بهینه‌سازی

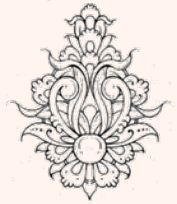
زبان ماشین و اسمبلی
(۱۳-۱۱-۰۰۵)

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

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

فهرست مطالب

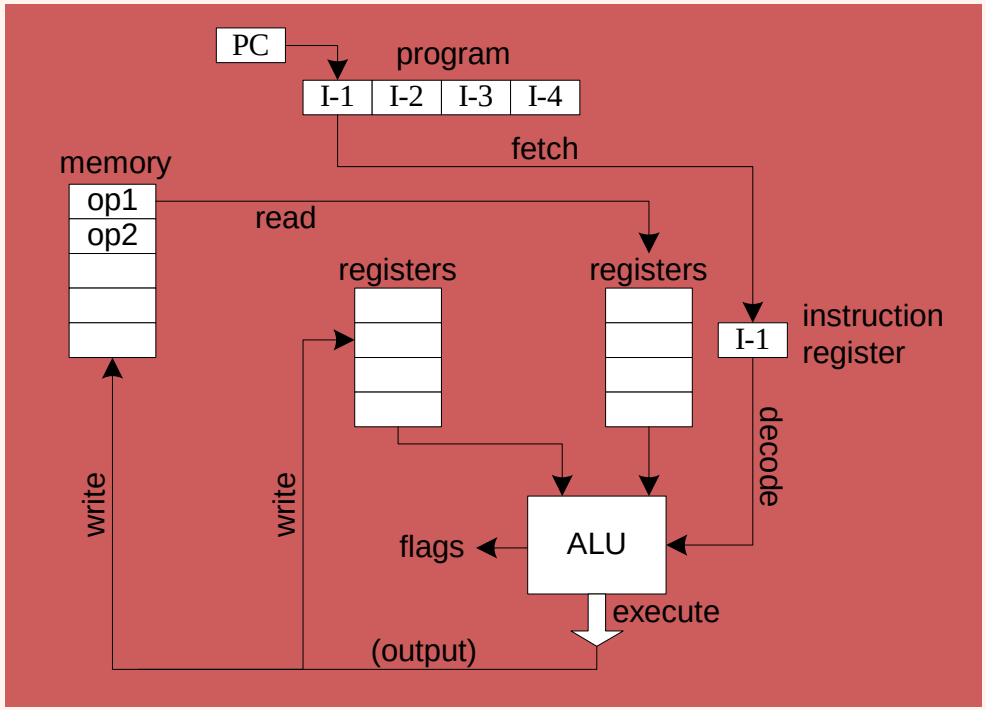
- آشنایی مقدماتی با خط لوله
  - موازی‌سازی در سطح دستورالعمل
- آشنایی مقدماتی با حافظه‌ی نهان
  - همجواری
- نقش دستورهای انشعاب
- پیش‌بینی انشعاب‌هی شرطی
- باز کردن حلقه



گام‌های مختلف اجرای یک برنامه

Instruction execution cycle

- پیش از این با مراحل اجرای یک دستور در پنج گام آشنا شدیم:
  - واکشی (Fetch)
  - رمزگشایی (Decode)
  - خواندن عملوندها (Fetch operands)
  - اجرای دستور (Execute)
  - نوشتن نتیجه‌ی دستورالعمل (Store output)



یادآوری

اجرای دستورات به صورت سری

Stages

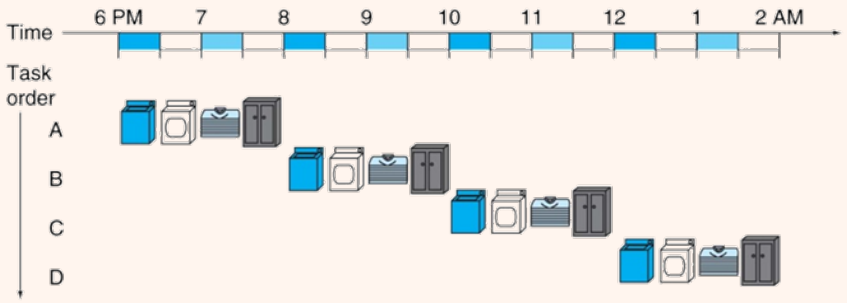
Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2		I-1				
3			I-1			
4				I-1		
5					I-1	
6						I-1
7	I-2					
8		I-2				
9			I-2			
10				I-2		
11					I-2	
12						I-2

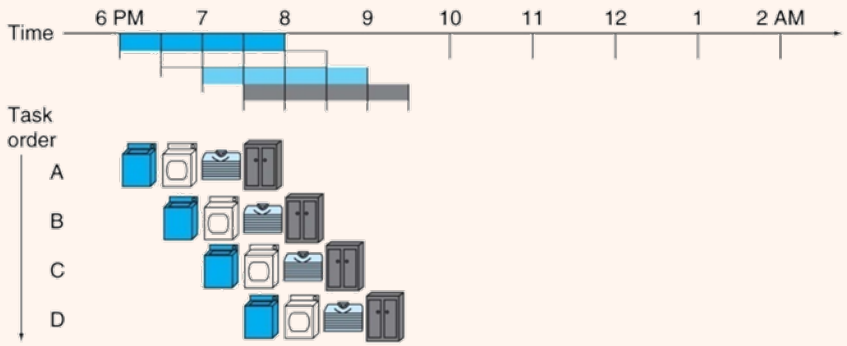
- تا پیش از پردازنده‌های ۸۰۴۸۶، اجرای دستورات به صورت ترتیبی در شش مرحله صورت می‌پذیرفت. در پردازنده‌های ۸۰۴۸۶ این شش مرحله به صورت «خط لوله» در آمد.

مروری بر خط لوله

- در یک سیستم «خط لوله»، اجرای چندین دستورالعمل دارای همپوشانی است.
- پایه‌ی «خط لوله» شبیه خط تولید کارخانه‌هاست.
- تقریبا در تمامی پردازنده‌های موجود از این تکنیک استفاده می‌شود.

