زبان ماشین و اسمبلی
(۱۳-۱۱-۰۰۵)
بخش دوم

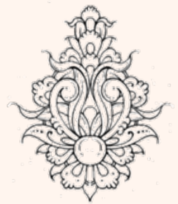
آشنایی با MIPS

دانشگاه شهید بهشتی
دانشکده‌ی مهندسی برق و کامپیوتر
زمستان ۱۳۹۲
احمد محمودی ازناوه

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

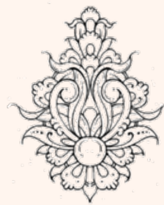
فهرست مطالب

- آشنایی با دستورالعمل‌ها
- انواع دستورالعمل
- کد ماشین
- دسترسی به حافظه
- عملوندهای بلاواسطه
  - قالب دستور
- دستورهای منطقی



فهرست مطالب (ادامه ...)

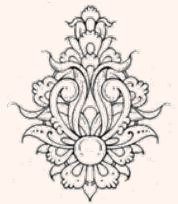
- دستورهای شرطی
- آدرس‌دهی شبه‌مستقیم
- سایر دستورهای شرطی
- اعداد علامت‌دار و بدون علامت
- ضرب و تقسیم
- ثابت‌های سی‌دوبیتی
- فراخوانی روال
- آشنایی با مفهوم پشته
- ساختار برنامه
- جمع‌بندی



مجموعه‌ی دستورها

- کامپیوترهای مختلف، مجموعه دستورات متفاوتی دارند.
  - با وجود تفاوت، در بسیاری وجوه مشابه هستند.
- نخستین کامپیوترها دارای مجموعه‌ی دستورات ساده‌ای بودند. در برخی از سیستم‌های امروزی نیز حفظ این سادگی ترجیح داده می‌شود.
- در این بخش با مجموعه‌ی دستورات MIPS آشنا خواهیم شد.

Microprocessor without Interlocked Pipeline Stages



دستورهای حسابی

- جمع (دو منبع و یک حاصل)

عملوند مقصد (حاصل)
Destination operand

```
add a, b, c   # a gets b + c
```

عملگر
operator

عملوندهای منبع
Source operands

توضیح
Comment

destination ← source1 op source2

برخلاف سایر زبان‌ها، در زبان اسمبلی هر دستور در یک سطر نوشته می‌شود.

دستورات زبان اسمبلی باید به گونه‌ای باشند که توسط ماشین قابل اجرا باشند. سخت‌افزار طراحی شده قادر به انجام دستورها باشد.

مثال

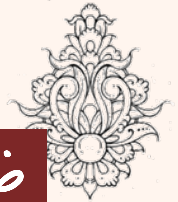
- C code:

```
f = (g + h) - (i + j);
```

Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f, t0, t1   # f = t0 - t1
```

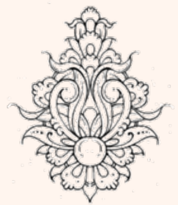
کامپایلر ممکن است به فضاهای موقت احتیاج داشته باشد



ضمنا در مورد نوع عملوندها هم ممکن است محدودیت‌هایی وجود داشته باشد. همان گونه که سخت‌افزار هر دستوری را نمی‌تواند اجرا کند، یک عملیات خاص روی هر عملوندی هم قابل انجام نخواهد بود.

ثبات‌ها در نقش عملوند

- در پردازنده‌ی MIPS برای دستورات حسابی، محدود به استفاده از **ثبات‌ها** هستیم.
- MIPS دارای ۳۲ ثبات ۳۲ بیتی است.
  - که با شماره‌های 0 تا 31 مشخص می‌شوند.
  - داده‌هایی که بناست در پردازش‌گر مورد استفاده قرار گیرند، به این ثبات‌ها منتقل می‌شوند.
  - هر ۳۲ بیت را یک کلمه (Word) می‌نامند.
- در زبان اسمبلی ثبات‌ها به صورت زیر نام‌گذاری می‌شوند.
  - $s0, $s1, ..., $s7
  - $t0, $t1, ..., $t9 ثبات‌های موقت که در کامپایل مورد استفاده قرار می‌گیرند.



ثبات‌ها در نقش عملوند (ادامه...)

- C code: `f = (g + h) - (i + j);`

  f, …, j in $s0, …, $s4

- Compiled MIPS code:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

یکی از خصوصیات MIPS32 این است که همه‌ی دستورالعمل‌های آن سی‌ودوبیتی هستند. در واقع همه‌ی دستورهای بالا به یک رشته‌ی چهاربایتی تبدیل می‌شوند.

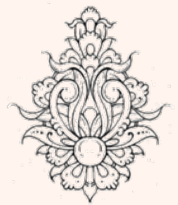
پرسش

به نظر شما محدودیتی در مورد تعداد دستورالعمل‌ها وجود دارد؟

قالب دستورهای MIPS

op	rs	rt	rd	sa	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- فیلدهای دستور:
  - op: دستور را مشخص می‌کند.
  - rs: ثبات منبع اول
  - rt: دومین ثبات منبع
  - rd: ثبات مقصد
  - shamt: میزان شیفت
  - funct: نوع خاصی از دستور
- با توجه به این که سی‌دو ثبات داریم، با پنج بیت می‌توان هر ثبات را مشخص کرد.



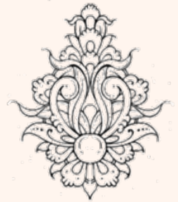
نمایش دستورات

machine code

- همان‌طور که داده‌ها با اعداد دودویی نمایش داده می‌شوند، دستورها هم با اعداد دودویی نشان داده می‌شوند.
- دستورات MIPS:
  - سی و دو بیتی هستند
  - بخشی از این سی و دو بیت به عملیات بستگی دارد.

opcode (operation code)

- به هر ثبات عددی نسبت داده می‌شود.
  - شماره‌ی ۸ تا ۱۵ برای $t0 – $t7
  - شماره‌های ۲۴ و ۲۵ برای $t8 – $t9
  - اعداد ۱۶ تا ۲۳ برای $s0 – $s7



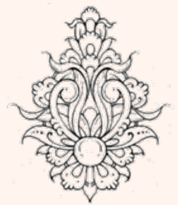
مثال

op	rs	rt	rd	sa	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

```
add $t0, $s1, $s2
```

Special code	$s1	$s2	$t0	0	add
0	**17**	**18**	**8**	**0**	**32**
000000	**10001**	**10010**	**01000**	**00000**	**100000**

$$00000010001100100100000000100000_2 = 02324020_{16}$$



پرسش

یک بار دیگر به این پرسش پاسخ دهید:

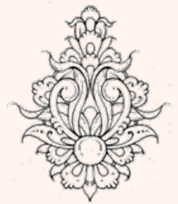
به نظر شما محدودیتی در مورد تعداد دستورالعمل ها وجود دارد؟

مثال

`sub $t0, $s1, $s2`

destination ← source1 op source2

0	17	18	8	0	34



حافظه به عنوان عملوند

پرسش

- معماری MIPS32 سی‌ودو خط آدرس دارد.

حداکثر حافظه‌ای که می‌تواند آدرس‌دهی کند، چقدر است؟

- هر خانه‌ی حافظه به یک بایت اشاره می‌کند.
- در MIPS، خانه‌ی حافظه با آدرس کوچک‌تر، حاوی بایت پرارزش‌تر است.

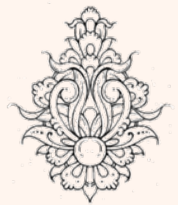
big-endian

بخش پرارزش‌تر در خانه‌ی اول قرار می‌گیرد

- کلمات در حافظه «هم‌تراز» شده‌اند، شروع هر کلمه مضربی از چهار است.

alignment restriction

  - جزییات بیشتر در این مورد را به آینده موکول می‌کنیم.



حافظه به عنوان عملوند

- ساختارهای پیچیده‌تر در حافظه ذخیره می‌شوند.
  - مانند آرایه‌ها، ساختارها و ...
- به ناچار باید اعمال حسابی روی این ساختارها هم صورت پذیرد.

پرسش

برای دسترسی مستقیم به «آدرس یک خانه‌ی حافظه» به چند بیت احتیاج داریم؟

آیا دسترسی مستقیم به حافظه در MIPS32 امکان‌پذیر است؟

چه راه حلی پیشنهاد می‌کنید؟

op	rs	rt	rd	sa	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

نشانی‌دهی غیرمستقیم ثباتی

- با توجه به این که دستورالعمل‌ها سی‌ودو بیتی هستند، در صورتی که بخواهیم به آدرس یک خانه از حافظه را به صورت مستقیم مورد استفاده قرار دهیم جایی برای کددستورالعمل باقی نمی‌ماند.
- در این حالت می‌توان به سراغ نشانی‌دهی غیرمستقیم از طریق ثبات رفت، به این معنا که ابتدا آدرس در یک ثبات قرار بگیرد و به جای آدرس حافظه، شماره‌ی ثبات مورد استفاده قرار گیرد.

دستورهای خواندن و نوشتن در حافظه

destination

```
lw    $t0, ($s3)   #load word from memory
sw    $t0, ($s3)   #store word to memory
```

source

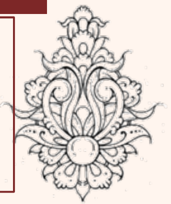
base register

($s3)

جایی که $s3 به آن اشاره می‌کند.

offset

```
lw    $t0, 8($s3)   #load word from memory
sw    $t0, 12($s1)   #store word to memory
```



12($s3)

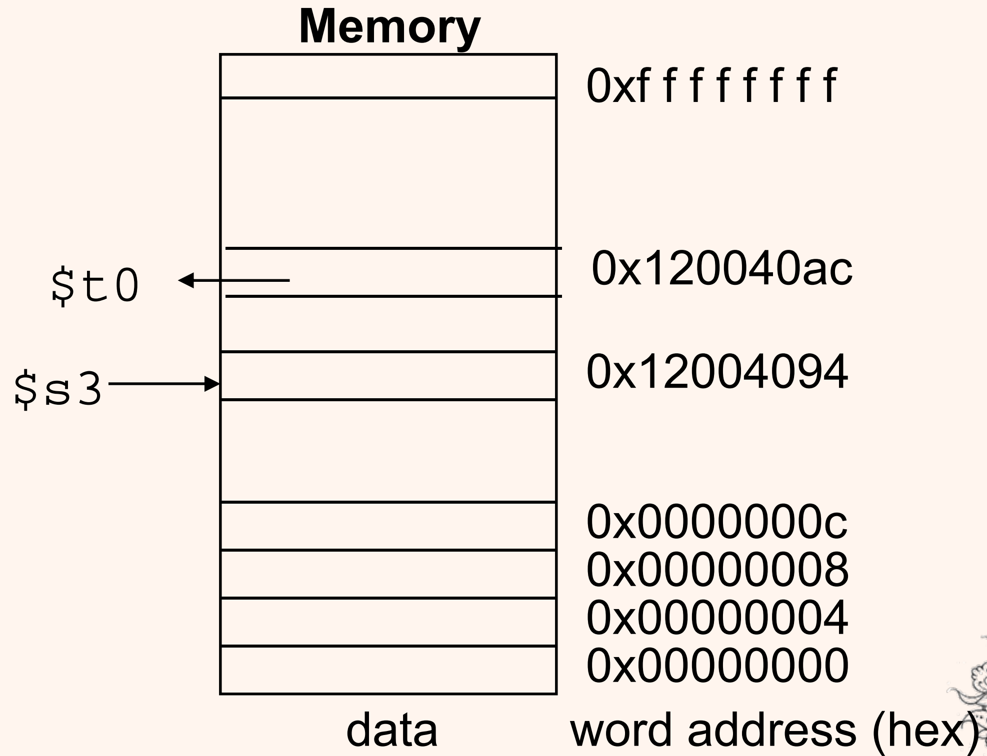
جایی که $s3+12 به آن اشاره می‌کند.

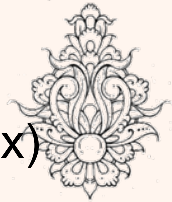
مثال

```
lw $t0, 24($s3)
```

24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac





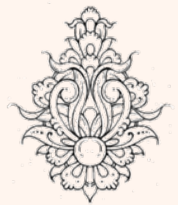
حافظه به عنوان عملوند (ادامه...)

برای اعمال دستورهای حسابی
داده از حافظه به ثبات منتقل شده و پس از انجام محاسبات،
حاصل در حافظه نوشته می‌شود.

- زبان C:

```
g = h + A[8];
```

h در $s2، آدرس پایه‌ی A در $s3 این آرایه شامل داده‌های سی‌ودو بیتی است و نهایتا g در $s1 ذخیره خواهد شد.



- کد کمپایل شده:

```
lw  $t0, 32($s3)     # load wrd
add $s1, $s2, $t0
```

حافظه به عنوان عملوند (ادامه...)

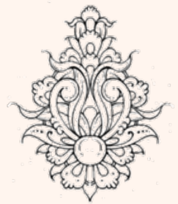
- زبان C:

```
A[12] = h + A[8];
```

h در $s2 و آدرس پایه‌ی A در $s3

- کد کمپایل شده:

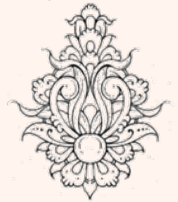
```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```



ثبات و حافظه

- دستیابی به محتوای ثبات‌ها بسیار سریع‌تر از محتوای حافظه می‌باشد.
- برای هر بار دستیابی به حافظه، اجرای دستورات lw و sw لازم است. یعنی تعداد دستورات بیشتر است.
- کامپایلر باید تا جایی که ممکن است از رجیسترها به عنوان متغیر استفاده کنند.
  - در صورت در اختیار نداشتن ثبات، از بین متغیرها، آن‌هایی که کم‌تر مورد استفاده قرار می‌گیرند، از ثبات خارج می‌شوند.
  - استفاده بهینه از فضای ثبات‌ها مهم است.

spilling register



استفاده از اعداد ثابت

- یکی از ثابت‌های پراستفاده، صفر است.
- رجیستر $zero، حاوی ثابت 0 است.
  - این ثبات قابل تغییر نیست! برای خیلی کاربردها مفید است، به عنوان مثال برای انتقال بین رجیسترها

```
add $t2, $s1, $zero
```

پرسش

چگونه می‌توان عدد ثابتی را به یک ثبات اضافه کرد؟

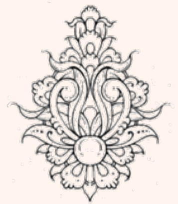
استفاده از اعداد ثابت

- در بسیاری موارد لازم است، از اعداد ثابت در برنامه‌ها استفاده کرد. چه راهی پیشنهاد می‌دهید؟
  - به عنوان مثال در صورتی که بخواهیم به $s3 چهار واحد اضافه کنیم؟

```
lw  $t0,AddrConstant4($s1) # $t0= constant 4
add $s3,$s3,$t0
```

  - با توجه به استفاده مکرر از چنین دستوراتی دستور جدیدی پیشنهاد می‌شود:

```
addi $s3, $s3, 4
```



استفاده از اعداد ثابت (ادامه...)

- مثال: برای این که یک واحد از ثبات $s1 کم کنیم، چه پیشنهادی دارید؟

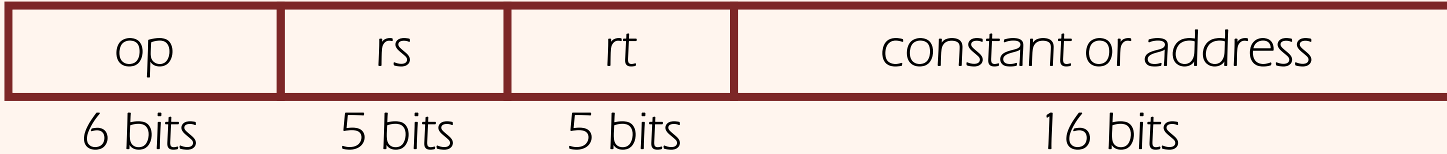
```
addi $s2, $s1, -1
```

- بدین ترتیب با استفاده از این دستورات، تعداد دستورات برنامه کاهش می‌یابد.

دستورالعمل‌های خواندن و نوشتن و دستورات کار با اعداد ثابت چگونه به کد قابل فهم برای ماشین تبدیل خواهند شد؟

دستورهای با عملوند ثابت

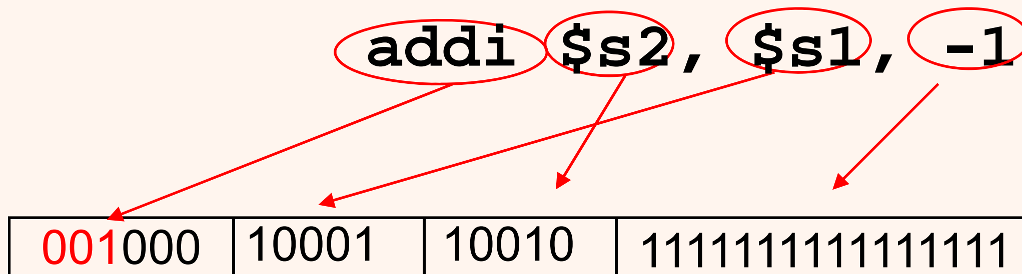
MIPS I-format Instructions



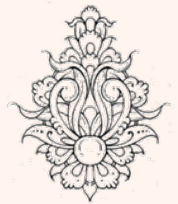
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- برای دستورات lw، sw و دستوراتی که نیاز به استفاده از ثابت‌ها دارند، قالب دیگری مطرح می‌شود.
  - rt: (بسته به دستور)ثبات منبع یا مقصد
  - rs: (بسته به دستور)ثبات منبع یا ثبات حاوی آدرس پایه
  - بدین ترتیب می‌توان ثابتی از -2^{15} تا $+2^{15}-1$ را در این گونه دستورها به کار برد.
  - همچنین، برای دستوراتی که با آدرس حافظه کار می‌کنند، بخش آخر دربردارنده‌ی آدرس می‌باشد.

مثال



0x2232ffff



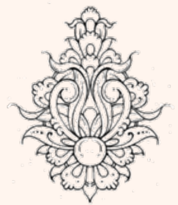
دستورهای با عملوند ثابت(ادامه...)

```
addi $s2, $s1, -1
```

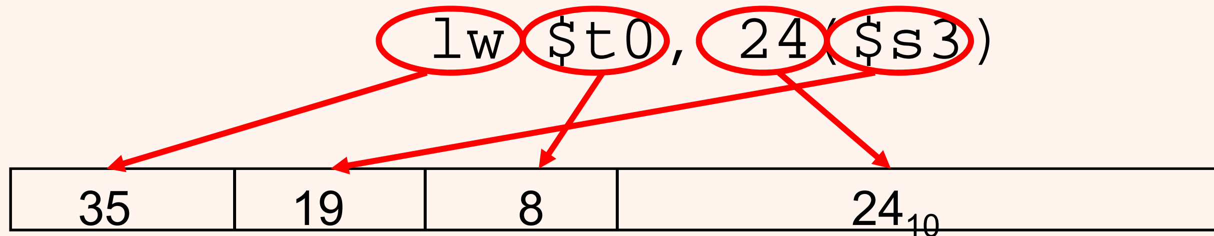
001000	10001	10010	1111111111111111

0x2232ffff

- مقدار بلاواسطه (immediate) در خود دستور جای گرفته است!
- محدودیت بیت‌های اختصاص داده شده باعث محدودیت بازه‌ی اعداد مورد استفاده خواهد شد.
- واحد کنترل پیش از رمزگشایی دستور باید قالب آن را تشخیص دهد.
  - سه بیت آخر، اگر ۰۰۱ باشد، به معنای دستورات محاسباتی با داده‌ی بلاواسطه است.

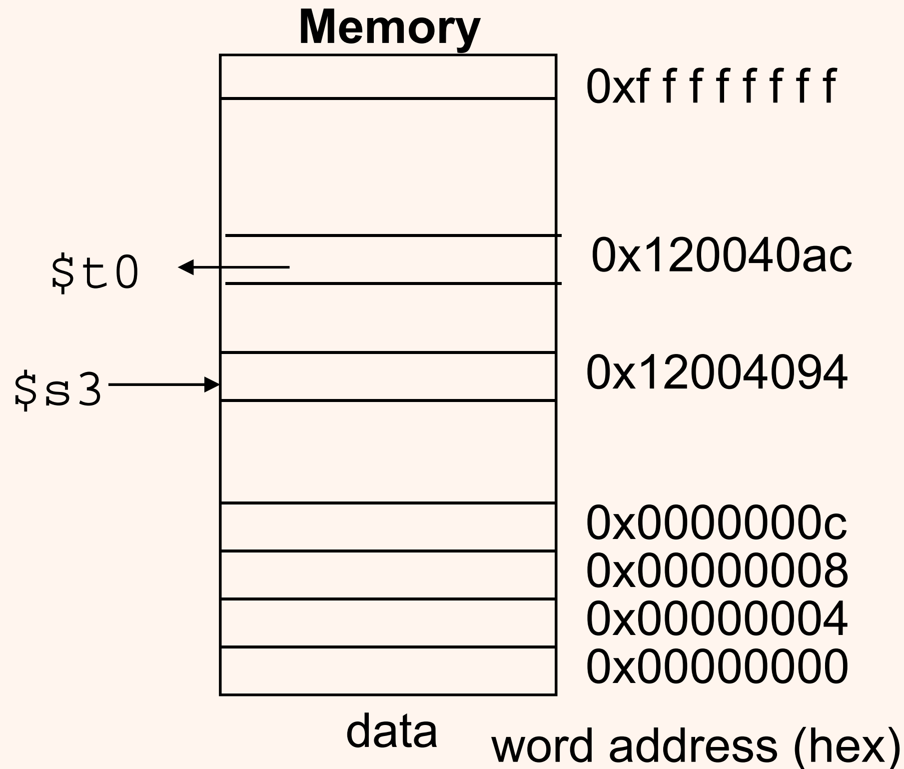


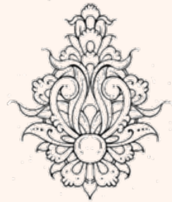
مثال(تکرار با جزییات بیشتر)



24_{10} + $s3 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac

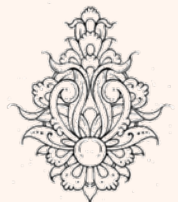




دستورات منطقی

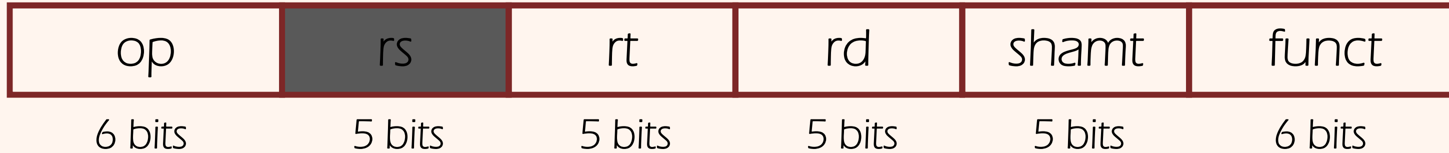
- دستورات منطقی برای دست‌کاری بیت‌ها مورد استفاده قرار می‌گیرند.
- با کمک آن‌ها می‌توان بیت‌های خاصی را انتخاب و مقدار آن‌ها را تغییر داد.

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right logical	>>	>>>	`srl`
Shift right arith.	>>	>>	`sra`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`



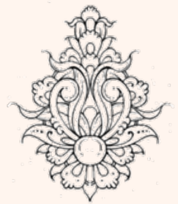
دستور شیفت

MIPS R-format Instructions



Shift amount

- قالب این دستور از نوع R است.
- shamt: میزان شیفت
- هنگام شیفت به چپ(راست)، کم(پر)ارزش‌ترین بیت با '0' پر می‌شود.
- شیفت به چپ، معادل ضرب در دو است.
- شیفت به راست برای اعداد بدون‌علامت معادل تقسیم بر دو است.



دستور شیفت (ادامه...)

```
sll $t2, $s0, 8      #$t2 = $s0 << 8 bits
```

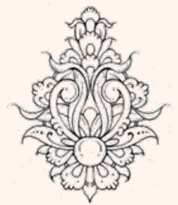
000000	00000	10000	01010	01000	000000

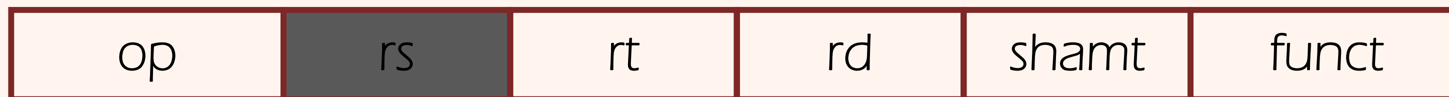
```
srl $t2, $s0, 8      #$t2 = $s0 >> 8 bits
```

000000	00000	10000	01010	01000	000010

```
sra $t2, $s0, 8      #$t2 = $s0 >>> 8 bits
```

000000	00000	10000	01010	01000	000011





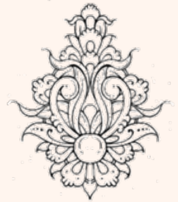
6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

عملگر and

- با استفاده از and به عنوان ماسک می‌توان بخشی از یک کلمه را استخراج کرد.

```
and $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0000 1100 0000 0000

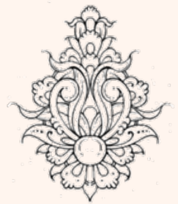


عملگر or

- با استفاده از or می‌توان برای مقداردهی بیت‌های خاص اقدام کرد.

