Bilgisayar Organizasyonu
4.Hafta:

2.7. Bilgisayar donanımında prosedürlerin (Fonksiyonların) desteklenmesi *(4.hafta başlangıcı)*

- Alt program (Subroutine - yordam), belirli bir işi yapmaya odaklanmış bağımsız programdır. Kendisine gelen parametrelere göre işlemini yapıp elde ettiğ sonucu dışarıya göndermektedir. Yani ana program veya başka programlar tarafından çağrılan programlardır.

- Alt programlar "Fonksiyon" veya "Prosedür" olarak iki şekilde isimlendirilir. Fonksiyon tek değer hesaplar ve çağırana gönderir. Karekök alma, sinüs kosinüs v.b. Prosedürler ise daha geneldir. Esnek şekilde alt programlar yazılabilir.C programlamada fonksiyon ve prosedür aynı yapıda yazılır.

- *Prosedür;* özel bir görevi başarmak için kullanılan altprogram (function - subroutine – tool) gibi düşünülebilir.

- Bunlar kendileri için tanımlanmış parametrelerle ilgili işlemi başarıp sonucu verirler.

- Prosedürler, programcıya karmaşık bir problemi çözerken, programı parçalara ayırıp, her parçanın ayrı parametreler kullanılarak sonuçlarının diğer bölümlerde kullanılması kolaylığını sağlar.

- Prosedürler sadece kendisi için tanımlanmış işi yapabilir, programda , programcı onu sık sık çağırarak başvurabilir. Yani programın farklı noktalarında yapılması gereken aynı işlemler için de prosedürlere başvurulur.

ANA Program - Fonksiyon Örneği

```
main()
{..
   int x, y, toplam;
.
.
   toplam = topla (x,y);
   z= toplam*y
.
.
topla (int a, int b)
{
   int g;
   g = a+b;
   return g;
}
```

Bir prosedürün yürütülmesi için yazılan bir program aşağıdaki altı adımı yerine getirmelidir.

1- Kullanılacak parametrelerin, prosedürün ulaşabileceği bir yere yerleştirilmesi.

2- Prosedüre transferin kontrolu.

3- Prosedür için gereken hafıza kaynaklarının oluşturulması.

4- İstenen görevin yapılması.

5- Sonuç değerlerinin, *çağıran programın* erişebileceği yerde hafızalanması için bir yer.

6-Bir programda, bir prosedür değişik değişik noktalardan çağrılabilir. Dolayısıyla bu orijin noktaların kontrol edilebilir olması gerekir. Çünkü prosedür tamamlandıktan sonra program tekrar o noktadan devam edebilsin diye.

- Daha önce de belirtildiği gibi, registerler en hızlı veri yerleştirme ve tutma yerleridir. Biz onları verimli kullanmak isteriz.
- MIPS-32'nin ***procedurlar*** için kullandığı registerlerin standart isimleri aşağıdadır.

- ***$a0 - $a3 :***Kullanılacak parametreler için 4 tane argüman registeri; ***Ana programdan (Çağıran -Caller) veya başka bir prosedürden, çağrılan prosedüre (callee) gönderilen giriş argümanları*** içindir.
- ***$v0 - $v1: Prosedürden döndürülen değerler için 2 tane*** register.
- ***$ra :*** Dönülecek orijinal noktayı bilmek için bir tane dönüş adresi registeri (return address tarifi).

- Bu registerlar'a ek olarak; MIPS assemler dili; prosedürler için bir atlama komutu da bulundurur. Bu komut istenen adrese (prosedür adresine) atlar ve aynı zamanda, takip eden komutun adresini de *$ra 'ya* kaydeder. Bu komuta *jump-and-link (jal)* komutu denir.

jal Prosedür adresi

• ***Yürütülmekte olan komutun adresinin saklandığı özel bir register (register dosyasının parçası olmayan) vardır. Buna program sayacı (PC – Program counter) diyoruz.***

•**Jal** komutunun ***prosedür adresi*** kısmı, çağrılacak prosedürün adresidir (prosedürün ilk satırıdır). Bu adres o anda PC'deki aktüel adrestir.

- Bu fonksiyon aynı zamanda , tamamlandıktan sonra ana programda dönülecek yerin adresini; (dönüş adresini), ***$ra*** registerine depolar.
- **Jal komutu**; ***$ra*** registerına geri dönüş komut adresi olarak

  PC (Program Counter) + 4 adresini set eder.
- Prosedürün sonuna eklenen;

  Jr $ra

  (**jump register - jr**) komutu, $ra registerindeki belirtilen adrese şartsız dallanmayı sağlamanın garantisidir.

Dikkat!! Jr $ra → PC= R[ra] (ra reg'inin içeriğini PC'ye koyar.)

- Böylece çağıran program *(calling program - caller - ana program v.b olabilir)*, ilgili parametre değerlerini $a0-$a3 registerlerine koyar ve, jal X ‘ ile X prosedürüne bu parametre değerleri ile dallanır. *(Prosedür, değişik komutların işlendiği bir program parçasıdır. Bazen callee (çağrılan) diye isimlendirilir.)*

- Calle, yani çağrılan program yani, prosedür; kendisinden istenen hesaplamaları başarır ve sonuçlarını *$v0 - $v1* registerlerine koyup, çağıran programda kaldığı yere *jr $ra* komutu ile geri döner.

Prosedürün ihtiyaç duyduğu register sayısı fazla ise ne yapılmalı?

- Prosedürün attığı ilk adım, ana program tarafından kullanılan registerlerin içeriklerini kaydetmek olur. Çünkü bu registerları kendisi de kullanacağından bir önceki içerikler kaybolmamalıdır. Ana programa geri dönmeden önce de registerlerin ilk içeriklerini ilgili registerlere tekrar kaydetmelidir.