مثالی از خط لوله (در رختشوی‌خانه)





- در صورتی‌که کارها را با همپوشانی انجام دهیم، کارایی افزایش چشم‌گیری خواهد داشت.

Speedup= 8/3.5 = 2.3

توان عملیاتی در برابر زمان پاسخ

- زمان پاسخ(اجرا):
  - زمانی که طول می‌کشد تا یک کامپیوتر کاری را تمام کند.
- توان عملیاتی:
  - تعداد کارهایی که در واحد زمان انجام می‌شوند.

throughput versus response time

خط لوله (pipeline)

- «خط لوله» یک شگرد پیاده‌سازی است که در آن چندین دستورالعمل به طور هم‌پوشان (overlapped) به اجرا در می‌آید. در پردازنده‌های خانواده‌ی X86 نخستین بار در ۸۰۴۸۶ از خط لوله استفاده شد.

Stages

Cycles	S1	S2	S3	S4	S5	S6
1	I-1					
2	I-2	I-1				
3		I-2	I-1			
4			I-2	I-1		
5				I-2	I-1	
6					I-2	I-1
7						I-2

در یک خط لوله‌ی k مرحله‌ای، برای انجام n دستور چند سیکل ساعت زمان لازم است؟

$k + (n - 1)$

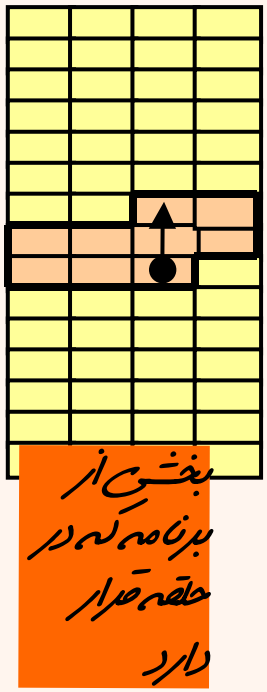
همجواری

Temporal locality

- **همجواری زمانی**: بخشی از حافظه که مورد مراجعه قرار گرفته است، با احتمال بالایی مجدداً مورد استفاده قرار می‌گیرد.
  - متغیرها، و یا دستورهایی که در حلقه قرار گرفته‌اند.

Spatial locality

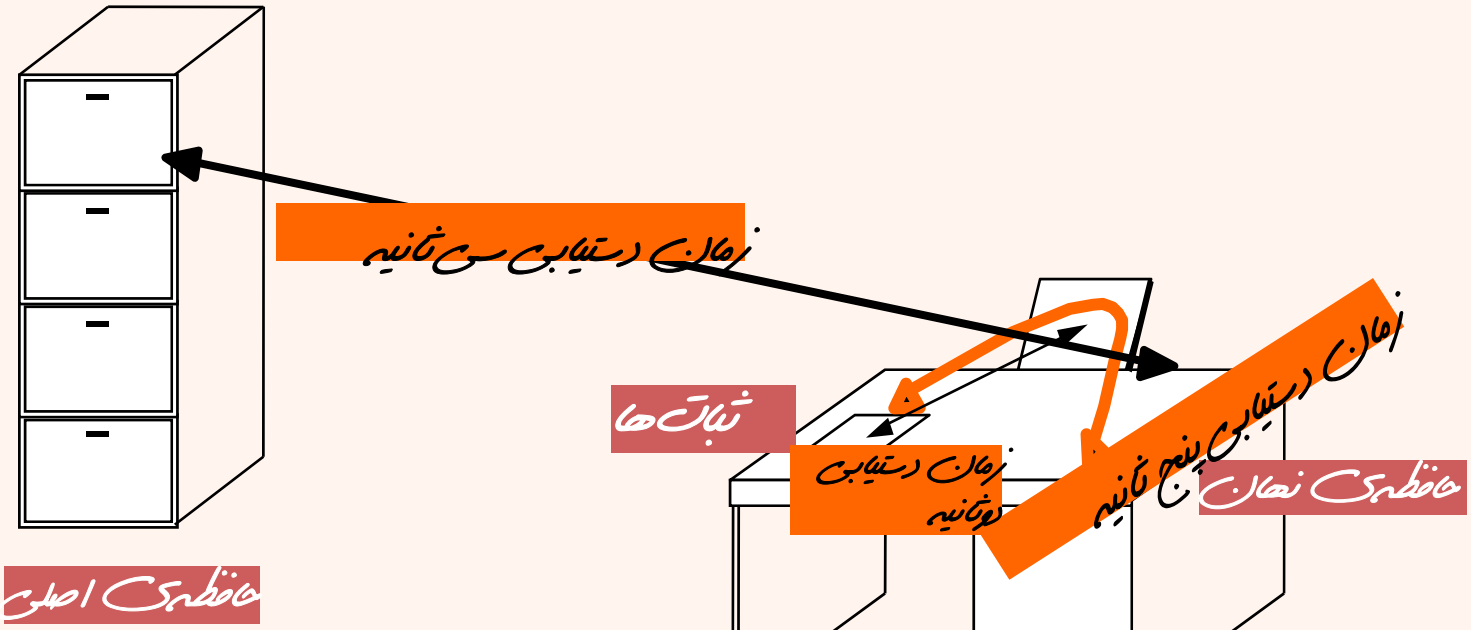
- **همجواری مکانی**: نواحی مجاور بخشی از حافظه که مورد مراجعه قرار گرفته است، با احتمال بالایی مورد استفاده قرار می‌گیرد.
  - آرایه‌ها، و یا توالی دستورالعمل‌های یک برنامه



همجواری (ادامه...)