```
or $t0, $t1, $t2
```

$t2	0000 0000 0000 0000 0000 1101 1100 0000
$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	0000 0000 0000 0000 0011 1101 1100 0000



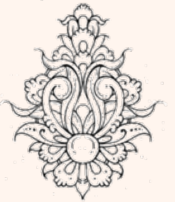
عملگر نقیض بیتی

- عملگر not جای ‘0’ و ‘1’ را عوض می‌کند.
- MIPS عملگر نقیض ندارد!
- MIPS دارای عملگر NOR است.

a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111



دستورات شرطی

یک کامپیوتر چه تفاوتی با ماشین حساب دارد؟!

- پرش به آدرس دستور دیگر اگر شرطی برقرار باشد، وگرنه روال عادی ادامه پیدا کند.

```
beq register1, register2, L1
```

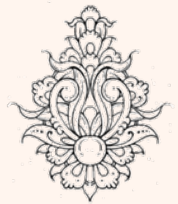
Branch if equal

```
bne register1, register2, L1
```

Branch if not equal

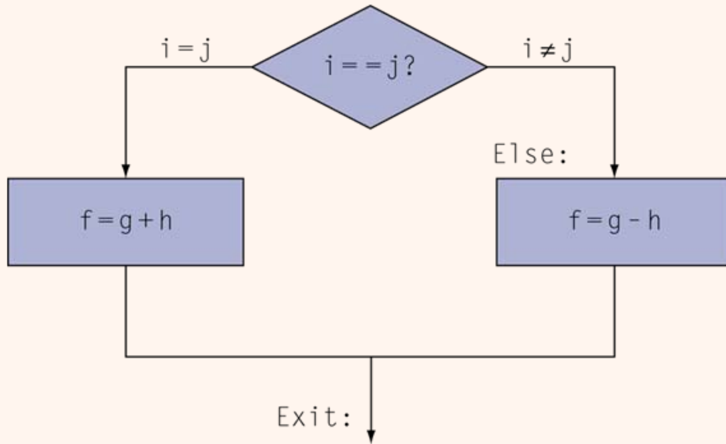
```
j L1
```

unconditional jump



برچسب‌ها در عمل معادل یک آدرس حافظه هستند، آدرس دستوری که باید به آن پرش انجام شود.

مثال



- زبان C

```
if (i==j)
 f = g+h;
else
 f = g-h;
```

f(s0), g(s1), h(s2), i(s3), j(s4)

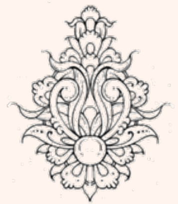
- کد کمپایل شده:

```
      bne $s3, $s4, Else
      add $s0, $s1, $s2
      j   Exit
Else: sub $s0, $s1, $s2
Exit: …
```

آدرس‌ها را اسمبلر محاسبه می‌کند

پرسش

این آدرس‌ها چگونه در سی‌ودو بیت کد ماشین جای می‌گیرند؟



حلقه

Loop

- زبان C

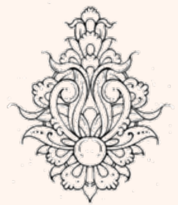
```
while (save[i] == k)
      i += 1;
```

i(s3), k(s5), Adr(save) is in s6

داده‌ها چهاربایتی هستند

- کد کامپایل شده:

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```



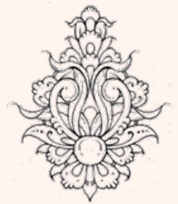
برچسب در دستورهای پرش شرطی

- دستورهای پرش شرطی از «قالب I» استفاده می‌کنند.

```
beq register1, register2, L1
```

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- آدرس هر خانه‌ی حافظه‌ی سی‌ودو بیت است! اما در این قالب شانزده بیت بیشتر فضا اختصاص داده نشده است!



برچسب (ادامه...)

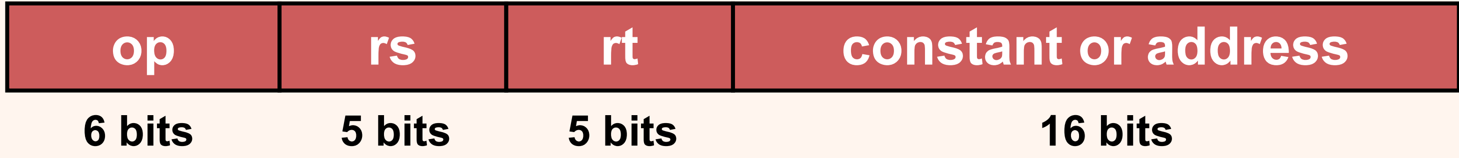
- در این جا نیز بخش آدرس تنها آفست را تعیین می‌کند.
- بخش پایه در یک ثبات قرار دارد.
- این ثبات PC است، در این حالت در بخش آدرس دستور موقعیت نسبی دستورها ذخیره خواهد شد.
- با توجه به این که کلمات (داده‌های چهار بایتی) در حافظه محدودیت هم‌ترازی دارند، اختلاف آدرس‌ها همیشه مضربی از چهار است، یعنی دو بیت کم ارزش اختلاف آدرس دو دستور همیشه صفر است، از این رو می‌توان از ذخیره کردن آن چشم پوشید.

در هنگام اجرای هر دستور PC حاوی آدرس دستور بعدی است

PC-relative addressing

آدرس‌دهی نسبی

- Target address = PC + offset × 4



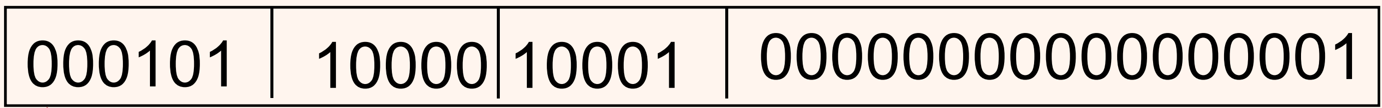
op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

مثال

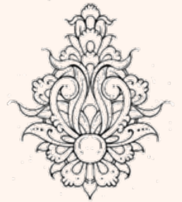
```
if (i==j) h = i + j;
```

i(s0), j(s1)

```
       bne $s0, $s1, Lbl1
       add $s3, $s0, $s1
Lbl1:  ...
```



000101	10000	10001	0000000000000001

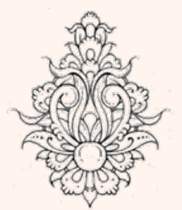


Pseudodirect addressing

آدرس‌دهی شبه‌مستقیم

```
j L1
```

- این دستورات از کدام نوع هستند؟
  - ~~نوع I؟~~
- آخرین نوع دستورها در MIPS، نوع **J** می‌باشد:



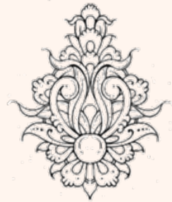
```
Target address = PC[31…28]:(address × 4)
```

الحاق

مثال

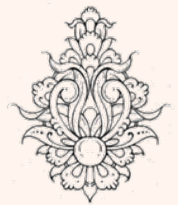
```
while (save[i] == k) i += 1;
```

Code	Address						
Loop: sll $t1, $s3, 2	80000	0	0	19	9	2	0
add $t1, $t1, $s6	80004	0	9	22	9	0	32
lw $t0, 0($t1)	80008	35	9	8	0		
bne $t0, $s5, Exit	80012	5	8	21	2		
addi $s3, $s3, 1	80016	8	19	19	1		
j Loop	80020	2	20000				
Exit: …	80024						



سایر دستورات شرطی

- دستورهای blt و bge
  - سخت‌افزار مدارهای < و یا ≤ نسبت به = یا ≠ کندتر هستند.
  - هنگامی که با پرش همراه شوند، به زمان بیشتری نیاز دارند و در نتیجه پالس‌ساعت کندتر خواهد شد.
  - بدین ترتیب تمام دستورها کند می‌شوند.
- در MIPS دستور پرش در حالتی که ثباتی از دیگری کوچک‌تر باشد، تعبیه نشده است. چنین دستوری پیچیده است در نتیجه استفاده از دو دستور ساده ترجیح داده شده است.



دیگر دستورات شرطی

```
slt rd, rs, rt
```

set on less than

```
if (rs < rt) rd = 1; else rd = 0;
```

```
slti rd, rs, constant
```

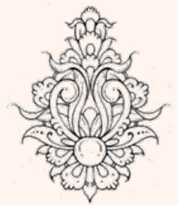
set on less than

```
if (rs < constant) rd = 1; else rd = 0;
```

- به صورت ترکیبی با سایر دستورات شرطی مورد استفاده قرار می‌گیرد.

```
blt $s1, $s2, L
```

```
slt $t0, $s1, $s2   # if ($s1 < $s2)
bne $t0, $zero, L   #   branch to L
```



پیاده‌سازی پرش شرطی

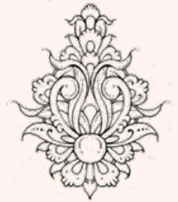
- در صورتی که نیاز به پرش شرطی داشتیم که فاصله‌ی نسبی آن با آدرس فعلی به بیش از شانزده بیت نیاز داشت، چه باید کرد؟

```
beq $s0,$s1, L1
```

مثلاً آدرس L1 خیلی دور است!

```
  bne $s0,$s1, L2
  j L1
L2:    …
```

البته زحمت انجام این کار به عهده‌ی اسمبلر است!

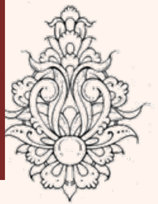


شبه‌دستور

- بیشتر دستورات اسمبلی دقیقاً دستوری معادل در زبان ماشین دارند.
- **شبه‌دستور:** دستورات اسمبلی که در عمل بر روی سخت‌افزار اجرا نمی‌شوند، بلکه در برابر آن‌ها سایر دستورات اجرا خواهند شد.

```
move $t0, $t1  →    add $t0, $zero, $t1
```

```
blt $t0, $t1, L       →   slt $at, $t0, $t1
                          bne $at, $zero, L
```



```
$at (register 1): assembler temporary
```

برای جلوگیری از تداخل، اسمبلر تنها حق دارد، از $at برای شبه‌دستورها استفاده کند.

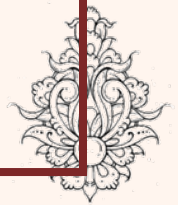
اسمبلر (شبه‌دستور)

```
not   $s0                # complement ($s0)
```

```
nor   $s0,$s0,$zero      # complement ($s0)
```

```
abs   $t0,$s0            # put |($s0)| into $t0
```

```
      add   $t0,$s0,$zero    # copy x into $t0
      slt   $at,$t0,$zero    # is x negative?
      beq   $at,$zero, l     # if not, skip next instr
      sub   $t0,$zero,$s0    # the result is 0 - x
l:
```



```
li    $t0,im16           # put im16 into $t0
```

```
addi  $t0,$zero,im16
```

انتقال داده‌های کوچک‌تر به حافظه

- می‌توان از عملگرهای بیتی استفاده کرد.
- با توجه به پرکاربرد بودن، MIPS دستورهای جداگانه‌ای تعریف کرده است:

```
lb rt, offset(rs)
```

load byte

```
sb rt, offset(rs)
```

store byte

```
lh rt, offset(rs)
```

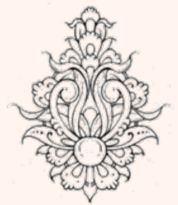
load half word

```
sh rt, offset(rs)
```

store half word

گسترش بیت علامت

- تا اینجا داده‌های چهار بایتی به ثبات‌های چهار بایتی انتقال پیدا می‌کردند، در صورتی که بخواهیم داده‌ای تک‌بایتی و یا دو بایتی را به ثبات‌ها منتقل کنیم، می‌توان از دستورهای گفته شد، استفاده کرد. در این حالت بخش پر ارزش با *بیت علامت* پر می‌شود.
- در برخورد با اعداد علامت‌دار، در صورتی که داده در فضای کم‌تر به فضایی بزرگ‌تر انتقال یابد، ارزش مکانی بیت‌ها عوض خواهد شد.
  - (یادآوری: در مکمل ۲ بیت‌علامت را با وزن منفی در نظر می‌گرفتیم.)
- با تکرار بیت علامت، مقدار عدد تغییری نخواهد کرد.
- آیا استفاده از این دستورها برای اعداد بدون علامت مجاز است؟



خواندن یک بایت (بدون علامت)

- برای اعداد بدون علامت از دستورهای زیر استفاده می‌شود.
- در این دستورها بخش پرارزش با صفر پر خواهد شد.

