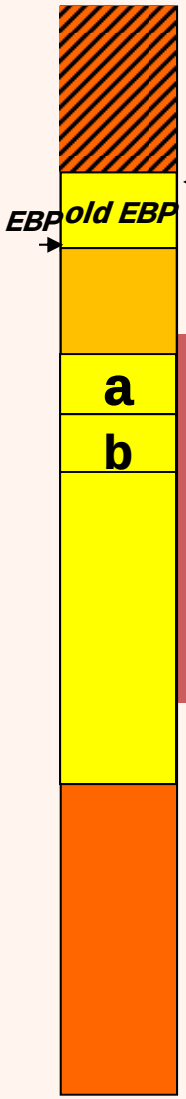
Address 1

data(15:8)=AB,data(7:0)=12

data(15:8)=12,data(7:0)=AB

مشکل هم‌ترازی توسط سخت‌افزار حل می‌شود، اما اگر خواهان برنامه‌های سریع‌تر هستید، بهتر است به صورت هم‌تراز شده از حافظه استفاده کنید.

ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)



۱

```
movl    $100, 28(%esp)
movl    $25, 24(%esp)
```

a

b

دستور if

```
movl    28(%esp), %eax
cmpl    24(%esp), %eax
jle     .L2
movl    $.LC0, %eax
movl    28(%esp), %edx
movl    %edx, 4(%esp)
movl    %eax, (%esp)
call    printf
jmp     .L3
```

a-b≤0

a>b-then

۲

!(a>b)-else

۳

```
.L2:
        movl    $.LC0, %eax
        movl    24(%esp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    printf
```

۴

```
.L3:
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident "GCC: (Ubuntu
4.4.3-4ubuntu5) 4.4.3"
        .section        .note.GNU-
stack,"",@progbits
```

```
movl    %ebp,%esp
pop     %ebp
```

```c
#include <stdio.h>
int main()
{
   int a = 100;
   int b = 25;
   if (a > b)
   {
      printf("The higher value is %d\n", a);
   } else
      printf("The higher value is %d\n", b);
   return 0;
}
```

ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

با یک کامپایلر دیگر

```
.file "ifthen.c"
.section .rodata
.LC0:
.string "The higher value is %d\n"
.text
.globl main
.type main, @function
main:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
andl $-16, %esp
movl $0, %eax
subl %eax, %esp
```

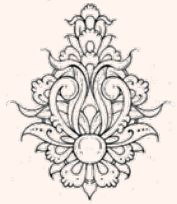
```
movl $100, -4(%ebp)
movl $25, -8(%ebp)
movl -4(%ebp), %eax
cmpl -8(%ebp), %eax
jle .L2
movl -4(%ebp), %eax
movl %eax, 4(%esp)
movl $.LC0, (%esp)
call printf
jmp .L3
```

```
.L2:
movl -8(%ebp), %eax
movl %eax, 4(%esp)
movl $.LC0, (%esp)
call printf
```

```
.L3:
movl $0, (%esp)
call exit
.size main, .-main
.section .note.GNU-stack,"",@progbits
.ident "GCC: (GNU) 3.3.2 (Debian)"
```

دستورات جابجایی اعداد علامت‌دار

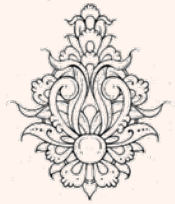
- در زبان‌های سطح بالا به سادگی نوع داده‌ی مورد نظر را انتخاب می‌کنیم.
- در زبان اسمبلی تفسیر مناسب داده‌های موجود در حافظه و ثبات‌ها از وظایف برنامه‌نویس است.
- در خانواده‌ی IA-32 انواع داده‌ای مختلفی مورد استفاده قرار می‌گیرند.



نوع داده‌ی صحیح

- **Byte:** 8 bits
- **Word:** 16 bits
- **Doubleword:** 32 bits
- **Quadword:** 64 bits

- اعداد صحیح را می‌توان به صورت علامت‌دار (مکمل ۲) و بدون علامت ذخیره کرد.
- سوال: جمع و تفریق اعداد علامت‌دار با اعداد بدون علامت چه تفاوتی دارد؟



مثال

```
# exp15.s
.section .data
data1:
    .int -16
data2:
     .quad -1
output:
      .string    "\n%u\t%d\n"
.section .text
```

```
.globl _start
_start:
    nop
    movl data1, %eax
    movl data2, %ebx
    movl $-345, %ecx
    movw $0xffb1, %dx
    pushl %ebx
    pushl %ebx
    pushl $output
    call printf
    addl $12,%esp

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

```
(gdb) info reg
eax            0xfffffff0       -16
ecx            0xfffffea7       -345
edx            0x11ffb1 1179569
ebx            0xffffffff       -1
```

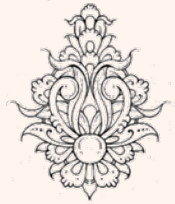
```
ahmad@ubuntu:~/Courses/Assembly/chapter3$ ./exp15