From Peter Denning's CACM paper, July 2005 (Vol. 48, No. 7, pp. 19-24)

سلسله‌مراتب حافظه

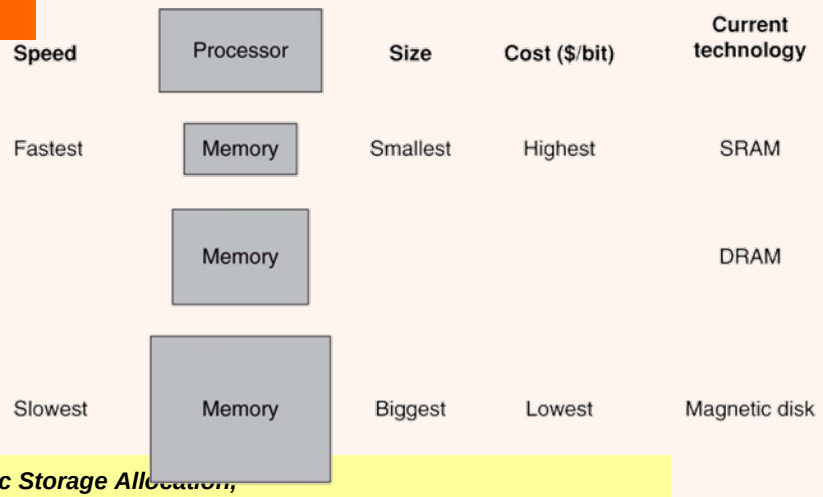


همجواری (ادامه...)

- سلسله مراتب در در حافظه:
  - همه چیز را در دیسک سخت ذخیره کن
  - یک نسخه از موارد نیاز را در DRAM بنویس
  - یک نسخه از موارد مورد نیاز که اخیرا به کار گرفته شده‌اند را در SRAM ذخیره کن.

Main memory

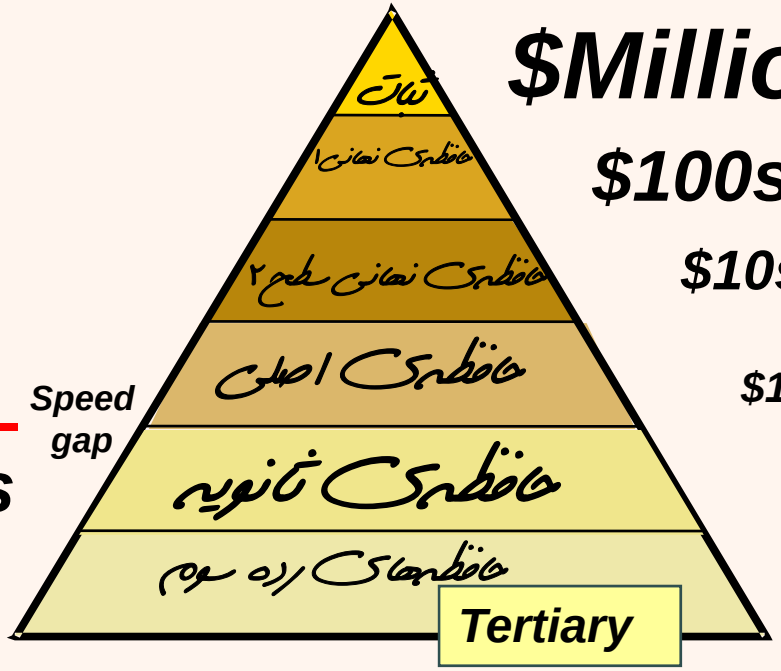
Cache memory attached to CPU



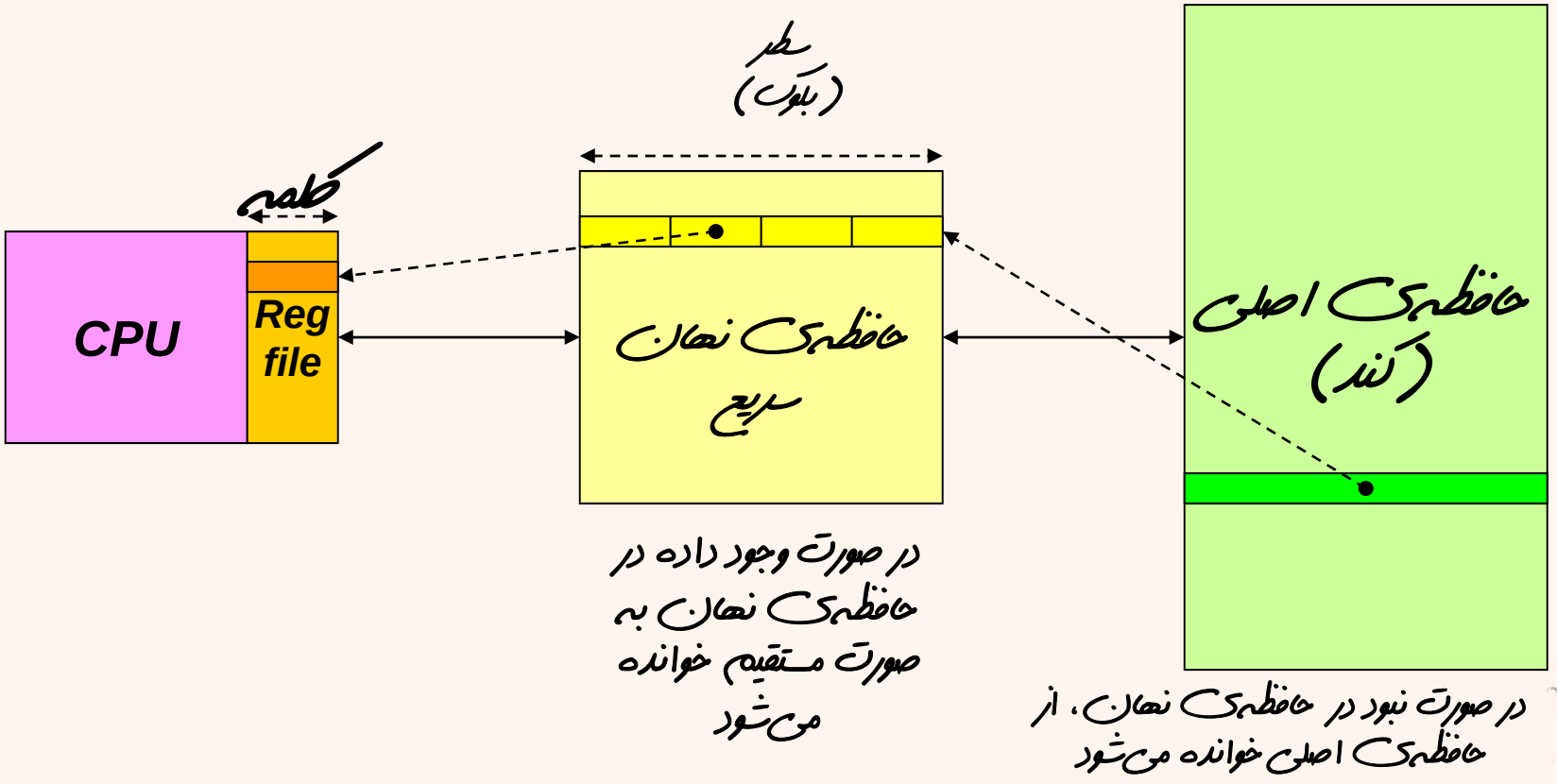
M. V. Wilkes, "Slave Memories and Dynamic Storage Allocation," IEEE Transactions on Electronic Computers, vol. EC-14, no. 2, pp. 270-271, April 1965.