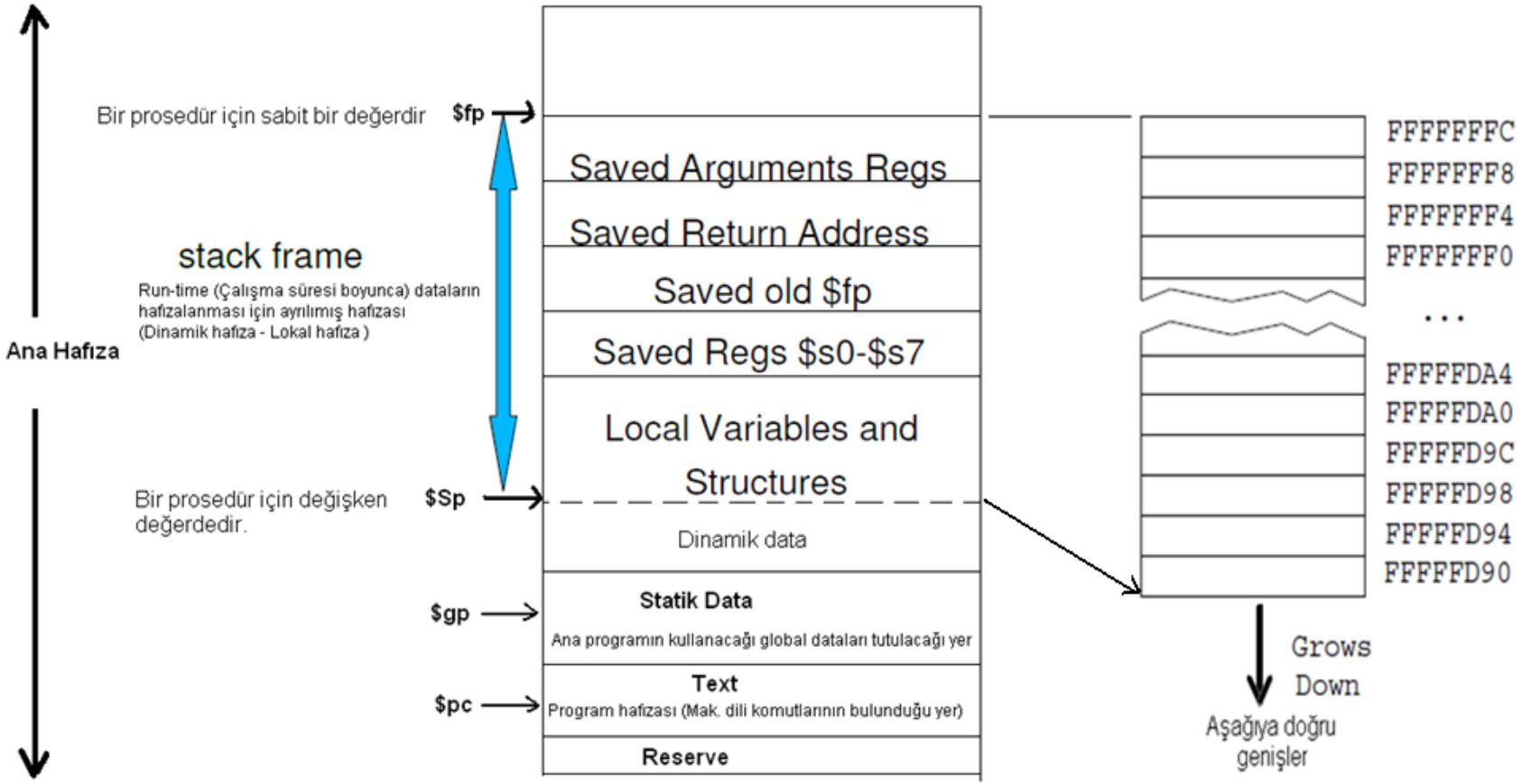
- Registerların kaydedildiği hafıza kısmı spilling memory olarak isimlendirilir.

- *stack = LIFO yapısında, registerlerin saklanma biçimini tarif eden veri yapısıdır.*

- *Stack Pointer ($sp) : Stack'teki en son işlenenin adresini işaret eder.*

- Stack'e yazma için push (itme) okuma için pop (çekme) işlemleri kullanılır.

Bir program için hafıza yapısının basit resmi



Aşağıdaki C prosedürünü inceleyelim

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

İlgili operandlar aşağıdaki registerlardadır. Bu reg'lerin bir özelliği varmıdır?

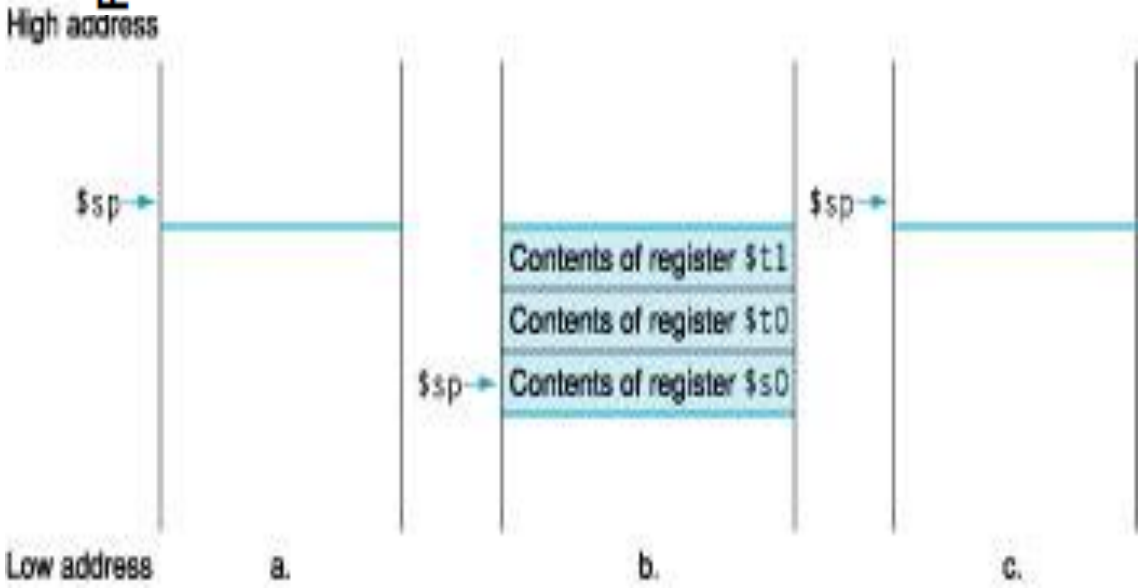
g →$a0, h → $a1, I → $a2, j → $a3, f → $s0

Bu prosedür derlendiğinde MIPS kodu nedir?