```
lhu rt, offset(rs)
```

load half word unsigned

```
lbu rt, offset(rs)
```

load byte unsigned

در دستورهای مقایسه نیز علامت را در نظر نگرفتیم،
مقایسه‌ی اعداد بدون علامت و علامت‌دار چه تفاوتی دارند؟

مقایسه و علامت

- دستورات slt و slti در مقایسه، اعداد را به صورت مکمل ۲ در نظر می‌گیرند، برای مقایسه‌ی بدون علامت از دستورهای sltu و sltui استفاده می‌شود.

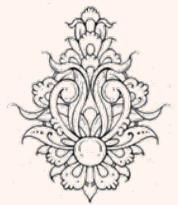
```
$s0 = 1111 1111 1111 1111 1111 1111 1111 1111
$s1 = 0000 0000 0000 0000 0000 0000 0000 0001
```

```
slt  $t0, $s0, $s1  # signed
```

$-1 < +1 \Rightarrow \$t0 = 1$

```
sltu $t0, $s0, $s1  # unsigned
```

$+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

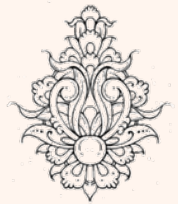


مثال

```
//out of bound check, bound is positive
if (i>=0 and i<bound)
     DO SOMETHING;
```

هر دو علامت‌دار هستند   **i(s0), bound(s1)**

```
     sltu     $t0, $s0, $s1
     beq      $t0, $zero, exit
     Do something
exit:
```



در اینجا با استفاده از مقایسه‌ی اعداد بدون علامت برای اعداد علامت‌دار می‌توان تعداد دستورها را کاهش داد.

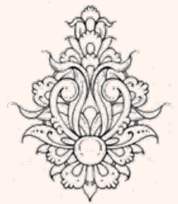
دستور NOP

0x00000000

این دستور زبان ماشین چه کاری انجام می‌دهد؟

```
sll $0,$0,0
```

```
nop
```

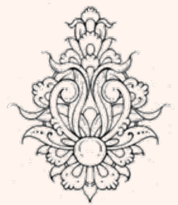


دستور nop (noop) در عمل هیچ کاری انجام نمی‌دهد.
چنین دستوری اثری بر حالت جاری ماشین ندارد؛
هیچ کدام از ثبات‌ها را تغییر نمی‌دهد.
بیشتر برای اهداف زمانی از این دستور استفاده می‌شود.

دستور پرش غیر مستقیم ثباتی

- پیش از اجرای این دستور، آدرس مقصد در یک ثبات قرار می‌گیرد.
- با اجرای آن کنترل برنامه به آدرسی که در ثبات عملوند قرار دارد، منتقل خواهد شد.
- برای کامپایل کردن دستور switch/case می‌تواند مورد استفاده قرار گیرد.

```
jr register
```
#PC = register



عملیات ضرب

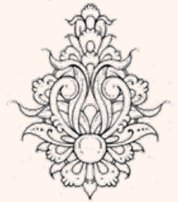
Multiplicand	$1100_2 = 12$	مضروب
Multiplier	$\times \quad 1101_2 = 13$	ضرب‌کننده

```
      1100
     0000
    1100
   1100
```

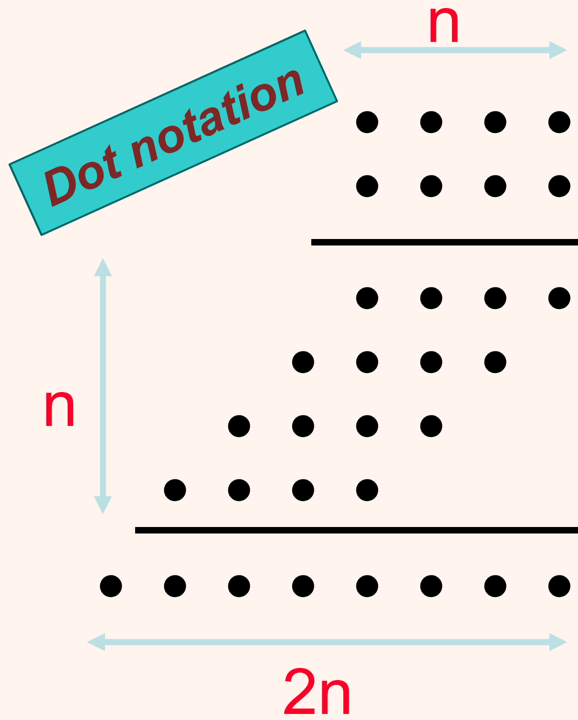
Product	$10011100_2 = 156$	حاصل ضرب

حاصل ضرب یک عدد m بیتی در یک عدد n بیتی یک عدد (n+m) بیتی است.

با یک سری شیفت و جمع متوالی می‌توان ضرب را انجام داد.



عملیات ضرب(ادامه...)



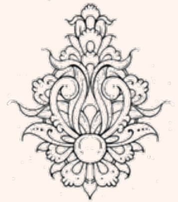
مضروب

ضرب‌کننده

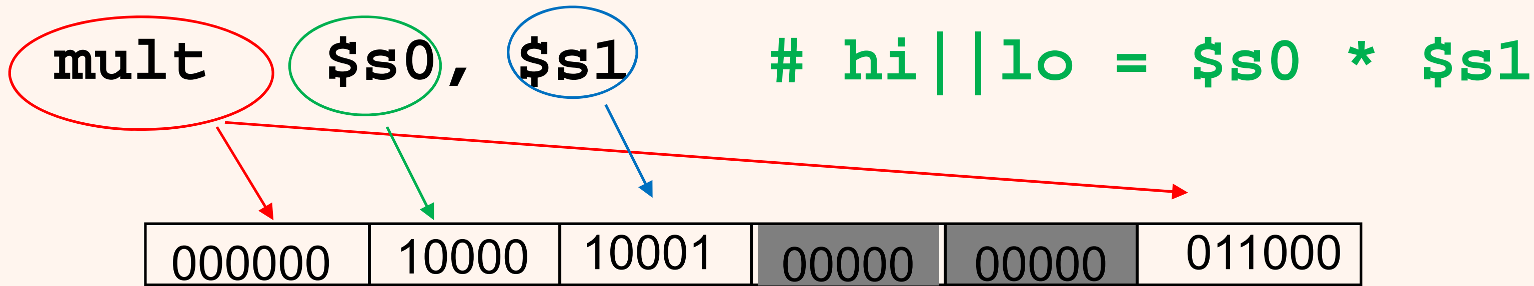
این بخش به صورت موازی توسط سخت‌افزار قابل انجام است.

حاصل ضرب

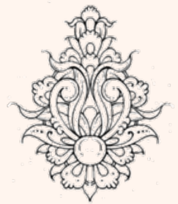
در زبان‌های سطح بالا مقصد و منبع‌ها از یک نوع هستند در نتیجه کل حاصل ضرب قابل ذخیره‌سازی نیست.



عملیات ضرب اعداد علامت‌دار



- در این‌جا خبری از عملوند مقصد نیست!
- عملوند مقصد به صورت «ضمنی» (implied) برای ماشین مشخص است.
- مقصد عمل ضرب دو ثبات به نام‌های lo و hi هستند که قسمت کم‌ارزش و پرارزش ضرب به ترتیب در این ثبات‌ها قرار می‌گیرد.
- این ثبات‌ها قابل استفاده در دستورهای عادی نیستند؛ آدرس‌پذیر توسط کاربر نیستند.



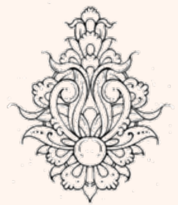
عملیات ضرب(ادامه...)

- با استفاده از دستورهای زیر می‌توان آن‌ها را به ثبات‌های آدرس‌پذیر منتقل کرد.

mfhi rd move from hi

mflo rd move from lo

- در این دستورها نیز تنها مقصد مشخص شده است و منبع به صورت ضمنی مشخص است.
- دستور ضرب گفته شده برای اعداد علامت‌دار است.
- برای ضرب اعداد بدون علامت از دستور multu استفاده می‌شود.



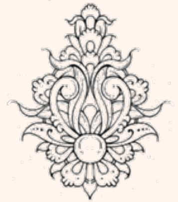
تمرین کلاسی

```
mul   regd,reg1,reg2
```

شبه‌دستور ضرب

معادل دستورهای واقعی ماشین شبه‌دستور فوق را بنویسید.

```
mult reg1, reg2
mflo regd
```

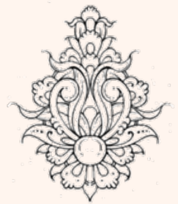


تقسیم

```
div(divu)   $s0, $s1
```

```
# lo= $s0/$s1, hi= $s0%$s1
```

- با انجام دستور تقسیم، «خارج‌قسمت» و «باقیمانده» در دو ثبات lo و hi قرار می‌گیرد.
- در این دستورها نیز عملوند مقصد به صورت **ضمنی** مشخص شده است.
- در تقسیم علامت‌دار، قوانین زیر باید رعایت شود:



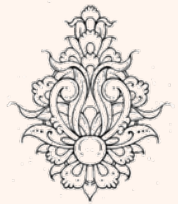
(1) Quotient × divisor + **remainder = dividend**
(2) |remainder| < |divisor|
(3) |A/B| must be equal to |A|/|B|.

ثابت‌های سی‌ودوبیتی

- بیشتر ثابت‌هایی که مورد استفاده قرار می‌گیرند، کوچک هستند و در شانزده بیت می‌گنجند.
- به ندرت مواردی پیش می‌آید که به ثابت‌هایی بزرگ نیاز داشته باشیم.
- در این موارد می‌توان داده‌ی مورد نیاز را در ثبات بارگذاری نمود و از دستورات که دارای عملوند ثبات هستند، استفاده کرد.
- هرچند، برای این منظور دستور زیر پیش‌بینی شده است:

```
lui rt, constant
```

load upper immediate



این دستور، مقدار ثابت را در شانزده بیت پرارزش قرار می‌دهد و بخش کم ارزش را صفر می‌کند.

ثابت‌های سی‌ودوبیتی (ادامه...)

```
lui $s0, 61
```

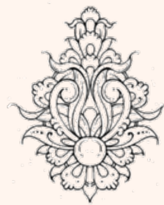
0000 0000 0111 1101	0000 0000 0000 0000

برای قرار دادن بخش کم ارزش چه پیشنهادی دارید؟

```
ori $s0, $s0, 2304
```

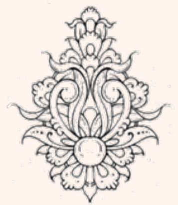
0000 0000 0111 1101	0000 1001 0000 0000

به نظر شما می‌توان در این حالت از دستور **addi** استفاده کرد؟



ثابت‌های سی‌ودوبیتی (ادامه...)

- کوچک بودن اندازه‌ی فیلد داده‌های ثابت، برای آدرس‌ها و به ویژه در دستورات lw و sw ایجاد دشواری می‌کند.
- در زبان‌های سطح بالا در این خصوص محدودیتی وجود ندارد.
- علیرغم محدودیت ذکر شده، در زبان اسمبلی می‌توان از ثابت با طول تا ۳۲ بیت هم استفاده کرد.
- ثابت‌های بزرگ توسط اسمبلر (کامپایلر) به اعداد کوچک‌تر شکسته شده و در یک ثبات جمع‌آوری می‌شوند.

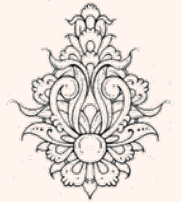


تمرین کلاسی

```
addi   $s0, $s1, 0x10000001
```

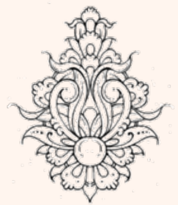
• دستور فوق با چه دستوری (هایی) جایگزین می‌شود؟

```
lui    $at, 0x1000
ori    $at, $at,0x0001
add    $s0, $s1, $at
```



فراخوانی روال

- برای فراخوانی یک روال، مراحل زیر انجام می‌شود:
  - ارسال پارامترها به روال
  - انتقال کنترل به روال
  - تخصیص حافظه‌ی مورد نیاز
  - اجرای روال
  - انتقال نتیجه‌ی به دست آمده به برنامه‌ی اصلی
  - بازگرداندن کنترل به برنامه‌ی اصلی



ارسال پارامترها

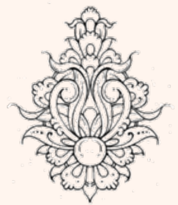
- در MIPS، برای انتقال پارامترها از ثبات‌ها استفاده می‌شود.

arguments

  - $a0 – $a3:
  - برای پارامترهای ارسالی (ثبات شماره ۴ تا ۷)

result values

  - $v0, $v1:
  - برای مقادیر برگشتی فرستاده شده(ثبات شماره ۲ تا ۳)
  - $ra: return address
  - آدرس بازگشت در این ثبات ذخیره می‌شود. (ثبات شماره ۳۱)



- قرارداد: در جریان اجرای تابع محتوای ثبات‌های S نباید تغییر کند.

دستور فراخوانی تابع

```
jal ProcedureLabel
```

jump-and-link instruction

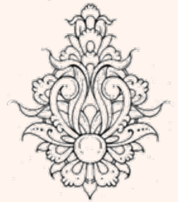
jal funcadd

#$31 = PC;

PC = funcadd

- با اجرای این دستور، افزون بر پرش به آدرس شروع رویه، آدرس بازگشت در $ra قرار می‌گیرد.
- برای بازگشت به برنامه کافیست از دستور پرشی که پیش از این با آن آشنا شدیم، استفاده کنیم.