4294967295      -1
```

نوع داده‌ی صحیح (ادامه...)

- هنگامی که عددی با طول بیت کم‌تر را به ثبات با طول‌بیت بیشتر انتقال می‌دهیم، باید از این که بیت‌های پرارزش دارای مقدار مناسب هستند، اطمینان حاصل شود.
- برای اعداد بدون علامت باید بیت‌های پرارزش صفر شوند.

%ax⟶%ebx

```
movl $0, %ebx
movw %ax, %bx
```



جابجایی اعداد بدون علامت

```
movzx source, destination
```

(386+)

Move with Zero Extend

ثبات یا حافظه

فقط ثبات

این دستور برای انتقال اعداد بدون علامت با طول کم‌تر به ثبات با طول بیشتر به کار می‌رود. این دستور، پرچم‌ها را تحت تأثیر قرار نمی‌دهد.

```
# example17.s
.section .text
.globl _start
_start:
    nop
    movl $384, %ecx
    movzx %cl, %ebx
    movl $1, %eax
    int $0x80
```

۳۸۴=۲۵۶+۱۲۸

```
(gdb) print $ebx
$1 = 128
(gdb)
```

Sign extension

جابجایی اعداد علامت‌دار

-1 (11111111) → 0000000011111111

۲۵۵

چگونه می‌توان در محیطی که هم اعداد علامت دار و هم اعداد بدون علامت با اندازه‌های مختلف وجود دارند، از این اعداد استفاده نمود؟

-1 (11111111) → 1111111111111111

-۱

movsx

(386+)

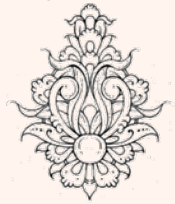
Move with Sign Extend

مثال-جابجایی اعداد علامت‌دار

```
# movsxtest.s - An
example of the MOVSX
instruction
.section .text
.globl _start
_start:
    nop
    movw $-79, %cx
    movl $0, %ebx
    movw %cx, %bx
    movsx %cx, %eax
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

0xffb1

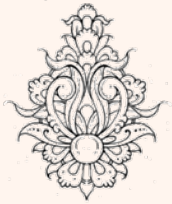
```
eax            0xffffffb1       -79
ecx            0x12ffb1 1245105
edx            0x11e0c0 1171648
ebx            0xffb1   65457
```



مثال-جابجایی اعداد علامت‌دار

```
# movsxtest2.s - Another
example using the MOVSX
instruction
.section .text
.globl _start
_start:
    nop
    movw $79, %cx
    xor %ebx, %ebx
    movw %cx, %bx
    movsx %cx, %eax
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

```
eax            0x4f       79
ecx            0x12004f   1179727
edx            0x11e0c0   1171648
ebx            0x4f       79
```



```
8048101:       31 db                    xor    %ebx,%ebx
8048103:       bb 00 00 00 00           mov    $0x0,%ebx
```

قرینه کردن

NEG-Two's Complement Negation

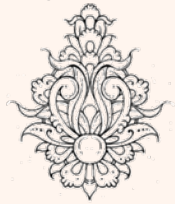
```
negx destination
```

این دستور معادل مکمل دو عملوند را محاسبه کرده و در آن قرار می‌دهد.

NOT-One's Compliment Negation (Logical NOT)

```
notx destination
```

این دستور نقیض بیتی است. (مکمل ۱)



ضرب اعداد بدون علامت

- بر خلاف جمع و تفریق این عملیات پیچیده به دستورات جداگانه‌ای برای اعداد علامت‌دار و بدون علامت احتیاج دارد.

Unsigned Multiply

```
mulx source
```

عملوند دیگر کجاست؟

عملوند دیگر به صورت ضمنی مشخص است

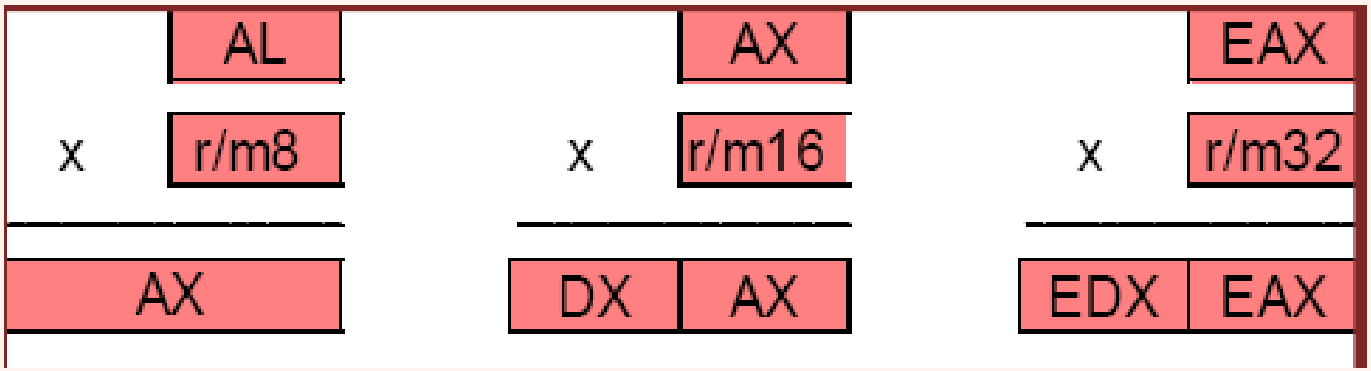
بسته به طول عملوندهای منبع، مقصد در ثبات edx:eax قرار می‌گیرد.

source	Implied source	destination
b: 8 bit	AL	AX
w: 16 bit	AX	DX:AX
l: 32 bit	EAX	EDX:EAX

Backward compatibility

ضرب اعداد بدون علامت

Jane Moorhead/ECE 3724