Bu prosedürün yapacağı 1.İşlem : Ana programda kullanılabilen reg'leri stack'e depolamaktır. Bu prosedürün, $to, $t1, $s0 reg'lerini kullandığını farz edelim. Stack'e atma işlemi aşağıdaki gibidir.

Register spilling in stack (push)

```
Leaf_example:      # prosedür bu etiketle başlar
addi   $sp, $sp, -12   # Stakc'te  3 birimlik yer açmak
sw     $t1, 8($sp)     # t1'in içeriğini stack'in içine kaydetmek (push)
sw     $t0, 4($sp)     # to'ın içeriğini stack'in içine kaydetmek
sw     $s0, 0($sp)     # s0'ın içeriğini stack'ın içine ittirmek.
```



Şekil: stack ve stack pointer'in değerleri;
a)Prosedür çağrılmadan önce
b)Prosedürün çalışma süresince
c)Prosedür sonlandıktan sonra

Prosedürün MIPS kodu

Register spilling in stack (push)

```
addi $sp, $sp, -12
sw   $t1, 8($sp)
sw   $t0, 4($sp)
sw   $s0, 0($sp)
```

Body or procedure

```
add $t0, $a0, $a1     # $t0 = g + h
add $t1, $a2, $a3     # $t1 = i + j
sub $s0, $t0, $t1     # f = (g + h) - (i + j)
```

Save return value

```
add $v0, $s0, $zero   # to return $v0 = f = $s0 + 0
```

Restore Registers (pop)

```
lw   $s0, 0($sp)      # restore $s0 from stack
lw   $t0, 4($sp)      # restore $t0 from stack
lw   $t1, 8($sp)      # restore $t1 from stack
addi $sp, $sp,12      # stack pointer'i eski yerine
```

Return

```
jr $ra                # return = jump back to caller
```

ÖZET

Caller (= Ana program veya diğer Prosedür)

- – Parametreli, **$a0 -$a3 registerlarına kayıt eder.**
- – **jal ProcedureAddress** prosedürün çağrılmasını sağlar.

- ## *Callee(= Prosedür)*
- – kendinden beklenen hesaplamaları başarır.
- – Elde ettiği sonuçları **$v0 -$v1** geri dönüş reg'lerine kayıt eder.
- – **jr $ra** komutu ile çağıran programa geri döner.
- ***Eğer bir prosedür, s ile başlayan reg'leri kullanacaksa, bunların içerikleri muhakkak korunmalı (stack'e atılabilir).***
- ***Eğer bir prosedür, t ile başlayan reg'leri kullanacaksa, bunların içerikleri korunmayabilir.***

özet

• Jal komutu prosedüre dallanmak için kullanılır ve PC içeriğine 4 ekleyerek geri dönüş adres registerine (ra) yazar.

• Argümanlar $a0-$a3 registerlerine, Geri dönüş değerleri, $v0-$v1 registerlerine atanır.

• Callee,callerin registerlerinin üzerine yazabildiği için, İlişkili değerler hafızaya(Stack) kopyalanmalıdır.

•Her bir procedur, yerel değişkenler için bir hafıza yerine İhtiyaç duyar (Stackt'e yer ayrılması).

•Stack; procedürler için ihtiyaç duyulan organize edilmiş hafıza bölgesidir.

İç İçe prosedürler (Nested Procedures)

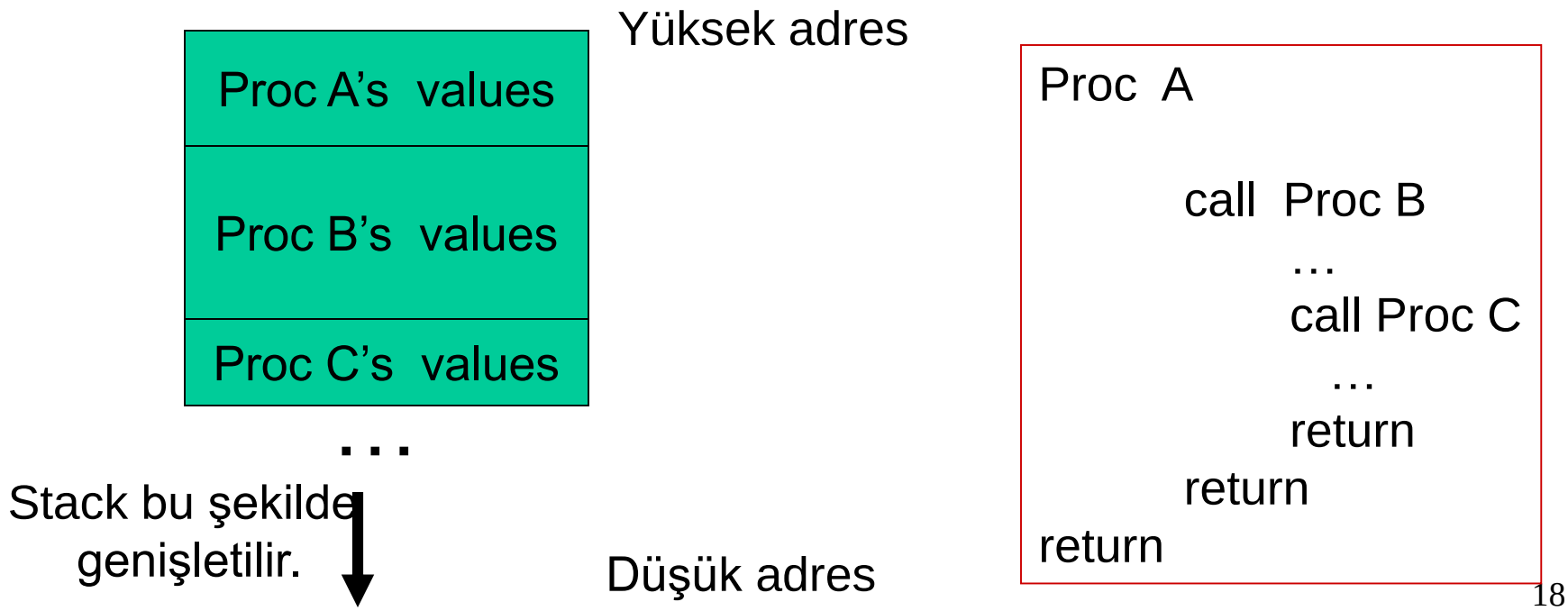
- Başka bir prosedür tarafından çağrılmayan prosedürlere yaprak prosedür (leaf prosedür) denir.
- Kullanılan prosedürler yaprak prosedür olsaydı işler çok kolay olurdu.
- Oysa; bir prosedür tanımlanmış bir işi kolay bir şekilde yapmak için, içerisinde başka bir prosedürün (veya prosedürlerin) çağrılmasına gerek duyabilir.
- Hatta; kendi kendilerini çağıran recursive (Yinelemeli) prosedürler bile mevcuttur.
- Bu şekilde çalışmalarda , kullanılan registerlardaki bilgilerin kaydedilmesi ve geri çağrılması durumları çok önemlidir.