سطوح مختلف حافظه

گنجایش	زمان دستیابی	قیمت هر GB
100s B	ns	$Millions
10s KB	a few ns	$100s Ks
MBs	10s ns	$10s Ks
100s MB	100s ns	$1000s
10s GB	10s ms	$10s
TBs	min+	$1s



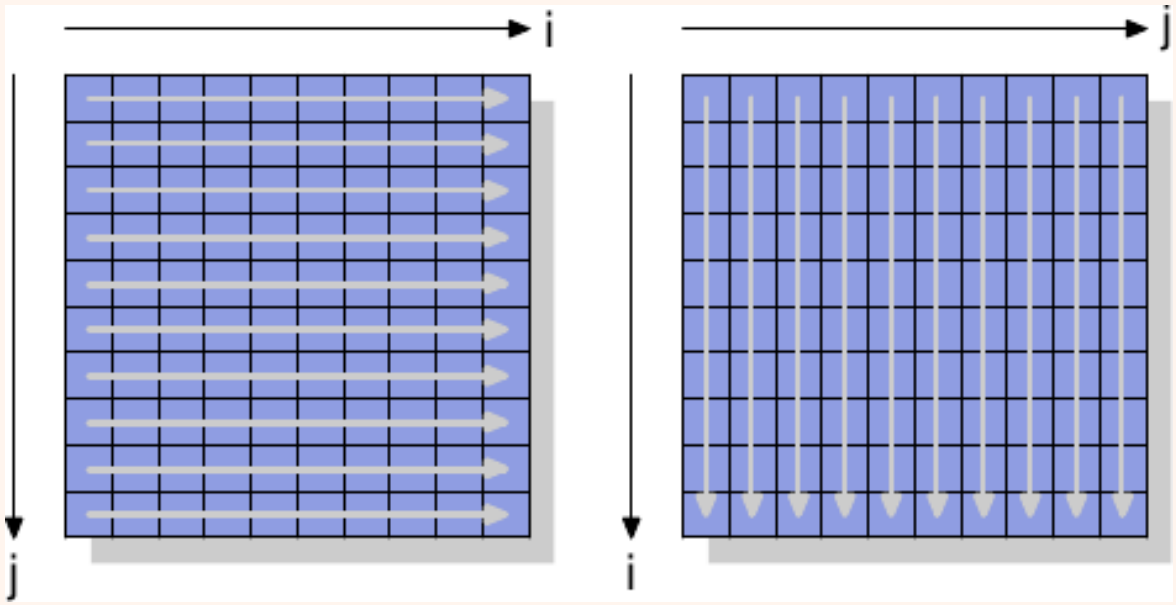
حافظه‌ی نهان (ادامه...)

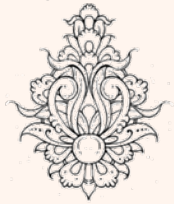


دسترسی به حافظه نهان از دید کاربر پنهان است

الگوی دسترسی به حافظه در ماتریس

در شیوه‌ی نخست، احتمال نبود داده در حافظه‌ی نهان کاهش می‌یابد.





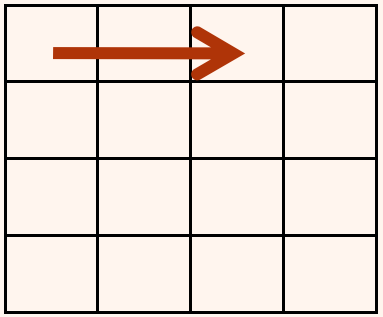
مثال- ضرب ماتریسی

```
for( i=0; i<N; i++ )
    for( j=0; j<N; j++ )
        for( k=0; k<N; k++ )
            c[i][j] += a[i][k] * b[k][j];
```

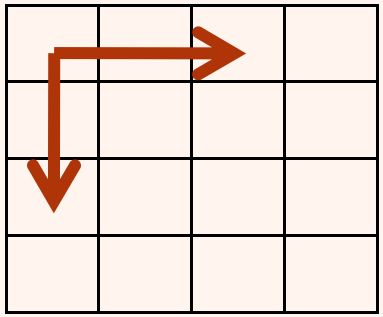
real	0m10.688s
user	0m10.581s
sys	0m0.068s