```
movb $0x5, %al
movb $0x10, %bl
mulb %bl
```

AX = 0050h

CF = 0

در این مثال سرریز رخ نمی‌دهد

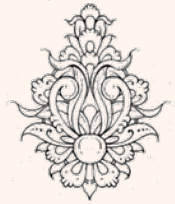
مثال

```
.section .data
val1:
.short 0x2000
val2:
.short 0x0100
.section .text
movw val1,%ax
mulw val2
```

DX:AX = 00200000h

CF = 1

در این مثال سرریز رخ می‌دهد



مثال

```
# multest.s
.section .data
data1:
    .int 315814
data2:
    .int 165432
result:
    .quad 0
output:
    .asciz "The result is %qd\n"
.section .text
```

```
.globl _start
_start:
    nop
    movl data1, %eax
    mull data2
    movl %eax, result
    movl %edx, result+4
    pushl %edx
    pushl %eax
    pushl $output
    call printf
    add $12, %esp
    pushl $0
    call exit
```

```
ahmad@ubuntu:~/Courses/Assembly/chapter3$ ./multest
The result is 52245741648
```

Signed Multiply

ضرب اعداد علامت‌دار

imulx

این دستور در ۸۰۲۸۶ به بعد مطرح شد

از نظر ترتیب عملگرها، دقیقا مانند حالت قبل است

پرچم سرریز و رقم نقلی زمانی «یک» می‌شوند، که بخش پرارزش معادل بیت علامت بخش کم ارزش نباشد

```
imulx multiplier,source/destination
```

```
imulx multiplier,source,destination
```

در این حالت «ضرب کننده» عدد ثابتی است

```
imulx source, destination
```

+۳۸۶

در این حالت منبع می‌تواند ثبات یا حافظه‌ی ۱۶ یا ۳۲ بیتی باشد و مقصد تنها ثبات ۱۶ یا ۳۲ بیتی

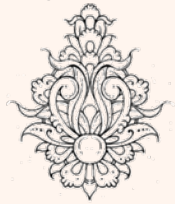
مثال

```
movb $48, %al
movb $4, %bl
imulb %bl
```

AX = 00C0h

OF = 1

بیت‌های Ah معادل بیت علامت Al نیستند



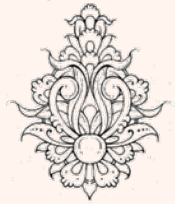
مثال

```
movb $4, %al
movb $-4, %bl
imulb %bl
```

AX = FFF0h

OF = 0

بیت‌های Ah معادل بیت علامت Al هستند

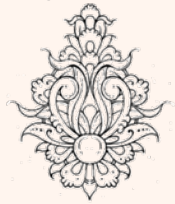


مثال

```
# imultest2.s - An example
of detecting an IMUL
overflow
.section .text
.globl _start
_start:
    nop
    movw $680, %ax
    movw $100, %cx
    imulw %cx
    jo over
    movl $1, %eax
    movl $0, %ebx
    int $0x80
over:
    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

```
8           imulw %cx
(gdb) print $eflags
$1 = [ SF IF ID ]
(gdb) s
9           jo over
(gdb) print $eflags
$2 = [ CF SF IF OF ID ]
(gdb) s
14          movl $1, %eax
(gdb) s
15          movl $1, %ebx
```

```
(gdb) print/x $eax
$6 = 0x1209a0
(gdb) print/x $edx
$7 = 0x110001
```