Çalışılan prosedür yaprak prosedür değilse çok dikkat edilmelidir.!!!!!!! *Örneğin;*

- *ProsedürA, 5 parametresi ile ($a0 = 5, $ra 'da set edilerek) ana program tarafından çağrılmış olsun.*
- *–ProsedürB ise 10 parametresi ile ($a0 = 10, $ra dönüş adresi tekrar set edilerek) ProsedürA tarafından çağrılmış olsun.*
- – A prosedürü doğru data ile devam edebilecek midir?
- – A prosedür'ü ana programa doğru bir şekilde dönebilecek midir?
- En son çağrılan prosedür, registerlerin üzerine veri yazabileceği için; ilgili reg'lerdeki değerlerin belleğe (stack) kaydedilmesi gerekir !!!!!TAMAM'MI?
- Önemli not: İç-içe çağrılan prosedürlerde veya rekürsif programlarda her çağırıdan önce bir önceki dallanma yeri ve parametreler stac'te kaydedilmelidir.

Stack (Yığıt)

Procedür'ün kullandığı her bir register değerleri stack'ta yedeklenmelidir. Aşağıda iç içe prosedürler için stack'taki değerlerin yerleşimi gösterilmiştir.

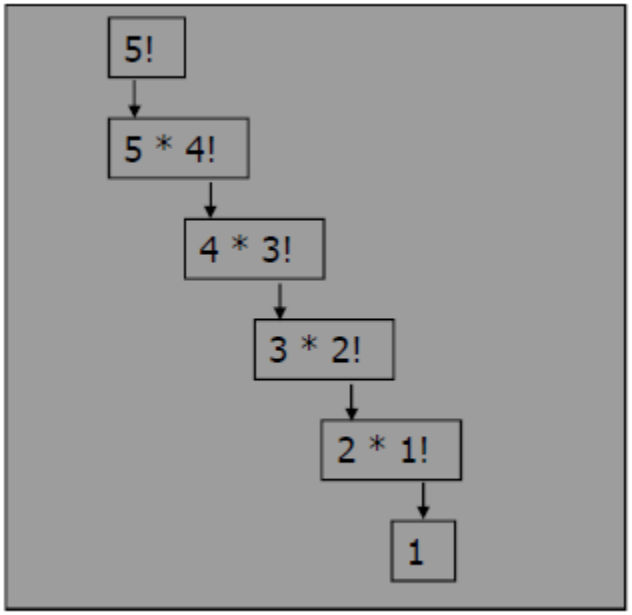


Örnek: Aşağıdaki rekürsif fonksiyonunun MIPS komu seti kodunu Yazınız.

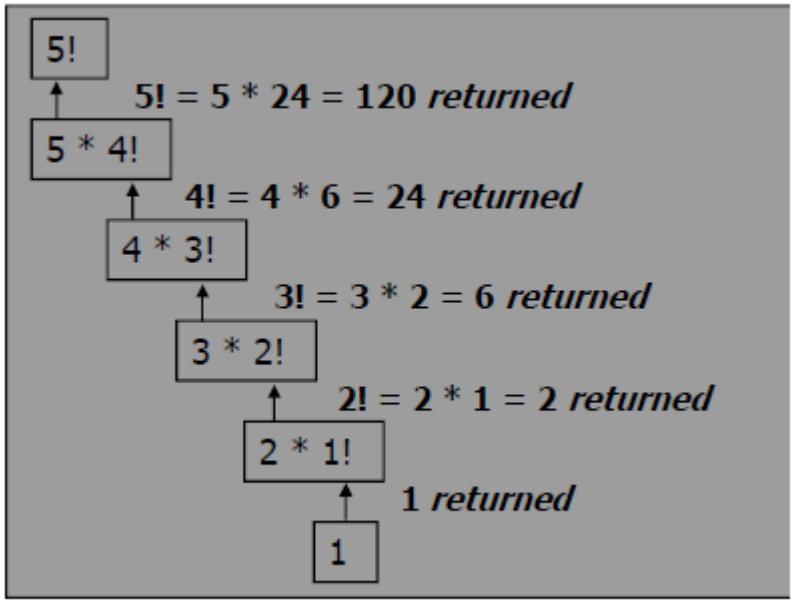
```
main ()
{.
 .
faktoriyel = fact(N)
 .
}
int   fact  (int N)
{
if (N < 1)  return (1);
else return (N* fact ( N - 1));
}
```