```
jr $ra
```

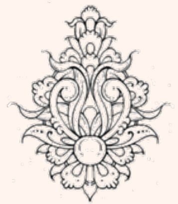


مثال

- زبان C

```
int leaf_example (int g, int h, int i, int j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- آرگومان‌ها در $a0 تا $a3 قرار داده می‌شوند.



ادامه‌ی مثال

- در این مثال برای f متغیر محلی در نظر نمی‌گیریم.
- MIPS code:

```
leaf_example:
    add   $t0, $a0, $a1
    add   $t1, $a2, $a3
    sub   $t2, $t0, $t1
    add   $v0, $t2, $zero
    jr    $ra
```

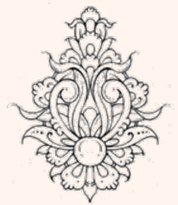
بدنه‌ی روال

نتیجه

بازگشت

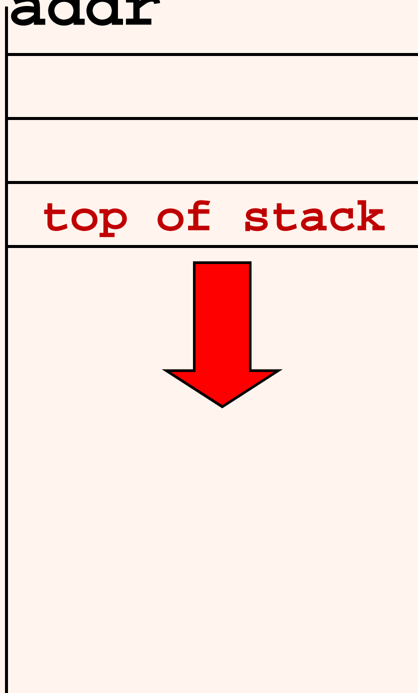
```
#preparing arguments
jal leaf_example
#using the results
```

برای فراخوانی:



پشته

- در صورتی که «تابع فراخوانی شده»(callee) به تعداد پارامتر بیشتری احتیاج داشته باشد(بیش از تعداد ثبات‌ها) یا تعداد خروجی بیشتری لازم باشد، این کار از طریق پشته صورت می‌پذیرد.



$sp

برای دسترسی به پشته از ثبات $sp استفاده می‌شود

push rs→

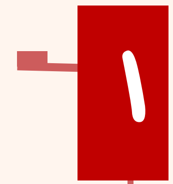
```
addi $sp,$sp, -4
sw $rs,($sp)
```

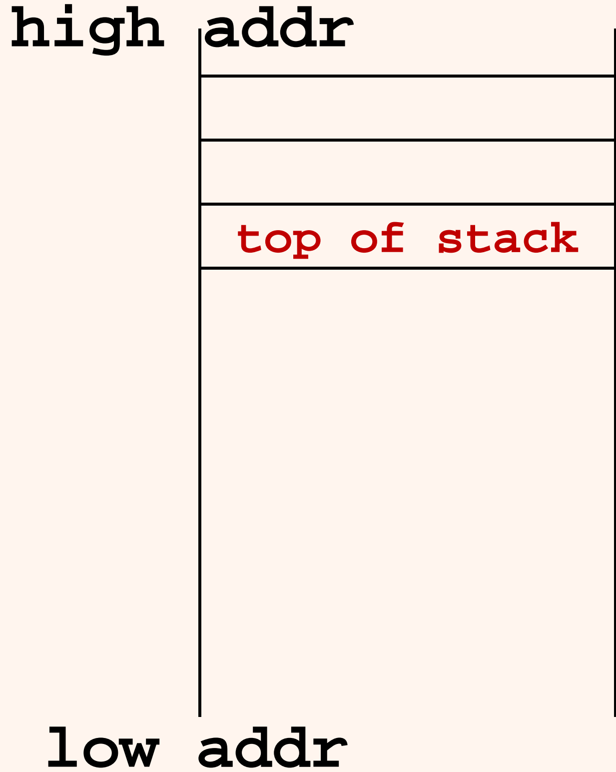
pop rd→

```
lw $rd,($sp)
addi $sp,$sp,4
```

البته با استفاده از آفست می‌توانیم به سایر خانه‌های پشته دسترسی داشته باشیم

push



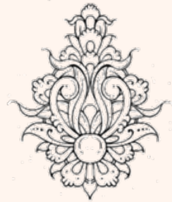


$sp

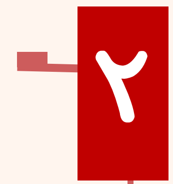
push rs

addi $sp,$sp, -4

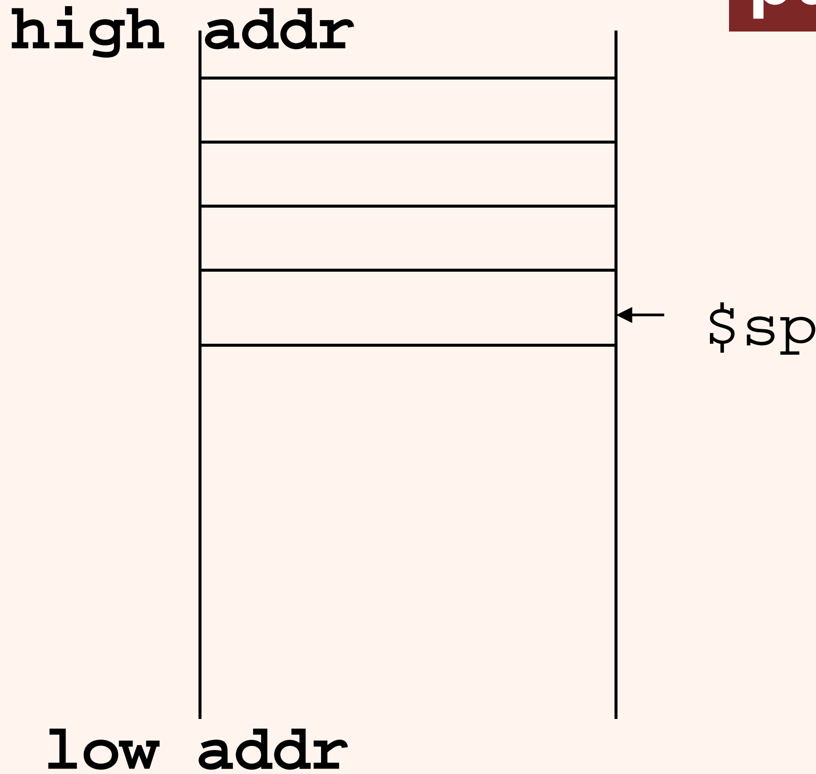
rs



push



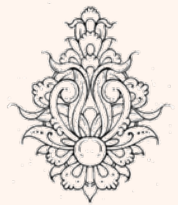
push rs

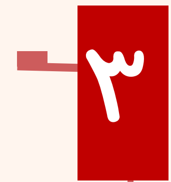


addi $sp,$sp, -4

sw $rs,($sp)

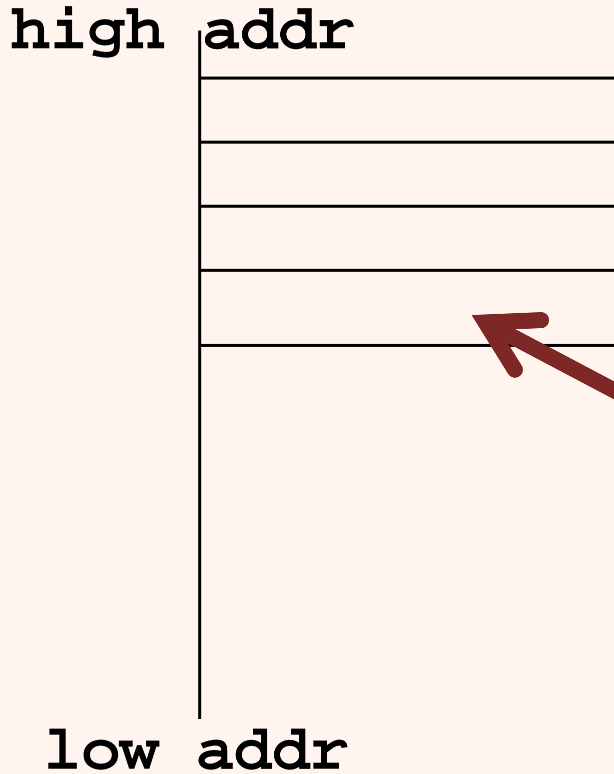
rs





push

push rd

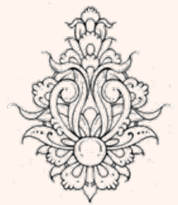


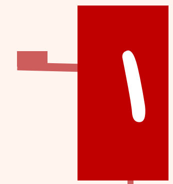
$sp

addi $sp,$sp, -4

sw $rs,($sp)

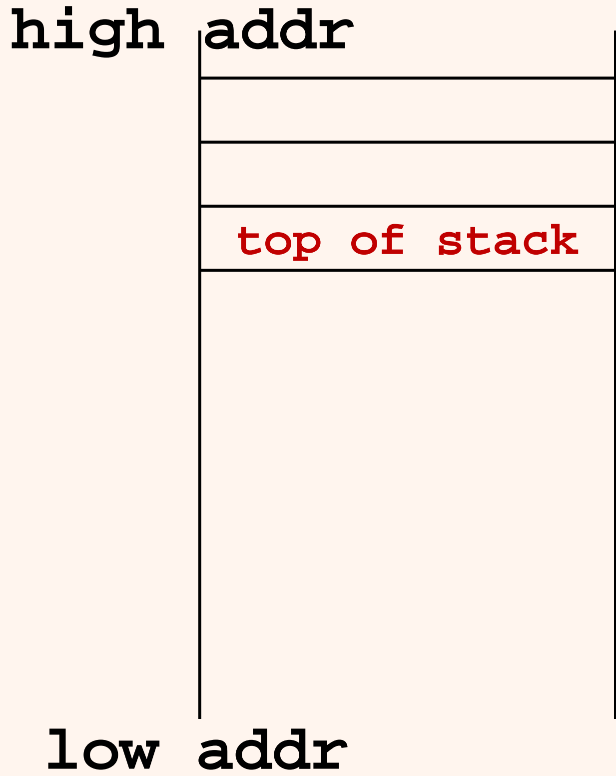
rs





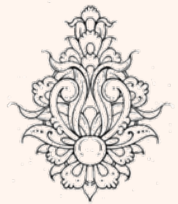
pop

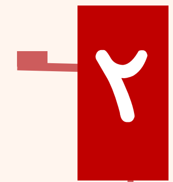
pop rd



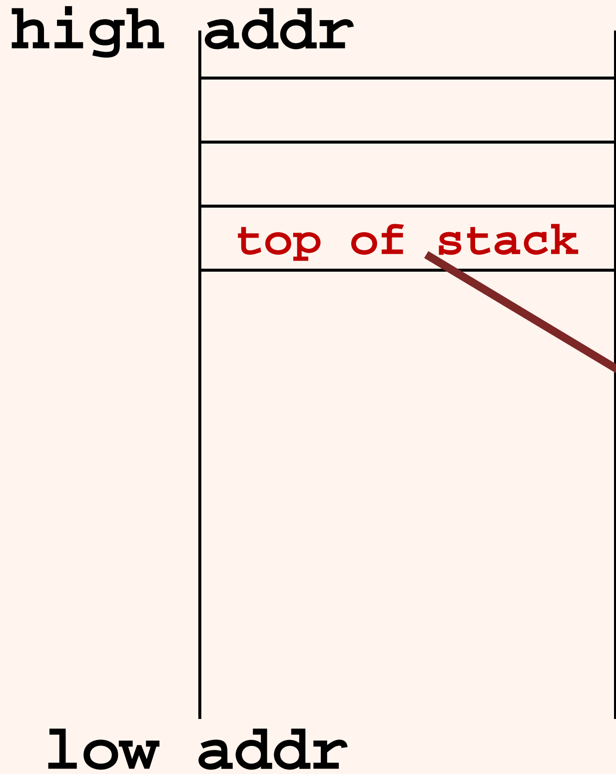
$sp

lw $rd,($sp)





pop

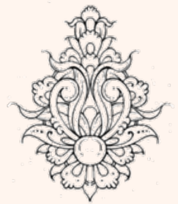


pop rs

lw $rd,($sp)