real	0m5.730s
user	0m5.668s
sys	0m0.052s

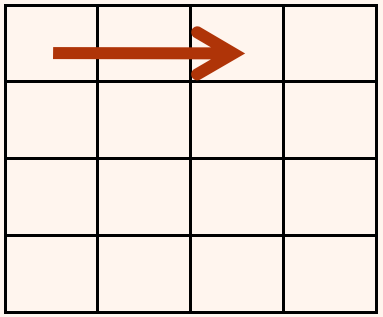
a

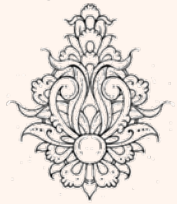


b



c

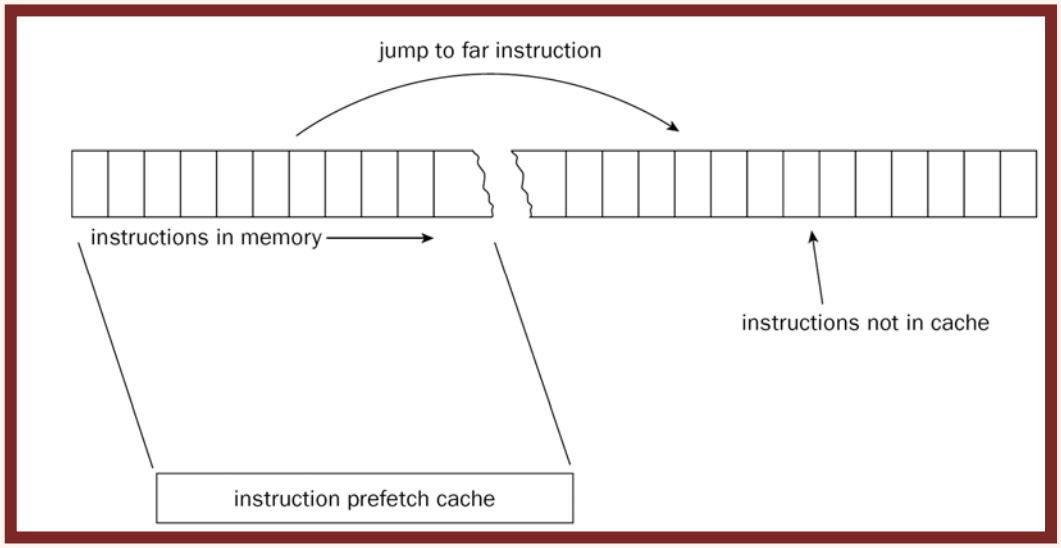




بهینه‌سازی دستورات انشعاب

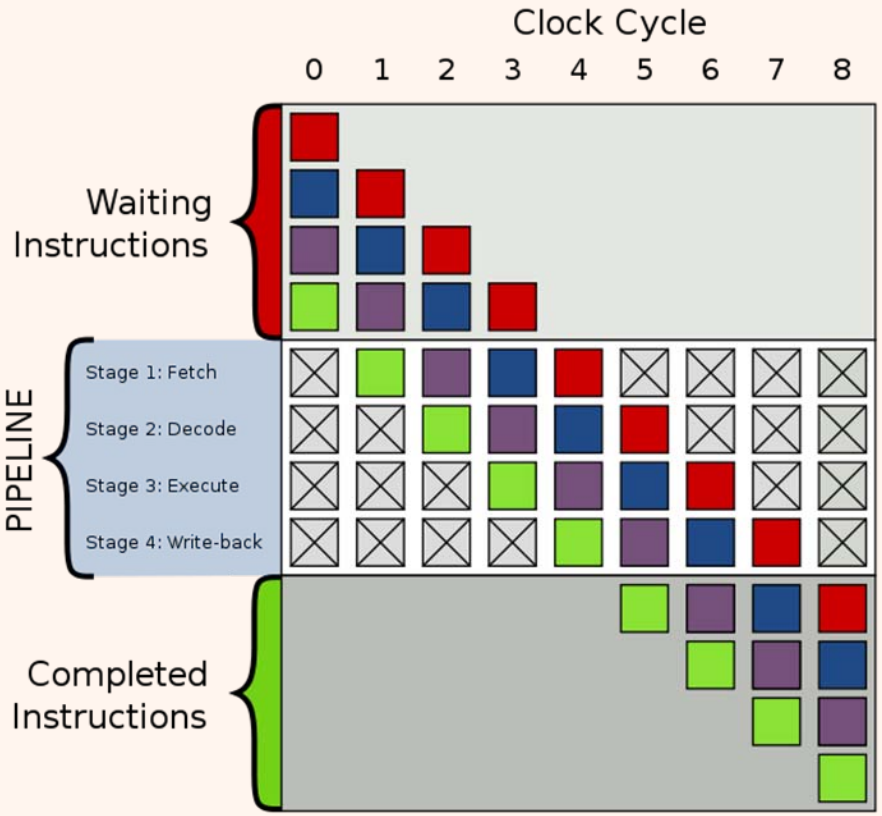
- دستورات انشعاب، به شدت در کارایی سیستم مؤثر هستند.
- در اکثر پردازنده‌های امروزی برای افزایش کارایی دستورات «**پیش‌واکشی**» می‌شود.
- پرش‌های غیرشرطی

– با افزایش فقدان داده در حافظه‌ی نهان باعث کاهش کارایی برنامه می‌شوند.



بهینه‌سازی دستورات انشعاب

- پرش‌های شرطی
  - مانع از عملکرد بهینه‌ی خط لوله می‌شوند.

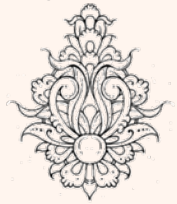


بهینه‌سازی دستورات انشعاب (ادامه...)

- پرش‌های شرطی
  - برای تشخیص توالی دستوراتی که هنوز نتیجه‌ی شرط مشخص نیست، از «الگوریتم‌های پیش‌بینی» استفاده می‌شود.
- پیش‌بینی نتیجه‌ی شرط به صورت‌های زیر انجام می‌شود:
  - در پرش رو به عقب، فرض بر آن است که شرط برقرار است.
  - در پرش رو به جلو، فرض بر آنست که شرط برقرار نیست.
  - دستورات پرشی که در اجرای قبلی، انجام شده‌اند، باز هم اجرا خواهند شد.
    - این فرض بر دو فرض قبل غلبه دارد.
    - BTB، برای ره‌گیری آخرین نتیجه مورد استفاده قرار می‌گیرد.

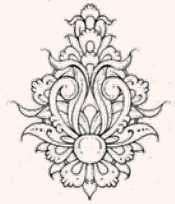
```
movl $100, $ecx
loop1:
addl %cx, %eax
decl %ecx
jns loop1
```

Branch Target Buffer



بهینه‌سازی دستورات انشعاب

- تا جایی که امکان دارد، از دستورات پرش استفاده نکنید!
  - «دستورات جابجایی شرطی» به منظور بهبود کارایی مطرح شده‌اند، به زودی با آن‌ها آشنا خواهیم شد.
  - گاهی اوقات با افزودن چند خط برنامه، می‌توان دستورات پرش کم‌تری استفاده کرد.