N! algoritması

$$N!=N*(N-1)!=N*((N-1)*(N-2)!.....)$$



a. *Process of recursive calls*

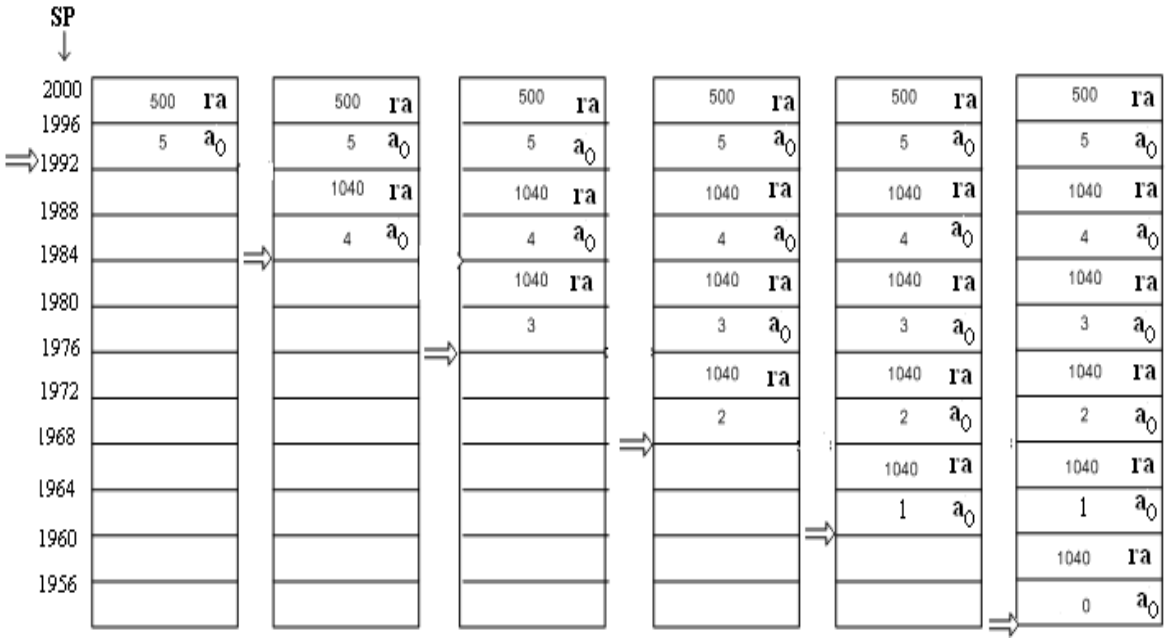


b. *Values returned from each recursive calls*

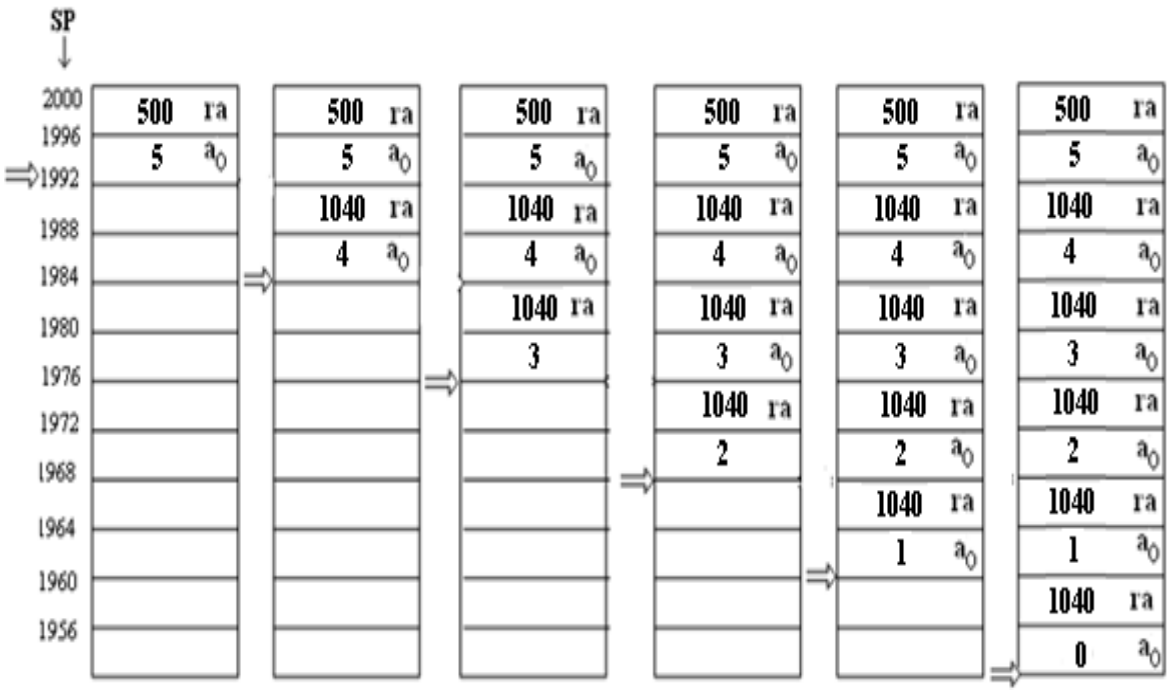
Örnek: Aşağıdaki rekürsif fonksiyonunun MIPS komu seti kodunu Yazınız.

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

Ana programda; çağrılan fact(n) fonksiyonundan sonraki komutun adresi 500 olsun. n=5 olsun.




```
1   1000  fact:  addi  $sp,$sp,-8
    1004         sw    $ra,4($sp)
    1008         sw    $a0,0($sp)

2   1012         slti  $t0,$a0,1
    1016         beq   $t0,$zero,L1

3   1020         addi  $v0,$zero,1
    1024         addi  $sp,$sp,8
    1028         jr    $ra

4   1032  L1:    addi  $a0,$a0,-1
    1036         jal   fact

5   1040         lw    $a0,0($sp)
    1044         lw    $ra,4($sp)
    1048         addi  $sp,$sp,8

6   1052         mul   $v0,$a0,$v0
    1056         jr    $ra
```


SP
2000
1996
1992
1988
1984
1980
1976
1972
1968
1964
1960
1956
500 ra
5 a0
1040 ra
4 a0
3
2
1
0