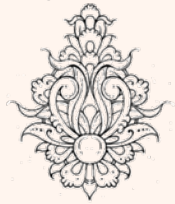
تقسیم بدون علامت

مقسوم‌علیه

```
divx divisor
```

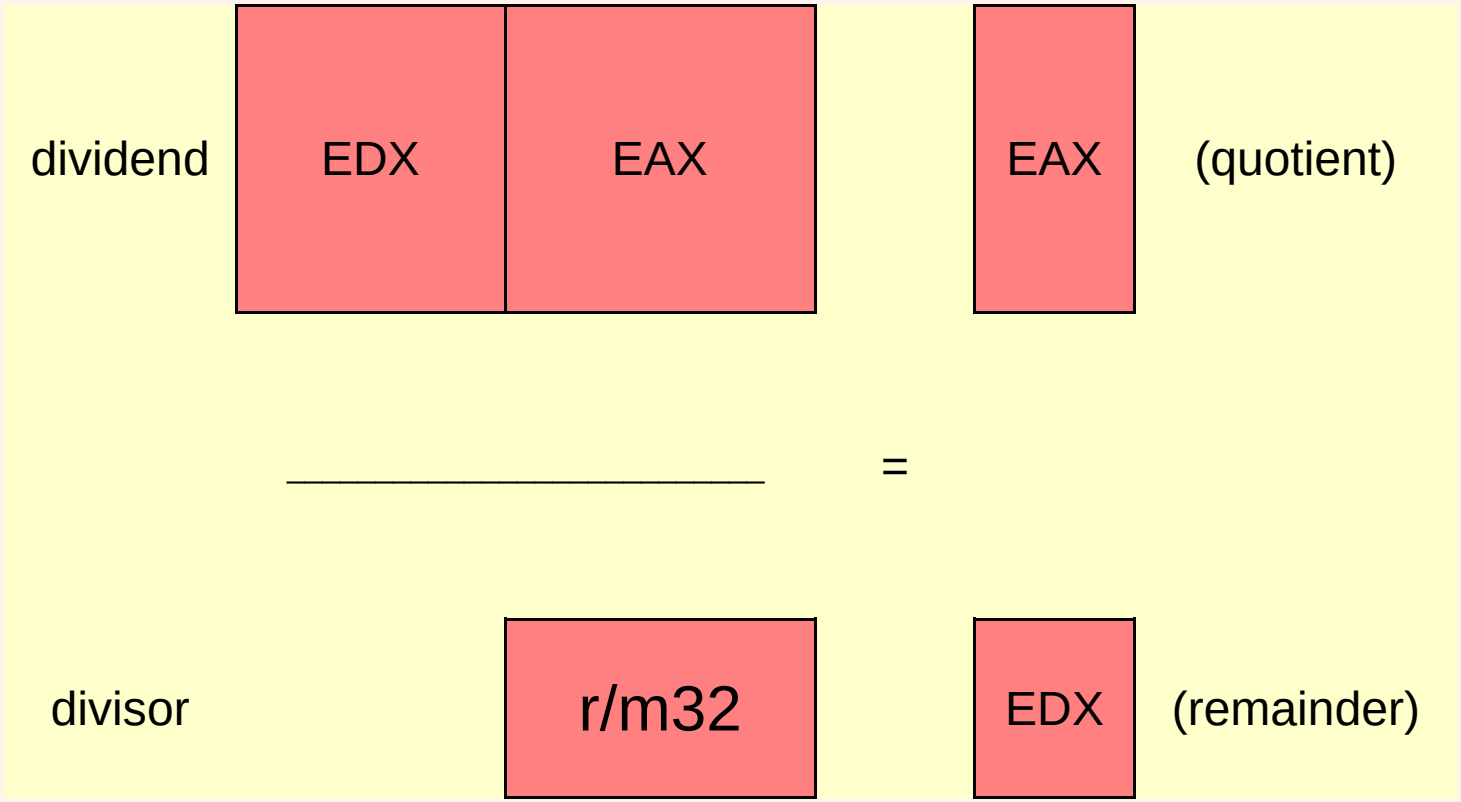
- دستور تقسیم صحیح، علاوه بر «خارج قسمت»،
- «باقیمانده» را هم برمی‌گرداند.
- حداکثر اندازه‌ی «مقسوم‌علیه» بستگی به اندازه «مقسوم» دارد و می‌تواند هشت، شانزده و سی‌ودوبیتی باشد.
- مقسوم‌علیه می‌تواند ثبات یا حافظه باشد.

مقسوم	مقسوم علیه	خارج قسمت	باقیمانده
Dividend	Maximum value allowed for the divisor	Quotient	Reminder
AX	b: 8 bit	AL	AH
DX:AX	w: 16 bit	AX	DX
EDX:EAX	l: 32 bit	EAX	EDX



ساختار تقسیم

Jane Moorhead/ECE 3724



تقسیم بر صفر؟؟؟

```
Floating point exception
```

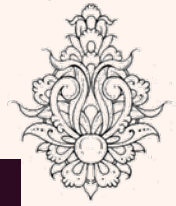
مثال

```
movl $0x43, %eax
movb $0x3, %bl
divb %bl
```

```
(gdb) print/x $eax
$3 = 0x116
```

```
movw $0,%dx
mov  $0x1391, %ax
movw $0x100, %cx
divw %cx
```

```
(gdb) print/x $eax
$3 = 0x120013
(gdb) print/x $edx
$4 = 0x110091
```

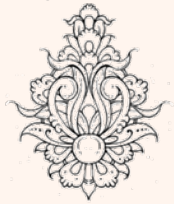


idivx divisor

تقسیم علامت‌دار

- در تقسیم اعداد علامت‌دار باقیمانده با مقسوم هم‌علامت خواهد بود.