بهینه‌سازی دستورات انشعاب

```
loop:
    cmp data(, %edi, 4), %eax
    je part2
    call function1
    jmp looptest
part2:
     call function2
looptest:
    inc %edi
    cmpl $10, %edi
    jnz loop
```

```
loop:
    cmp data(, %edi, 4), %eax
    je part2
    call function1
    inc %edi
    cmp $10, %edi
    jnz loop
    jmp end
part2:
    call function2
    inc %edi
    cmp $10, %edi
    jnz loop
end:
```

بهینه‌سازی دستورات انشعاب

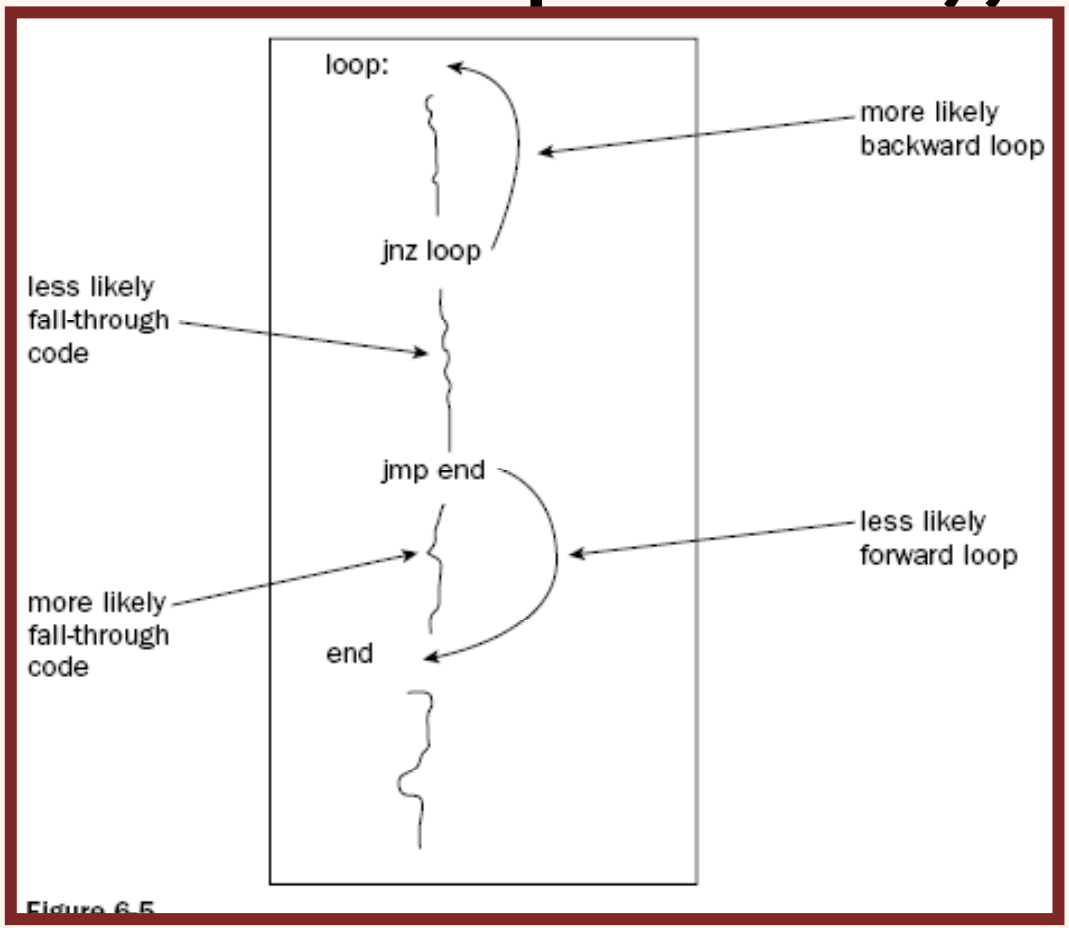


Figure 6.5

- برنامه به گونه‌ای نوشته شود که نتایج پیش‌بینی تقویت شود.

–به عنوان مثال در دستور if بخشی که با احتمال بیشتری اجرا می‌شود، در قسمت then قرار گیرد.

بهینه‌سازی دستورات انشعاب

```
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:
```

THEN

ELSE

هنگامی که کد کامپایل می‌شود، سعی بر این است که به گونه‌ای کامپایل شود، که کارایی بیشتری داشته باشد.

به این علت در پیاده‌سازی به جای jle از jg استفاده شده است.

```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;
}
```

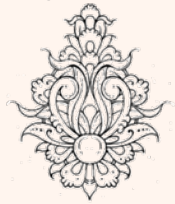
قسمتی که با احتمال بیشتر باید اجرا شود در قسمت then قرار گیرد

باز کردن حلقه

- برای حلقه‌ها از پرش‌های رو به عقب استفاده می‌شود، ولی با این وجود باز هم در کارایی ایجاد خلل می‌کند.
- برای کدهای کوچک، می‌توان حلقه‌ها را باز کرد:

```
movl values, %ebx
movl $0, %edi
loop:
movl values(, %edi, 4), %eax
cmp %ebx, %eax
cmova %eax, %ebx
inc %edi
cmp $4, %edi
jne loop
```

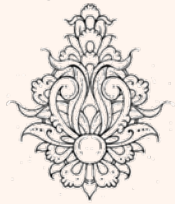
```
movl values, %ebx
movl $values, %ecx
movl (%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx

movl 4(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx

movl 8(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx

movl 12(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx
```



در مورد دستور cmova به زودی صحبت خواهد شد

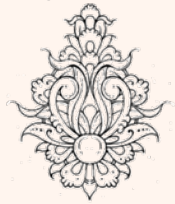
باز کردن حلقه (ادامه...)

- با این کار، به سرعت اجرای برنامه افزوده می‌شود، البته هزینه‌ی آن هم افزایش حجم فایل برنامه است.
- این کار را می‌توان هم به صورت دستی انجام داد، هم بر عهده‌ی بهینه‌ساز کامپایلر گذاشت.