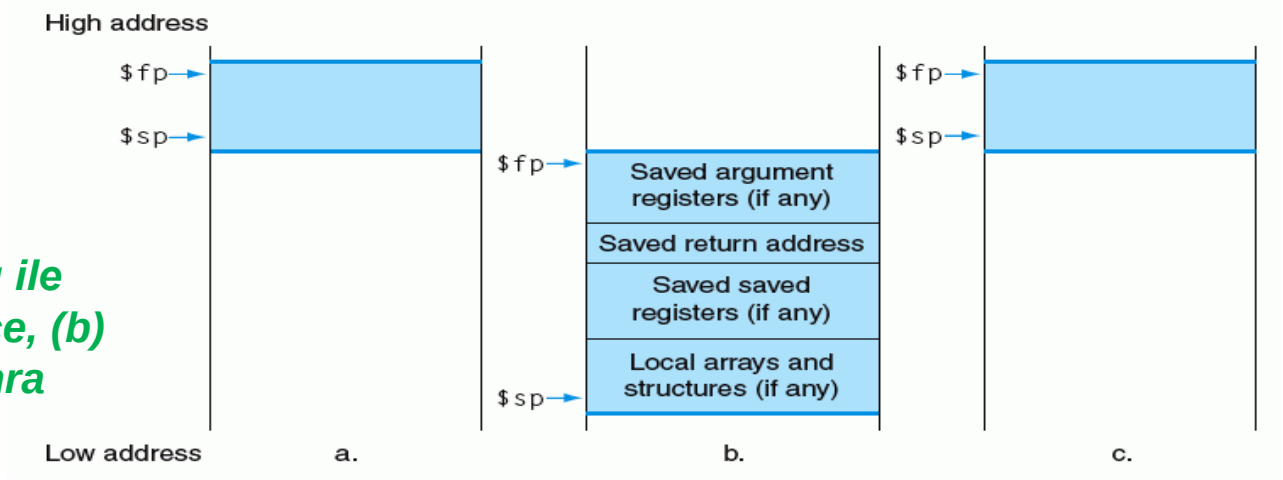
STACK'te yeni veriler için yer ayrılması

•Bir prosedür tarafından stack'ta tahsis edilen yer, "*prosedür frame*" veya "aktivasyon kayıtı" olarak adlandırılır. (Bu ayrılan yer,Prosedür için yerel verileri ve kaydedilen değerleri içerir).

•Frame Pointer; Verilen bir prosedür için yerel değişkenleri ve kaydedilen registerlerin yerini gösteren bir değerdir

• $fp registeri , stac'teki kayıtın başını, $sp sonunu gösterir.

•$fp prosedürün yürütülmesi boyunca değişmez.. $sp Prosedürün yürütmesi boyunca değişebilir.

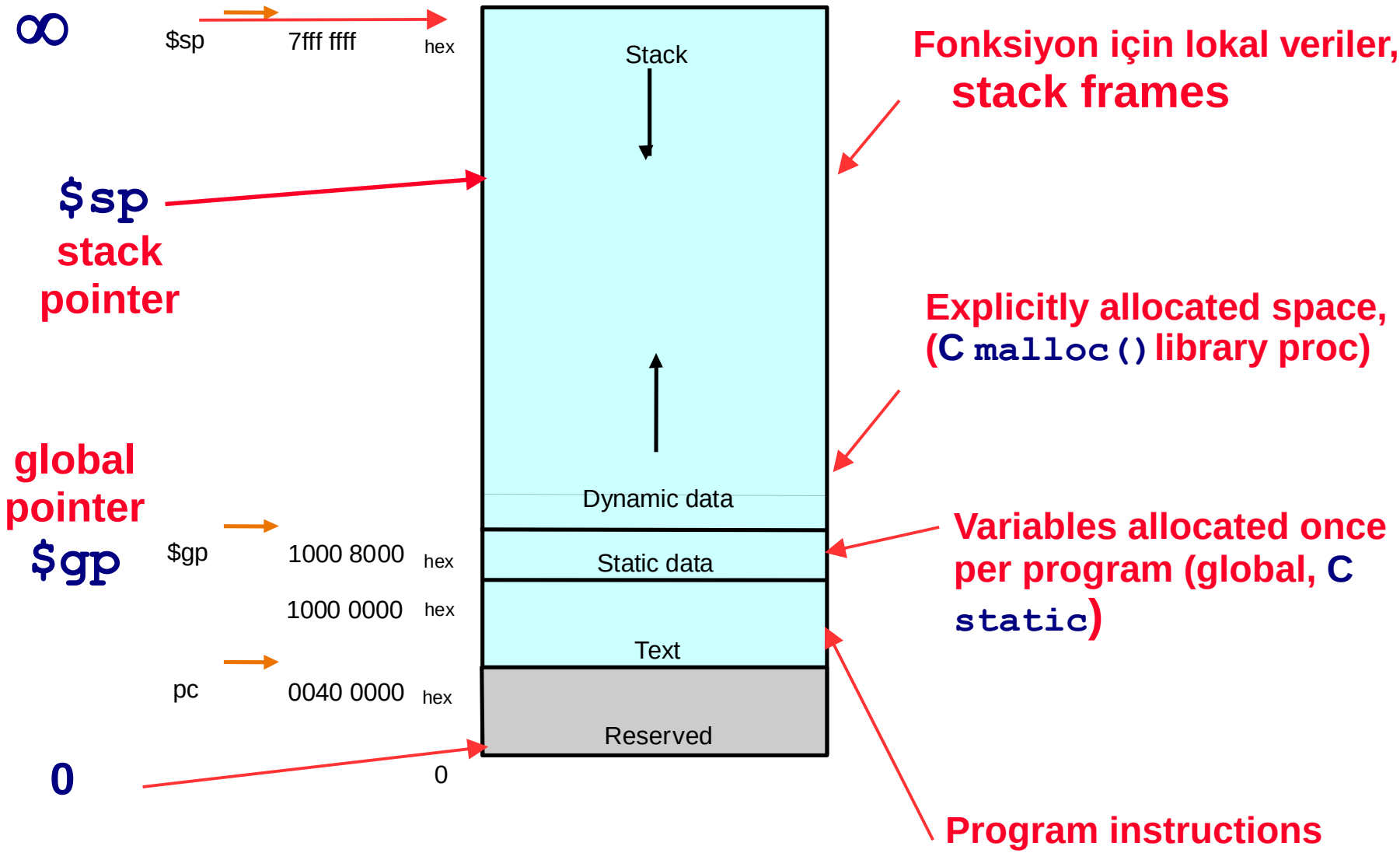


Prosedürün çağrılması ile Stack yerleşim, (a) önce, (b) süreç boyunca, (c) sonra

•$fp Frame'in 1.kelimesini gösterir.
•$sp ; stac'in en üstünü gösterir.
•$sp programın çalışması boyunca değişebilir.
•$fp sabit kalır, böylece $fp kullanılarak referans veri adresi tespit edilir.
•$fp kullanıldığında, $sp başlangıç noktasına bu sayede getirilir.