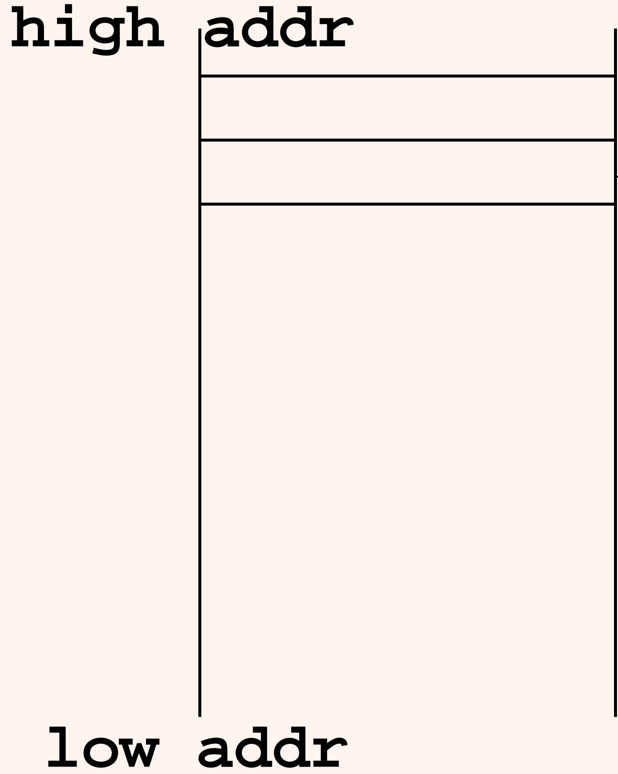
addi $sp,$sp, 4

rd



pop

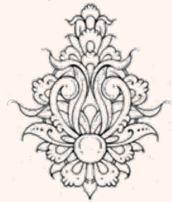
pop rd



lw $rd,($sp)

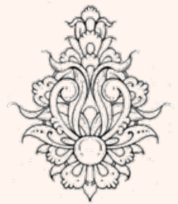
addi $sp,$sp, 4

rd



پشته

- در صورتی که در بدنه‌ی روال، نیاز به استفاده از ثبات‌هایی بود که باید مقدار آن حفظ شود، می‌توان قبل از اجرای بدنه‌ی روال آن‌ها را در پشته ذخیره کرد و قبل از بازگشت، آن‌ها را دوباره بازیابی نمود.
- بدین ترتیب اجرای تابع، اثری روی ثبات‌ها نخواهد داشت.



مثال

در صورتی که در مثال قبل بخواهیم در تابع از مقدار s0 برای f استفاده کنیم:

- MIPS code:

```
leaf_example:
      addi $sp, $sp, -4
      sw   $s0, 0($sp)
      add  $t0, $a0, $a1
      add  $t1, $a2, $a3
      sub  $s0, $t0, $t1
      add  $v0, $s0, $zero
      lw   $s0, 0($sp)
      addi $sp, $sp, 4
      jr   $ra
```

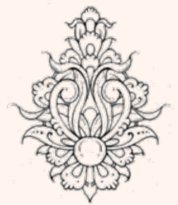
ذخیره‌ی $s0

بدنه‌ی روال

نتیجه

بازیابی $s0

بازگشت



فراخوانی روال‌های تودرتو

- هنگامی که یک روال، روال دیگری را صدا می‌زند ثبات‌های آرگومان، نتیجه و آدرس برگشت را باید ذخیره کرد.

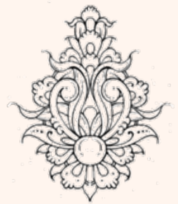
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

تابع بازگشتی

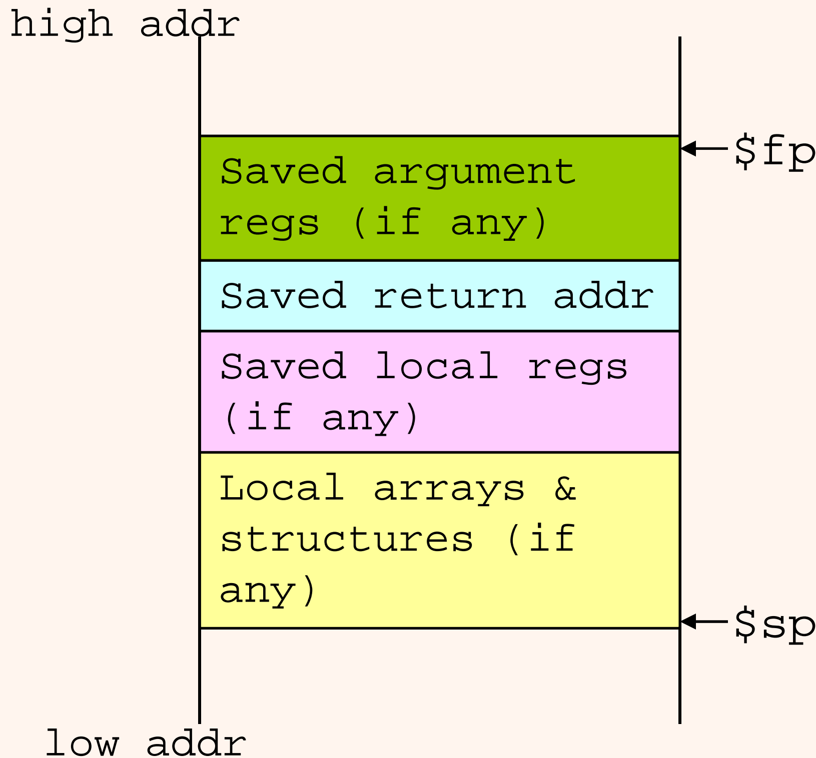
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

- MIPS code:

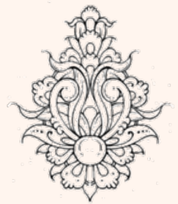
```
fact:
      slti $t0, $a0, 1        # test for n < 1
      beq  $t0, $zero, L1
      addi $v0, $zero, 1      # if so, result is 1
      jr   $ra                #   and return
L1:   addi $sp, $sp, -8       # adjust stack for 2 items
      sw   $ra, 4($sp)        # save return address
      sw   $a0, 0($sp)        # save argument
      addi $a0, $a0, -1       # else decrement n
      jal  fact               # recursive call
      lw   $a0, 0($sp)        # restore original n
      lw   $ra, 4($sp)        #   and return address
      addi $sp, $sp, 8        # pop 2 items from stack
      mul  $v0, $a0, $v0      # multiply to get result
      jr   $ra                # and return
```



داده‌های محلی



- برای نمایش متغیرهای محلی می‌توان از ثبات‌ها استفاده کرد. در صورتی که تعداد متغیرها بیش از تعداد ثبات‌ها باشد، از پشته استفاده می‌شود.
- در این حالت، در صورت استفاده از **$sp** مقدار آفست متغیر خواهد بود.

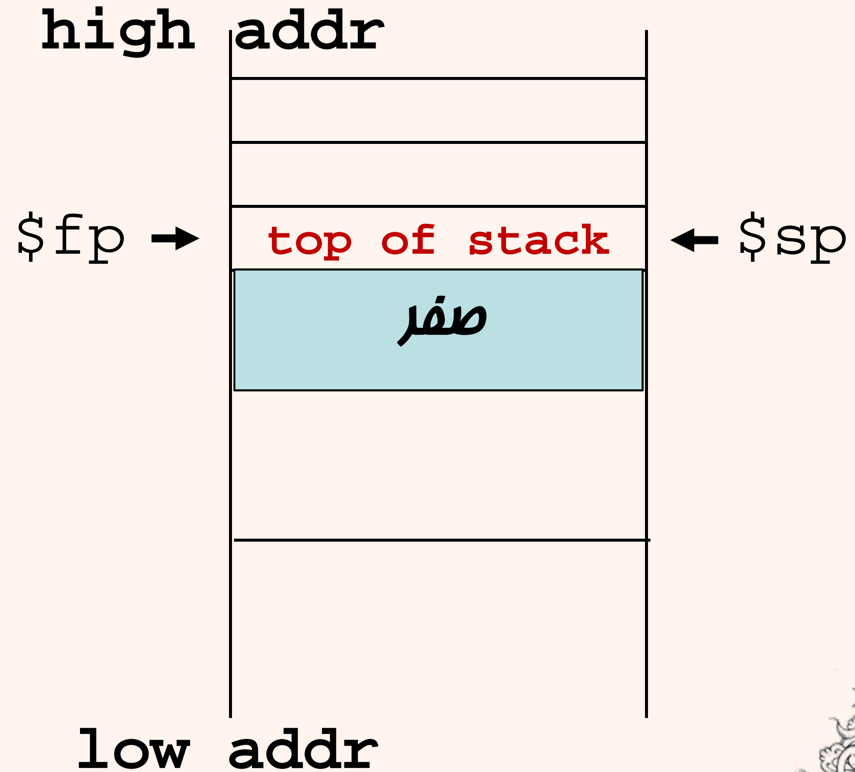


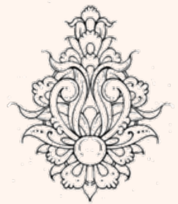
ثبات **$fp** برای دسترسی به داده‌های محلی به کار می‌رود.
به این بخش از پشته **procedure frame** می‌گویند.

داده‌های محلی

```
move $fp,$sp
addi $sp,$sp,-8
```

```
move $s0,$zero
sw $s0,-4($fp)
```



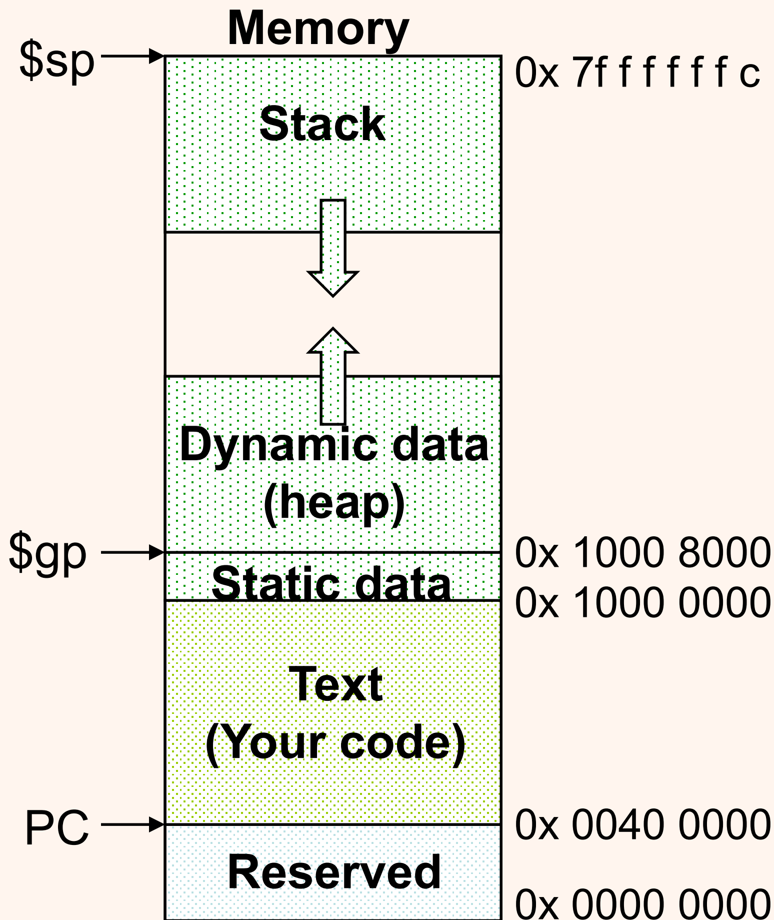


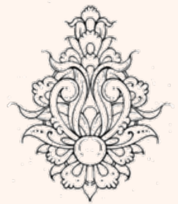
دستورهای فوق در صورتی درست است که ثبات **$fp** به خانه‌ای از حافظه اشاره کند که مضربی از چهار باشد.

Global pointer

داده‌های ایستا

- در زبان c، دو نوع متغیر وجود دارد:
  - automatic
  - static
- در MIPS برای دسترسی به متغیرهای ایستا از رجیستر اشاره‌گر عمومی $gp استفاده می‌شود.