معایب باز کردن حلقه

- این کار حجم برنامه را افزایش می‌دهد، برای سیستم‌های درون‌کار چنین چیزی مطلوب نیست.
  - افزایش حجم برنامه، به طور کلی منجر به افزایش فقدان داده در حافظه‌ی نهان می‌شود، که این خود ممکن است باعث کندی برنامه می‌شود.
- در صورتی که این کار بدون بهره‌گیری از کامپایلر انجام شود، خوانایی کند را کاهش می‌دهد.
- در صورتی که بدنه‌ی حلقه شامل تابع inline باشد، ممکن است چنین کاری عملی نباشد، چرا که حجم کد به شدت افزایش خواهد یافت.

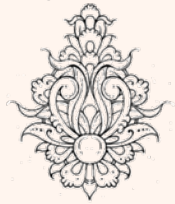


مثال

```c
#include <stdio.h>
int main() {
        int i;
        for(i=0;i<3;i++){
            printf("%d",i);
    }
}
```

```
ses/Asm/92_2/chap09$ gcc -O2  --save-temps unroll.c
```

```
movl    $1, (%esp)
call    __printf_chk
movl    $1, 8(%esp)
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk
movl    $2, 8(%esp)
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk
leave
```



مثال

```c
#include <stdio.h>
int main() {
        int i;
        for(i=0;i<5;i++){
            printf("%d",i);
    }
}
```

```
gcc -O2  --save-temps unroll.c
```

```
.L2:
        movl    %ebx, 8(%esp)
        addl    $1, %ebx
        movl    $.LC0, 4(%esp)
        movl    $1, (%esp)
        call    __printf_chk
        cmpl    $5, %ebx
        jne     .L2
```

```
gcc -O2 -funroll-all-loops --save-temps unroll.c
```

```
call    __printf_chk
movl    $1, 8(%esp)
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk
movl    $2, 8(%esp)
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
call    __printf_chk
movl    $3, 8(%esp)
movl    $.LC0, 4(%esp)
movl    $1, (%esp)
```

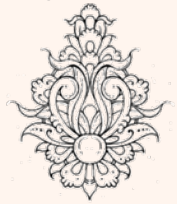
دستورات جابجایی شرطی

- دستور mov یکی از پراستفاده‌ترین دستورات است.
- تکه کد زیر به چه منظوری نوشته شده است؟

پرسش

```
addb $1, %al
jnc continue
movb $255, %al
continue:
```

- با استفاده از دستورات پرش شرطی می‌توان در ساختار فوق را ساده‌تر نوشت.
- هر چند مثال فوق یک مثال ساده است، اما به طور کلی استفاده از دستورات جابجایی شرطی میزان استفاده از دستورات jmp را کاهش می‌دهد که موجب افزایش سرعت برنامه‌ها خواهد شد.



دستورات جابجایی شرطی

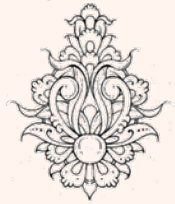
یک یا دو حرف است که شرط انتقال را نشان می‌دهد.

- قالب این دستورات به صورت زیر می‌باشد:

```
cmovc   source, destination
```

EFLAGS Bit	Name	Description
CF	Carry flag	A mathematical expression has created a carry or borrow
OF	Overflow flag	An integer value is either too large or too small
PF	Parity flag	The register contains corrupt data from a mathematical operation
SF	Sign flag	Indicates whether the result is negative or positive
ZF	Zero flag	The result of the mathematical operation is zero

پرچم هایی که در دستورات جابجایی شرطی مورد استفاده قرار می‌گیرند



جابجایی اعداد بدون علامت

دستورات جابجایی شرطی

Instruction Pair	Description	EFLAGS Condition
CMOVA/CMOVNBE	Above/not below or equal	(CF or ZF) = 0
CMOVAE/CMOVNB	Above or equal/not below	CF=0
CMOVNC	Not carry	CF=0
CMOVB/CMOVNAE	Below/not above or equal	CF=1
CMOVC	Carry	CF=1
CMOVBE/CMOVNA	Below or equal/not above	(CF or ZF) = 1
CMOVE/CMOVZ	Equal/zero	ZF=1
CMOVNE/CMOVNZ	Not equal/not zero	ZF=0
CMOVP/CMOVPE	Parity/parity even	PF=1
CMOVNP/CMOVPO	Not parity/parity odd	PF=0

جابجایی اعداد علامت‌دار

دستورات جابجایی شرطی

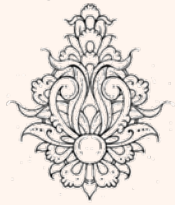
Instruction Pair	Description	EFLAGS Condition
CMOVGE/CMOVNL	Greater or equal/not less	(SF xor OF)=0
CMOVL/CMOVNGE	Less/not greater or equal	(SF xor OF)=1
CMOVLE/CMOVNG	Less or equal/not greater	((SF xor OF) or ZF)=1
CMOVO	Overflow	OF=1
CMOVNO	Not overflow	OF=0
CMOVS	Sign (negative)	SF=1
CMOVNS	Not sign (non-negative)	SF=0

مثال

```
.section .data
output:
.asciz “The largest value is %d\n”
values:
.int 105, 235, 61, 315, 134, 221,
     53, 145, 117, 5
.section .text
.globl _start
_start:
nop
movl values, %ebx
movl $1, %edi
```

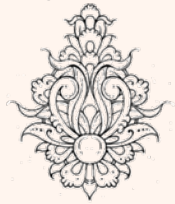
```
loop:
movl values(, %edi, 4), %eax
cmp %ebx, %eax
cmova %eax, %ebx
inc %edi
cmp $10, %edi
jne loop
pushl %ebx
pushl $output
call printf
addl $8, %esp
pushl $0
call exit
```

```
ahmad@ubuntu:~/Courses/Assembly/chapter3$ ./cmovtest
The largest value is 315
```



بهینه‌سازی با هدف کاهش حجم برنامه

- همیشه بهینه‌سازی با این هدف صورت نمی‌گیرد که سرعت اجرا افزایش یابد، در برخی مواقع هدف کاهش حجم بخش‌هایی از کد است، حتی به بهای کاهش سرعت!
- اما همیشه هم این‌طور نیست!
- حافظه‌ی نهان دستورالعمل حجم محدودی از کد را پذیرا خواهد بود. (مثلا ۸KB تا ۳۲KB) در صورتی که حجم کد افزایش یابد، cache miss افزایش خواهد یافت و در نتیجه با کاهش سرعت مواجه خواهیم شد.