- Global pointer: Hafıza alanına kaydedilen global değişkenlerin (ana program ve prosedürler tarafından kullanılabilen) kaydedildiği alanı gösterir. Bu değer $gp registerinde tutulur.
- Dinamik olarak tahsis edilen hafıza (C'de malloc() fonksiyonu ile ayrılır) **heap** alanıdır.

Yürütülebilir programlarda gerçek zamanda hafıza-Organizasyonu



Name	Register number	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$v0–$v1	2–3	values for results and expression evaluation	no
$a0–$a3	4–7	arguments	no
$t0–$t7	8–15	temporaries	no
$s0–$s7	16–23	saved	yes
$t8–$t9	24–25	more temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	three register operands
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Data from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Data from register to memory
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $$s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,L`	if (`$s1 == $s2`) go to `L`	Equal test and branch
	branch on not equal	`bne $s1,$s2,L`	if (`$s1 != $s2`) go to `L`	Not equal test and branch
	set on less than	`slt $s1,$s2,$s3`	if (`$s2 < $s3`) `$s1 = 1`; else `$s1 = 0`	Compare less than; used with `beq`, `bne`
	set on less than immediate	`slt $s1,$s2,100`	if (`$s2 < 100`) `$s1 = 1`; else `$s1 = 0`	Compare less than immediate; used with `beq`, `bne`
Unconditional jump	jump	`j L`	`go to L`	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For procedure return
	jump and link	`jal L`	`$ra = PC + 4`; go to `L`	For procedure call

MIPS machine language

Name	Format	Example						Comments
add	R	0	18	19	17	0	32	add $s1,$s2,$s3
sub	R	0	18	19	17	0	34	sub $s1,$s2,$s3
lw	I	35	18	17	100			lw $s1,100($s2)
sw	I	43	18	17	100			sw $s1,100($s2)
and	R	0	18	19	17	0	36	and $s1,$s2,$s3
or	R	0	18	19	17	0	37	or $s1,$s2,$s3
nor	R	0	18	19	17	0	39	nor $s1,$s2,$s3
andi	I	12	18	17	100			andi $s1,$s2,100
ori	I	13	18	17	100			ori $s1,$s2,100
sll	R	0	0	18	17	10	0	sll $s1,$s2,10
srl	R	0	0	18	17	10	2	srl $s1,$s2,10
beq	I	4	17	18	25			beq $s1,$s2,100
bne	I	5	17	18	25			bne $s1,$s2,100
slt	R	0	18	19	17	0	42	slt $s1,$s2,$s3
j	J	2	2500					j 10000 (see Section 2.9)
jr	R	0	31	0	0	0	8	jr $ra
jal	J	3	2500					jal 10000 (see Section 2.9)
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	I	op	rs	rt	address			Data transfer, branch format

Karakterlerle çalışma için temel komutlar

•Çoğu bilgisayarlar, text prosesleri için karakter gösteriminde 8-bitlik kelime kullanır (ASCII kodlamada olduğu gibi) .

•Bazı komutlar, 32 bitlik bir kelimeden 8 bitlik bir kelime elde edebilir. Bunlardan **lb** ve **sb** transfer komutları 32 bitlik kelimeler yerine 8 bitlik kelimeleri transfer edebilir.

•MIPS bunu desteklemelidir. Çünkü bazı programlar 8 bit çalışmak içindir.

• lb (load byte) Komutu, ana hafızadaki 8 bitlik veriyi registerin en sağdaki 8 bitine yükler.

•Sb(Store byte) komutu ise bunun tersini yapar.

- lb $t0, 0($sp)
- sb $t0, 0($sp)

Örnek

String (dizi-karakter topluluğu) karakterlerden oluşur. Örneğin C'de tanımlı *"Cal"* stringi 4 byte ile gösterilir (67 (C), 97 (a),108 (l),0(null) olur.

Aşağıdaki kodu assemlere dönüştürün.

```
void strcpy (char x[], char y[])
{
   int i;
   i=0;
   while  ((x[i] = y[i]) != `\0')
   i + = 1;
}
```

<u>Varsayım:</u> x ve y dizilerinin başlangıç adresleri $a0 ve $a1'de olsun. i için $so reg'i kullanılsın. $so reg'i muhakkak stakc'te yedeklenmelidir.

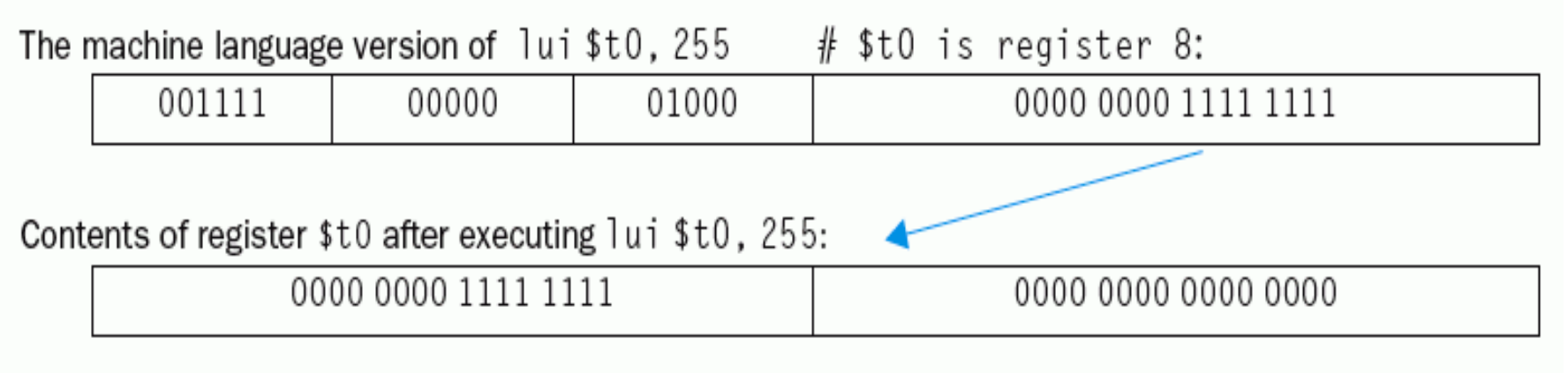
```
strcpy:
      addi   $sp, $sp, -4
      sw     $s0, 0($sp)
      add    $s0, $zero, $zero
L1:   add    $t1, $s0, $a1
      lb     $t2, 0($t1)
      add    $t3, $s0, $a0
      sb     $t2, 0($t3)
      beq    $t2, $zero, L2
      addi   $s0, $s0, 1
      j      L1
L2:   lw     $s0, 0($sp)
      addi   $sp, $sp, 4
      jr     $ra
```