- در تقسیم اعداد علامت‌دار با توجه به این اندازه‌ی مقسوم دوبرابر مقسوم‌علیه است، در برخی موارد استفاده از دستورات همراه با گسترش بیت علامت می‌باید مورد استفاده قرار گیرد.



دستورالعمل‌های گسترش بیت علامت

Sign Extension Instructions

CBW — Convert byte to word

Sign-extend **al→ax**

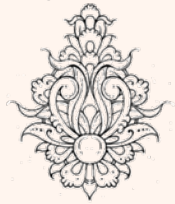
CWD — Convert word to double

Sign-extend **ax→dx:ax**

CDQ (+۳۸۶) — Convert double to quadword

Sign-extend **eax→edx:eax**

Modifies flags: None



انتقال بیتی- شیفت

- عملیات ضرب و تقسیم در مقایسه با جمع و تفریق بسیار زمان‌بر هستند.
- با استفاده از دستورالعمل‌های شیفت می‌توان ضرب و تقسیم را با سرعت بیشتری انجام داد.

ضرب با شیفت

```
salx/shlx destination
```

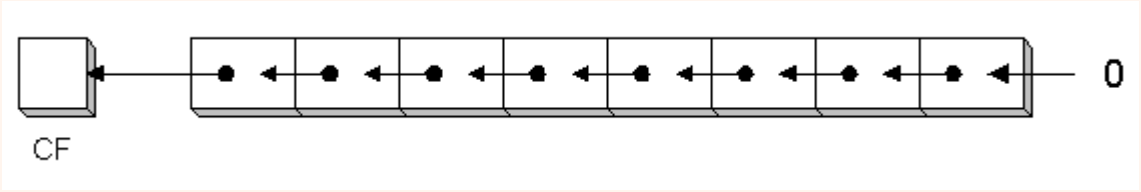
Modifies flags: CF OF PF SF ZF

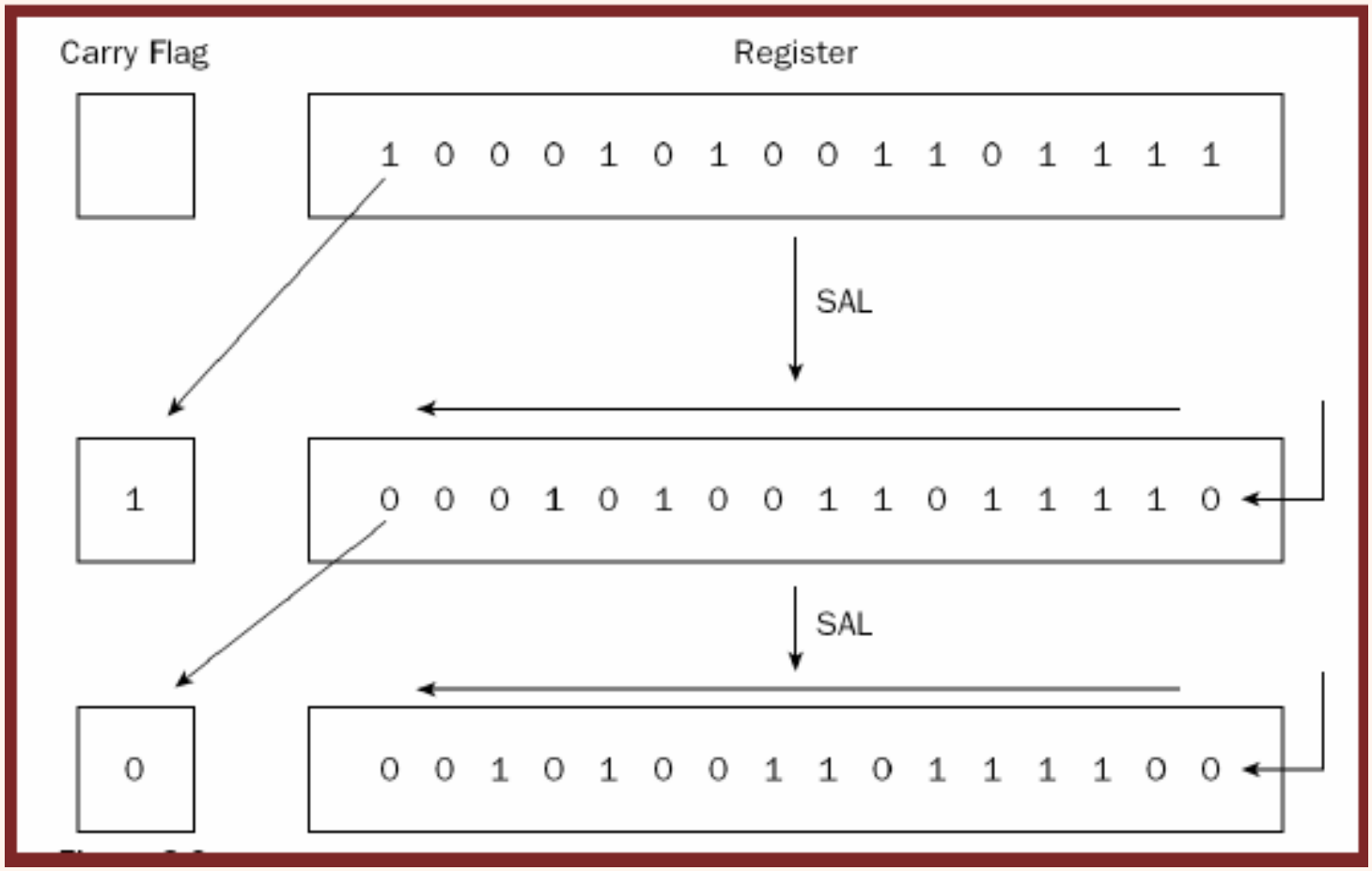
```
salx/shlx %cl, destination
```

```
salx/shlx shifter, destination
```