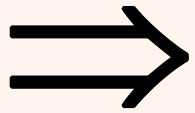
بهینه‌سازی با هدف کاهش حجم برنامه (ادامه...)

- به عنوان مثال در خانواده‌ی core2 بلوک‌های حافظه‌ی نهان ۶۴ بایتی هستند، از این رو در صورتی که حجم بدنه‌ی حلقه کم‌تر از این میزان باشد، در افزایش کارایی تاثیرگذار خواهد بود.
- از این جهت برای برخی دستورها که شیوه‌های متعددی وجود دارد، مطلوبست سراغ دستوری رویم که حجم کم‌تری داشته باشد.

پیش از این در مثال‌ها نمونه‌ای از چنین رویکردی را دیدیم، آیا آن را به خاطر می‌آورید؟

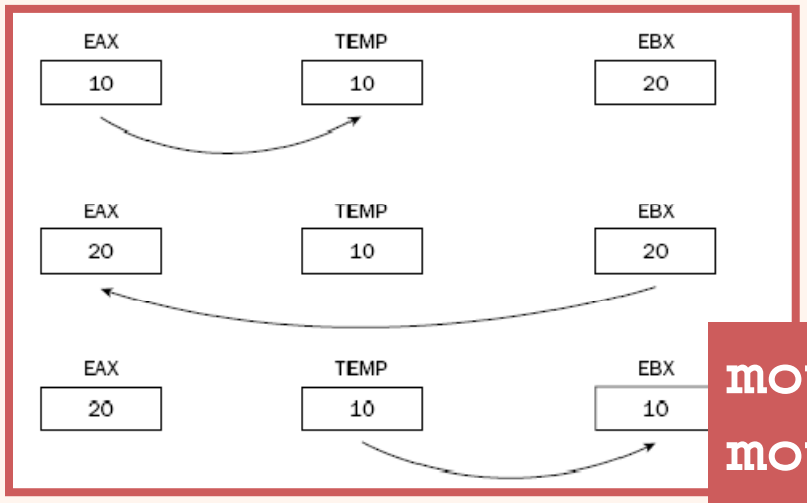
Loop splitting

```
for I = exp1 to exp2
    A(I)
    B(I)
```



```
for Ia = exp1 to exp2
    A(Ia)

for Ib = exp1 to exp2
    B(Ib)
```

تعویض محتوای دو متغیر




```
movl %eax, %ecx
movl %ebx, %eax
movl %ecx, %ebx
```

- گاهی لازم است محتوای دو ثبات را جابجا کنیم.
- با استفاده از دستور mov ناچاریم یک متغیر موقت بگیریم و این کار را با استفاده از سه دستور انجام دهیم.
- لابد حدس زده‌اید قضیه چیست؟!

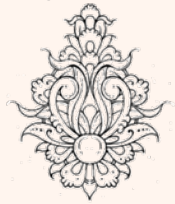
تعویض محتوای دو متغیر (ادامه...)

```
xchg operand1, operand2
```

- دستور xchg محتوای دو متغیر را جابجا می‌کند.
- عملوندهای این دستور هم می‌توانند ثبات همه‌منظوره باشند و هم حافظه، با این شرط که هر دو نمی‌توانند حافظه باشند.

```
movl %eax, %ecx
movl %ebx, %eax
movl %ecx, %ebx
```

```
xchg %ebx, %eax
```



بهینه‌سازی با هدف کاهش حجم برنامه (ادامه...)

پیش از لینک کردن

```
# example20.s
.section .data
values:
    .int 100, 101
.section .text
.globl _start
_start:
    nop
    movl values, %eax
    movl values+4, %ebx
    movl %eax, %ecx
    movl %ebx, %eax
    movl %ecx, %ebx
    xchg %eax,%ebx
    xchg %eax,values
    pushl $0
    call exit
```

```
00000000 <_start>:
   0:   90                      nop
   1:   a1 00 00 00 00          mov    0x0,%eax
   6:   8b 1d 04 00 00 00       mov    0x4,%ebx
   c:   89 c1                   mov    %eax,%ecx
   e:   89 d8                   mov    %ebx,%eax
  10:   89 cb                   mov    %ecx,%ebx
  12:   93                      xchg   %eax,%ebx
  13:   87 05 00 00 00 00       xchg   %eax,0x0
  19:   6a 00                   push   $0x0
  1b:   e8 fc ff ff ff          call   1c <_start+0x1c>
```

```
(gdb) disas _start+1
Dump of assembler code for function _start:
   0x08048184 <+0>:     nop
   0x08048185 <+1>:     mov    0x8049254,%eax
   0x0804818a <+6>:     mov    0x8049258,%ebx
   0x08048190 <+12>:    mov    %eax,%ecx
   0x08048192 <+14>:    mov    %ebx,%eax
   0x08048194 <+16>:    mov    %ecx,%ebx
   0x08048196 <+18>:    xchg   %eax,%ebx
   0x08048197 <+19>:    xchg   %eax,0x8049254
   0x0804819d <+25>:    push   $0x0
   0x0804819f <+27>:    call   0x8048174 <exit@plt>
```

پس از لینک کردن

یک نکته

```
# example20.s
.section .data
values:
    .int 100, 101
.section .text
.globl _start
_start:
    nop
    xchg %eax,%eax
    xchg %ebx,%ebx
    xchg %ecx,%ecx
```

```
00000000 <_start>:
   0:   90                      nop
   1:   90                      nop
   2:   87 db                   xchg   %ebx,%ebx
   4:   87 c9                   xchg   %ecx,%ecx
```