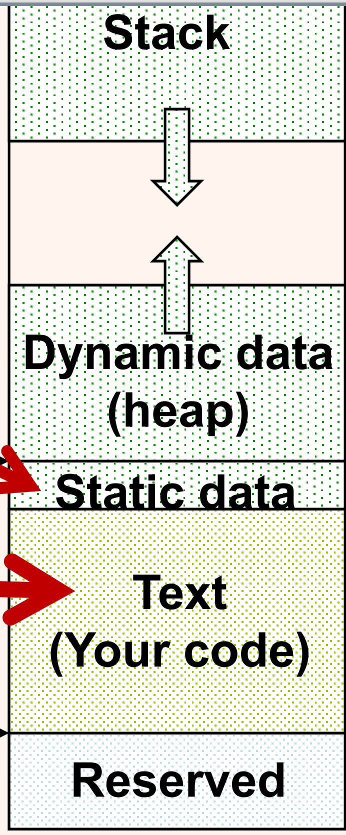
...e	Basic	Source
...11001	lui $1,0x00001001	5: lw $t0, var1
...30000	lw $8,0x00000000($1)	
...90005	addiu $9,$0,0x00000005	6: li $t1, 5
...11001	lui $1,0x00001001	7: sw $t1, var1
...90000	sw $9,0x00000000($1)	

ساخ...

```
.data
var1:    .word    23
array1:  .space   12

.text
    lw    $t0, var1
    li    $t1, 5
    sw    $t1, var1
```

...x 7f f f f f f c



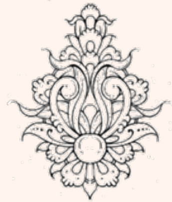
$gp → 0x 1000 8000

0x 1000 0000

PC → 0x 0040 0000

0x 0000 0000

کد در این قسمت نوشته می‌شود

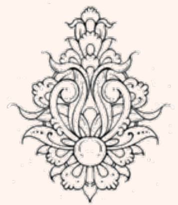


فراخوانی سیستمی

- برای خواندن و نوشتن داده نیاز به ارتباط با سخت‌افزار داریم، این کار از طریق سیستم‌عامل انجام می‌شود. در واقع از طریق فراخوانی توابع سیستم‌عامل این کار انجام می‌شود.
- برای این کار دستور زیر مطرح شده است:

```
syscall
```

- عملوندهای این دستور به صورت بلاواسطه است. بدین‌ترتیب که شماره‌ی تابع در v0 و پارامتر در ثبات‌های آرگومان قرار می‌گیرند.



مثال

```
li   $v0, 5      # service 5 is read integer
syscall
add $s0, $v0, $zero
```

فراخوانی سیستمی شماره‌ی ۵ یک عدد صحیح از ورودی می‌خواند. تابع فراخوانی شده این عدد را در ثبات **$v0** بر می‌گرداند.

```
add $a0, $s1, $zero
li   $v0, 1           # service 1 is print integer
syscall
```

فراخوانی سیستمی شماره‌ی ۱ یک عدد صحیح را که در ثبات **$a0** قرار دارد را چاپ می‌کند.

MIPS | MARS | License | Bugs/Comments | Acknowledgements | Instruction Set Song

Operand Key for Example Instructions

`label, target`	any textual label
`$t1, $t2, $t3`	any integer register
`$f2, $f4, $f6`	*even-numbered* floating point register
`$f0, $f1, $f3`	*any* floating point register
`$8`	any Coprocessor 0 register

Basic Instructions | Extended (pseudo) Instructions | Directives | Syscalls | Exceptions | Macros

Step 4. Retrieve return values, if any, from result registers as specified.

Example: display the value stored in $t0 on the console

```
li  $v0, 1           # service 1 is print integer
add $a0, $t0, $zero  # load desired value into argument register $a0, using pseudo-op
syscall
```

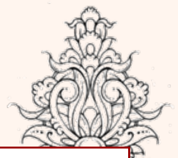
Table of Available Services

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read

مثال

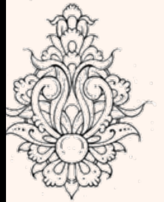
```
1  .data
2    str1: .asciiz  "please enter a number:\n"
3    str2: .asciiz  "the result is:"
4  .text
5       li   $v0, 4            # service 4 is print string
6       la   $a0,str1
7       syscall
8       li   $v0, 5            # service 5 is read integer
9       syscall
10      add $s0, $v0, $zero
11      li   $v0, 4            # service 4 is print string
12      la   $a0,str2
13      syscall
14      li   $v0, 1            # service 1 is print integer
15      add $a0, $s0, $zero
16      syscall
```

```
please enter a number:
7
the result is:7
-- program is finished running (dropped off bottom) --
```



نگاهی کلی به ثبات‌ها

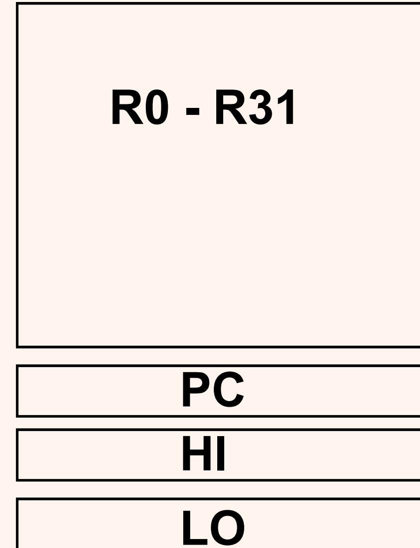
Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr	yes



ISA در MIPS32

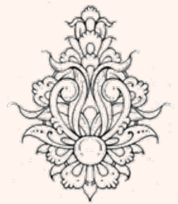
- انواع دستورها:
  - دستورهای محاسباتی
  - Load/store
  - Jump و branch
- سایر دستورهای MIPS بررسی نشده‌اند.

ثبات‌ها



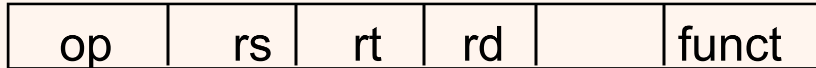
سه قالب برای دستورها

op	rs	rt	rd	sa	funct	R format
op	rs	rt	immediate			I format
op	jump target					J format



مرور شیوه‌های نشانی‌دهی مطرح شده

1. Register addressing

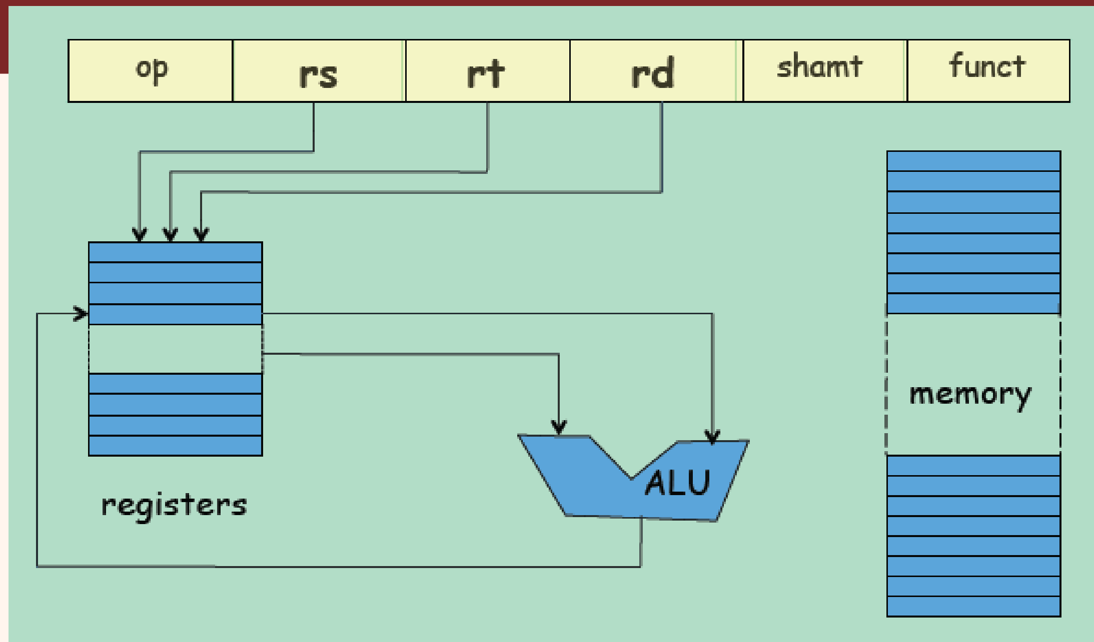


Register

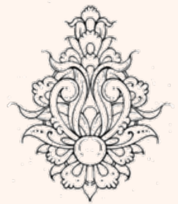
word **operand**

add $1, $2, $3

در این حالت عملوند در یک ثبات قرار دارد، ما تنها آدرس ثبات را مشخص می‌کنیم. با توجه به محدودیت ثبات، فیلد آدرس کوچک خواهد بود روش سریعی می‌باشد که پیاده سازی راحتی هم دارد

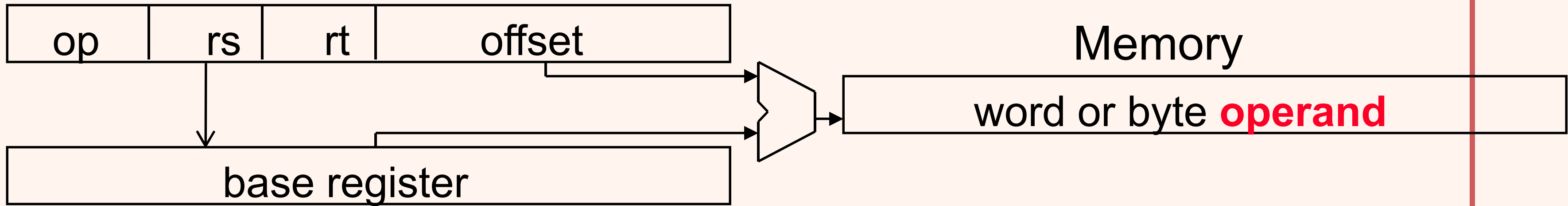


CMPE 110 – Spring 2011 – J. Ferguson

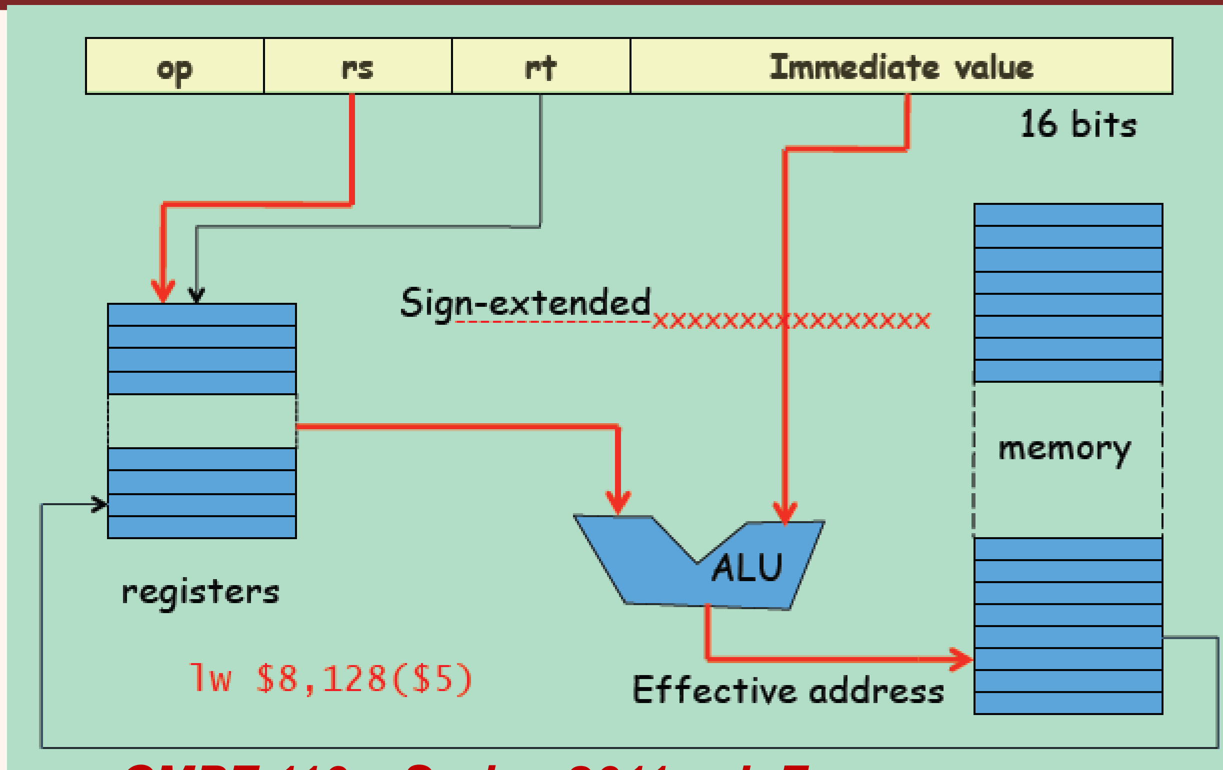


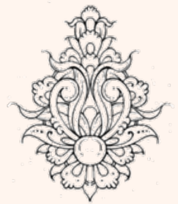
مرور شیوه‌های نشانی‌دهی مطرح شده

2. Base (displacement) addressing



عملوند در حافظه است، اما آدرس واقعی به دو قسمت شکسته شده است:
پایه و جابجایی (آفست)





CMPE 110 – Spring 2011 – J. Ferguson

نشانی‌دهی پایه(مثال)

lw $s1, 10($s2)