Shift Arithmetic Left / Shift Logical Left

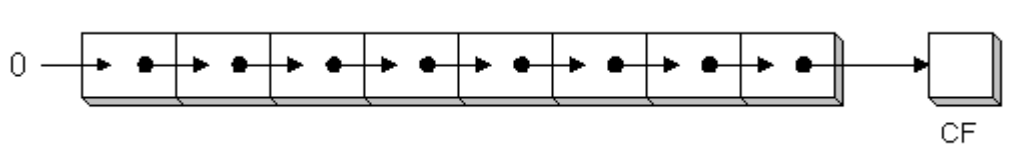
شیفت به چپ


0
CF

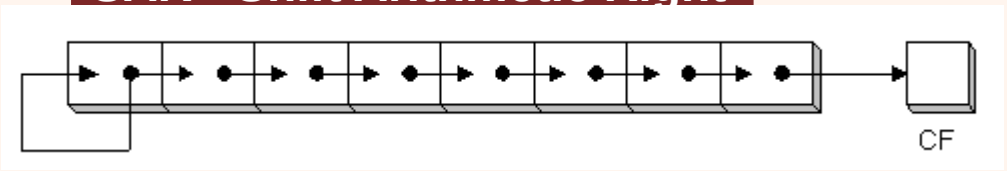

Carry Flag
Register
1 0 0 0 1 0 1 0 0 1 1 0 1 1 1 1
SAL
1
0 0 0 1 0 1 0 0 1 1 0 1 1 1 1 0
SAL
0
0 0 1 0 1 0 0 1 1 0 1 1 1 1 0 0

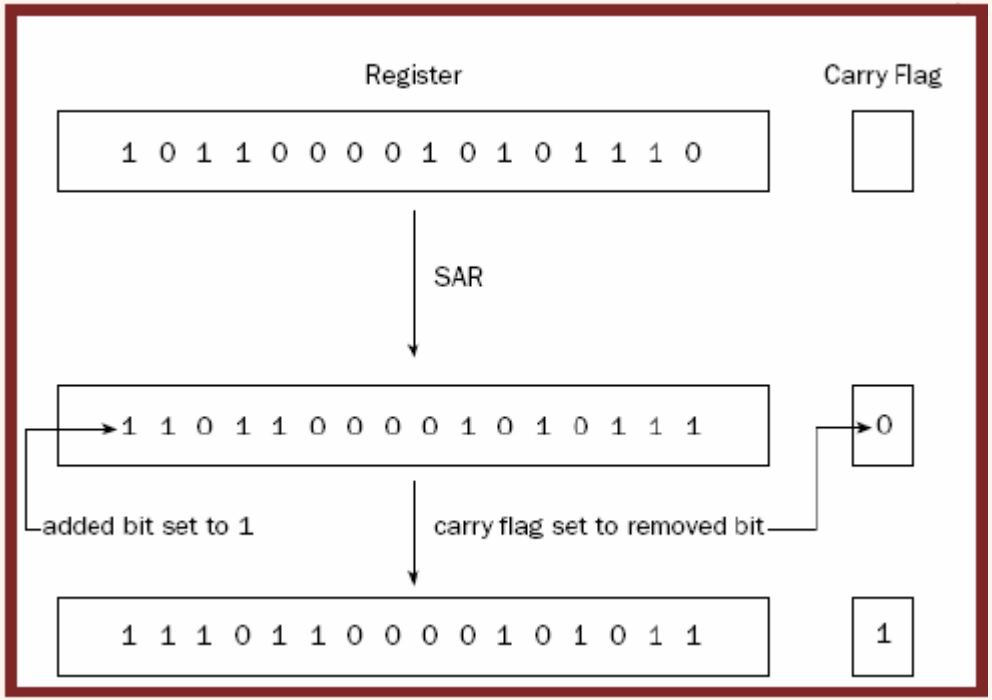
شیفت به راست

SHR - Shift Logical Right



SAR - Shift Arithmetic Right



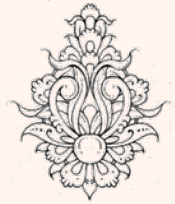


Modifies flags: CF OF PF SF ZF (AF undefined)

شیفت همراه با چرخش

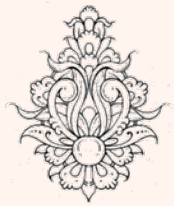
- این دستور همانند شیفت معمولی است، با این تفاوت که بیت خارج شده از سوی دیگر وارد می‌شود.
- شیوه‌ی استفاده از آن همانند شیفت معمولی است.
- در دو دستور آخر جدول، بیت نقلی را هم در چرخش دخالت می‌دهند.

Instruction	Description
ROL	Rotate value left
ROR	Rotate value right
RCL	Rotate left and include carry flag
RCR	Rotate right and include carry flag



مثال

```c
//ifthen2.c
#include <stdio.h>

int main(){
    int a = 0;
    int b = 25;
    if (a++ && --b)
            printf("Then Part\t%d\t%d\n", a, b);
    else
            printf("Else Part\t%d\t%d\n", a, b);
    return 0;
}
```



دستورهای بیتی

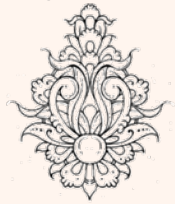
- در مجموعه دستورالعمل‌های خانواده‌ی IA-32 دستورهای منطقی نیز پیش‌بینی شده است:

- AND
- OR
- XOR
- NOT

```
andx source, destination
```

- TEST

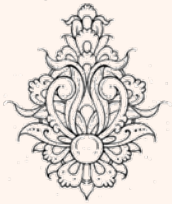
این دستور شبیه and است با این تفاوت که نتیجه در مقصد نوشته نمی‌شود و فقط روی flagها اثر می‌گذارد



ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

دستور if

```
if:
<condition to evaluate>
jxx else ; jump to the else part if the
   condition is false
<code to implement the "then" statements>
jmp end ;jump to the end
else:
< code to implement the "else" statements>
end:
```

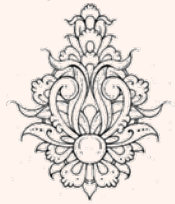


مثال

```
if (eax < ebx) && (eax == ecx) then
```

اعداد علامت‌دار هستند

```
if:
   cmpl %ebx, %eax
   jnl else
   cmpl %eax, %ecx
   jne else
then:
      < then logic code>
      jmp end
else:
   < else logic code >
   end:
```

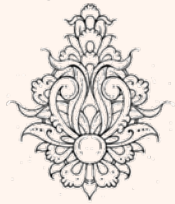


مثال

```
if (eax < ebx) || (eax == ecx) then
```

اعداد علامت‌دار هستند

```
if:
   cmpl %ebx, %eax
   jl then
   cmpl %eax, %ecx
   jne else
then:
      < then logic code>
      jmp end
else:
   < else logic code >
   end:
```



مثال

```c
//ifthen2.c
#include <stdio.h>

int main(){
    int a = 0;
    int b = 25;
    if (a++ && --b)
        printf("Then Part\t%d\t%d\n", a, b);
    else
        printf("Else Part\t%d\t%d\n", a, b);
    return 0;
}
```

```
        .file   "ifthen2.c"
        .section        .rodata
.LC0:
        .string "Then Part\t%d\t%d\n"
.LC1:
        .string "Else Part\t%d\t%d\n"
        .text
```

```
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
```

مثال

a

```
movl    $0, 28(%esp)
movl    $25, 24(%esp)
```

b

۱

```
cmpl    $0, 28(%esp)
setne   %al
addl    $1, 28(%esp)
testb   %al, %al
je      .L2
subl    $1, 24(%esp)
cmpl    $0, 24(%esp)
je      .L2
```

THEN

۲

```
movl    $.LC0, %eax
movl    24(%esp), %edx
movl    %edx, 8(%esp)
movl    28(%esp), %edx
movl    %edx, 4(%esp)
movl    %eax, (%esp)
call    printf
jmp     .L3
```

```
.L2:
        movl    $.LC1, %eax
        movl    24(%esp), %edx
        movl    %edx, 8(%esp)
        movl    28(%esp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    printf
```

۳

ELSE

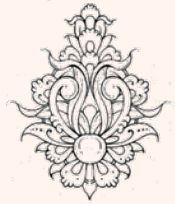
```
//ifthen2.c
#include <stdio.h>

int main(){
   int a = 0;
   int b = 25;
   if (a++ && --b)
         printf("Then Part\t%d\t%d\n", a, b);
   else
         printf("Else Part\t%d\t%d\n", a, b);
   return 0;
}
```

ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

دستور if

```c
#include <stdio.h>
int main()
{
    int a = 1;
    int b = 0;
    if (a-- || b++)
    {
        printf("THEN: a=%d\tb=%d\n",a , b);
    } else
        printf("ELSE: a=%d\tb=%d\n",a , b);
    return 0;
}
```

```
ahmad@ubuntu:~/Assembly/code/chap06$ ./ifthen2
THEN: a=0       b=0
ahmad@ubuntu:~/Assembly/code/chap06$
```



ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

```
main:
    pushl       %ebp
    movl %esp, %ebp
    andl $-16, %esp
    subl $32, %esp
    movl $1, 28(%esp)
    movl $0, 24(%esp)
    cmpl $0, 28(%esp)
    setne %al
    subl $1, 28(%esp)
    testb %al, %al
    jne  .L2
    cmpl $0, 24(%esp)
    setne %al
    addl $1, 24(%esp)
    testb %al, %al
    je   .L3
```

a

b

دستور if

THEN

```
.L2:
    movl $.LC0, %ea
    movl 24(%esp), %edx
    movl %edx, 8(%esp)
    movl 28(%esp), %edx
    movl %edx, 4(%esp)
    movl %eax, (%esp)
    call printf
    jmp  .L4
.L3:
    movl $.LC1, %eax
    movl 4(%esp), %edx
    movl %edx, 8(%esp)
    movl
    movl
    movl
    call
```

ELSE

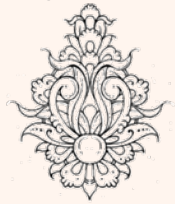
```c
#include <stdio.h>
int main()
{
   int a = 1;
   int b = 0;
   if (a-- || b++)
   {
      printf("THEN: a=%d\tb=%d\n",a , b);
   } else
      printf("ELSE: a=%d\tb=%d\n",a , b);
   return 0;
}
```

ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

دستور for

```c
/* for.c | A sample C for program */
#include <stdio.h>

int main()
{
   int i = 0;
   int j;
   for (i = 0; i < 1000; i++)
   {
      j = i * 5;
      printf("The answer is %d\n", j);
   }
   return 0;
}
```

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
```



ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

دستور for

```
movl    $0, 28(%esp)      i
movl    $0, 28(%esp)
jmp     .L2
```

۱

```
.L2:
cmpl    $999, 28(%esp)
jle     .L3

movl    $0, %eax
leave
ret
```

۳

```
.L3:
movl    28(%esp), %edx
movl    %edx, %eax
sall    $2, %eax
addl    %edx, %eax
movl    %eax, 24(%esp)    j
movl    $.LC0, %eax
movl    24(%esp), %edx
movl    %edx, 4(%esp)
movl    %eax, (%esp)
call    printf
addl    $1, 28(%esp)
```

۲

```c
/* for.c | A sample C for program */
#include <stdio.h>

int main()
{
   int i = 0;
   int j;
   for (i = 0; i < 1000; i++)
   {
      j = i * 5;
      printf("The answer is %d\n", j);
   }
   return 0;
}
```

Load Effective Address

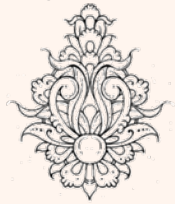
LEA

```
LEA src(mem), dest(reg)
```

Modifies flags: None

این دستور، آدرس عملوند منبع را محاسبه کرده
و در ثبات مقصد می‌ریزد.

به نظر شما این دستور چه کاربردهایی می‌تواند
داشته باشد؟



ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

```
main:
pushl %ebp
movl %esp, %ebp
subl $24, %esp
andl $-16, %esp
movl $0, %eax
subl %eax, %esp
movl $0, -4(%ebp)
movl $0, -4(%ebp)
```

۱

i

```
.L3:
movl $0, (%esp)
call exit
.size main, .-main
.section .note.GNU-
stack,"",@progbits
.ident "GCC: (GNU)
3.3.2 (Debian)"
```

۴

```
.L2:
cmpl $999, -4(%ebp)
jle .L5
jmp .L3
```

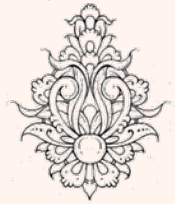
۲

```
.L5:
movl -4(%ebp), %edx
movl %edx, %eax
sall $2, %eax
addl %edx, %eax
movl %eax, -8(%ebp)
movl -8(%ebp), %eax
movl %eax, 4(%esp)
movl $.LC0, (%esp)
call printf
leal -4(%ebp), %eax
incl (%eax)
jmp .L2
```

۳

j

```
    for (i = 0; i < 1000; i++)
    {
        j = i * 5;
        printf("The answer is %d\n", j);
    }
    return 0;
}
```



دستور for با کامپایلر متفاوت

ساختارهای کنترلی در زبان‌های سطح بالا (ادامه...)

دستور for

```
for:
    <condition to evaluate for loop counter value>
    jxx forcode; jump to the code of the condition is true
    jmp end; jump to the end if the condition is false
forcode:
    < for loop code to execute>
    <increment for loop counter>
jmp for; go back to the start of the For statement
end:
```

به صورت‌های مختلفی می‌توان معادل زبان ماشین این تکه کد را نوشت.