$s2==150, M[160] == 12

op	rs	rt	Offset/displacement

register

150

10

Effective address

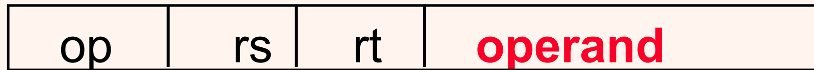
Memory

12

Base or displacement addressing

مرور شیوه‌های نشانی‌دهی مطرح شده

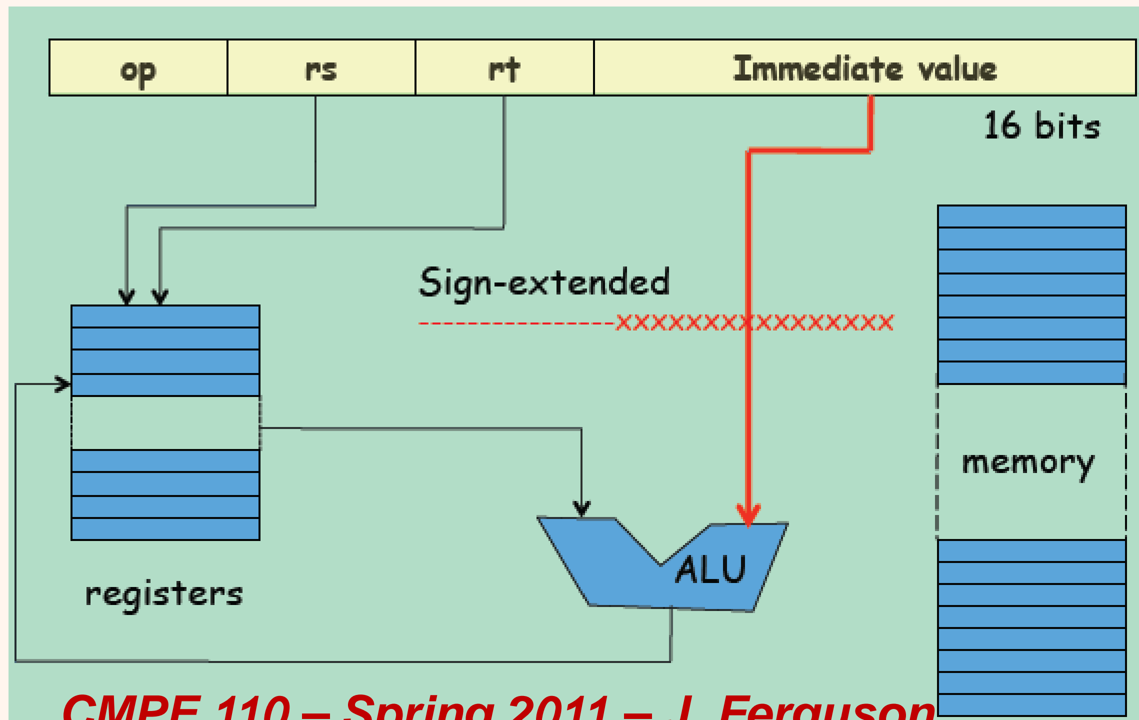
– داده‌ی ثابت به صورت مستقیم مورد استفاده قرار می‌گیرد.

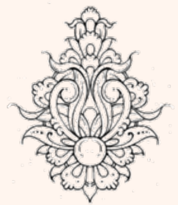


3. Immediate addressing

op	rs	rt	**operand**

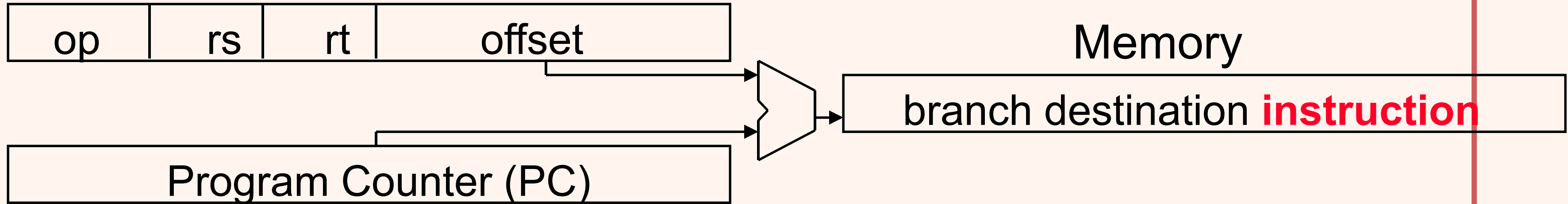
خود عملوند ذکر شده است



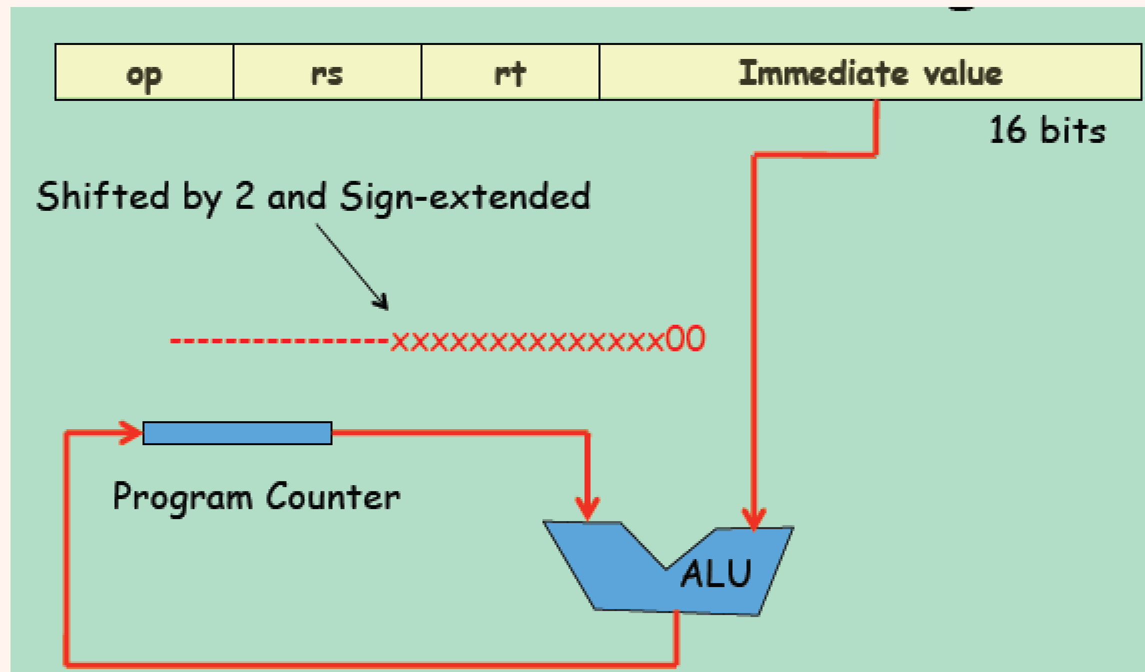


Immediate/literal/constant addressing

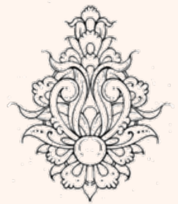
مرور شیوه‌های نشانی‌دهی مطرح شده



عملوند یک آدرس است که به صورت نسبی (نسبت به PC) بیان می‌شود.



CMPE 110 – Spring 2011 – J. Ferguson



آدرس دهی نسبی (نسبت به PC)

- مانند آدرس‌های پرش شرطی که نسبت به آدرس دستور بعدی سنجیده می‌شوند

PC-relative addressing

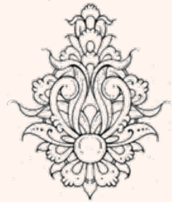
beq $s1, $s2, 100 # if ($1==$2) PC = PC + 100×4

op	*rs*	*rt*	*address*

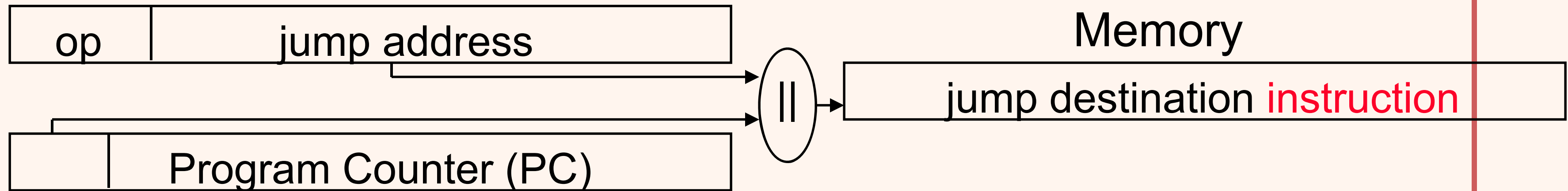
PC

Memory

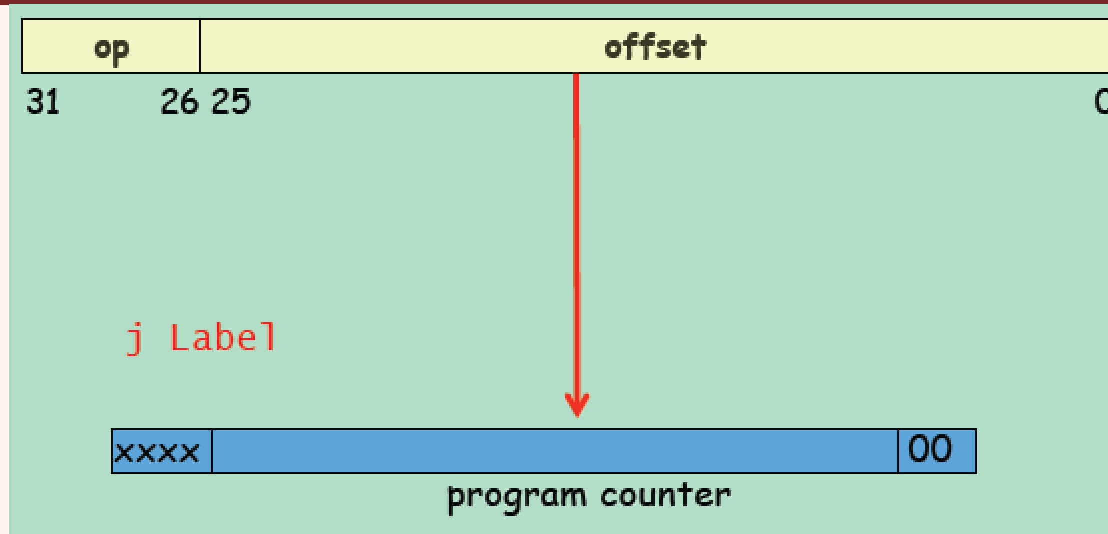
Effective address

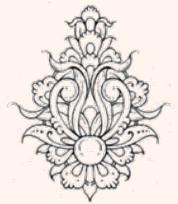


مرور شیوه‌های نشانی‌دهی مطرح شده



عملوند یک آدرس است که بخشی از آن به صورت مستقیم آمده است و بخش دیگر با کمک PC تامین می‌شود.





CMPE 110 – Spring 2011 – J. Ferguson

برخی منابع آن را augmented addressing هم گفته‌اند.

مرور شیوه‌های نشانی‌دهی مطرح شده

– عملوند آشکارا گفته نمی‌شود، هنگامی اجرا به صورت ضمنی برای پردازنده مشخص می‌شود.

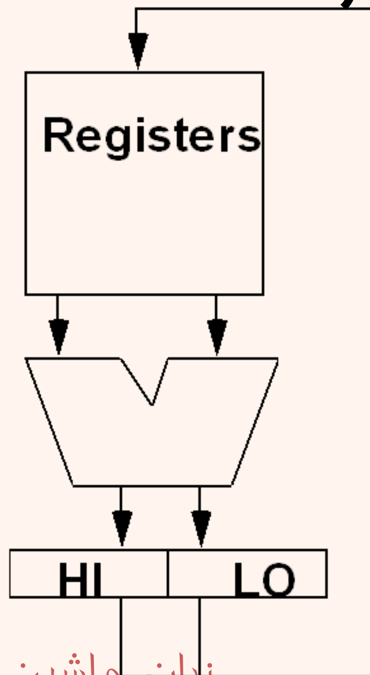
عملوند در دستور مطرح نمی‌شود.

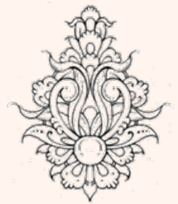
6. Implied addressing

op	rs	rt	operand

– مثال: عملوند مقصد در دستورهای ضرب و تقسیم (ثبات‌های lo و hi)

– عملوند دستور syscall





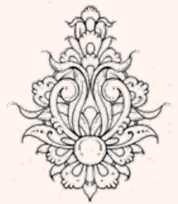
عملوند مقصد ضرب و تقسیم

بخش opcode

بخش کم ارزش

بخش پر ارزش

	000	001	010	011	100	101	110	111
000	REG		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	
010								
011	llo	lhi	trap					
100	lb	lh		lw	lbu	lhu		
101	sb	sh		sw				
110								
111								



بخش function

بخش کم ارزش

بخش پرارزش

	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111