- MIPS komut seti half-word (16 bitlik) yükleme ve depolama (transfer) işlemlerini de destekler.
- lh (load half) : hafızadan bir half word'ü registerin en ağırlıksız 16 bitine yükleme işini yapar.
- Sh(Store half) : Bu komut ise tersi işlemi yapar.

```
lh $t0,0($sp)  # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

32 bit immediate operandlar

- Immediate komutlar ile sadece 16-bit sabitler belirtilebilir.
- lui (load upper immediate) komutu: 16 bitlik sabiti, registerin upper register (en ağırlıklı 16 bit) alanına kaydetmek için kullanılır.



Örnek: 32 bitlik sabit sayı oluşturma

32 bitlik sabit tanımlamak için iki immediate komut kullanılır.

What is the MIPS assembly code to load this 32-bit constant into register $s0?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register $s0 afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to add the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

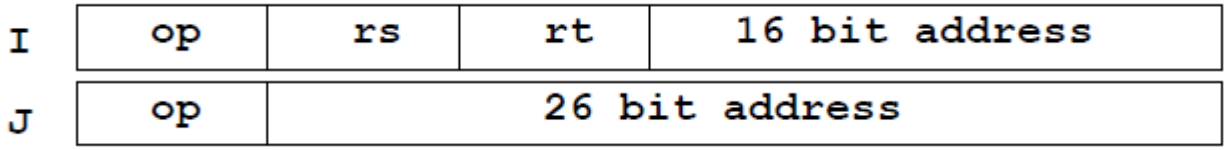
The final value in register $s0 is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

MIPS'te , assemler $at registerinde saklanmış değeri uzun sabit değerlerini veya adres değerlerini oluşturmaktan sorumludur.

Dallanma ve atlamada adreslemenin önemi

```
bne $t4,$t5,Label   $t4 ≠ $t5 ise yeni komut LABEL' etiketinin gösterdiği adrestir.
beq $t4,$t5,Label   $t4 = $t5 ise yeni komut LABEL' etiketinin gösterdiği adrestir.
j Label             yeni komut LABEL' etiketinin gösterdiği adrestir.
```



- Şartlı dallanmada, dallanılacak adres 16 bitlik sabit bir değerdir. Dallanılacak adres 16 bitden daha büyük ise ne olacaktır?
- Aynı şekilde şartsız dallanmada (atlama) da dallanılacak adres 26 bit'tir. Daha büyük bir değerde ne olacaktır?

1- Dallanma adresi, 32bitlik bir reg'in içeriği ile toplanıp PC'ye yazılır . Buna relative adresleme denir. Ama hangi registeri kullanalım?

PC = register + dallanma komutundaki adres

Şartlı dallanmalarda genellikle yakın komutlara dallanıldığından (İstatistiksel elde edilmiştir) , mevcut komutun adresinin tutulduğu PC reg'in kullanıma çok uygundur. Buna PC-Relativ adresleme denir.

2- PC'nin içeriği ile dallanma komutundaki adres toplanıp PC'ye yazılır. Buna PC-Relative adresleme denir. Genellikle bu kullanılır .

PC = PC + dallanma komutundaki adres

Böylece biz mevcut komuttan $\pm 2^{15}$ word adresi kadar uzağa dallanabiliriz.

- Şartlı dallanmada, belirtilen 16 bitlik sabit değer, hafızadaki word (32 bit – 4 byte) adresleridir. Oysa ana hafızada 1 baytlık adresleme söz konusuydu. Bu durumda dallanılacak doğru adresi bulmak için 16 bitlik sabit değeri 4 ile çarpıp PC + 4'e eklemek gerekir.

  PC = (PC+4) + (dallanma komutundaki adres*4)

Örnek:

While (save[i] == k)

i + 1 =1 ;

i ve k değişkenleri $s3 ve $s5 reg'lerinde save[] başlangıç adresi ise $s6 reg'inde olsun.

```
Loop:   sll $t1, $s3, 2      # $t1 = 4*I
        add $t1, $t1, $s6    # $t1 = address of save[i]
        lw  $t0, 0($t1)      # $t0 = save[i]
        bne $t0, $s5, Exit   # go to Exit if save[i] ≠ k
        addi $s3, $s3, 1     # i++
        j Loop               # loop again
Exit:                        # out of the loop
```

Assembled instructions and their addresses

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

bne $t0, $s5, Exit komutu için PC-relative adresleme;

- PC = (PC+4) + (dallanma adresi*4)
- → **PC = (80012+4) + (2*4) = 80024 (Exit etiketinin adresi)**
- Not: 80012, bne komutunun adresidir. Sonraki komutun adresi 80012 + 4 dür. bne komutundaki dallanılacak exit etiketinin adresi 2 olarak komutta verilmiştir. Bu değer word olarak adres değeridir.
- Oysa biz 32bit çalıştığımızdan bu değer 4 ile çarpılıp word'e dönüştürülerek dallanılacak gerçek adres elde edilir. (80024 adresi)

Assembled instructions and their addresses

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

j Loop (Atlama komutu) için Pseudodirect adresleme

Şartsız dallanma komutundaki 26 bitlik alan word (32 bitlik) olarak atlama adresini belirtir. Bunu byte olarak ifade etmek için 4 ile çarpmak gerekir.

Dolayısıyla bu alan PC'de 28 bit olarak ortaya çıkar. Niye?

PC = (PC'nin en ağ.4 biti) +(jump addresi*4)

$\longrightarrow$PC = $(0000)_2$+ $(20000*4)_{10}$= 80000_{10}(Loop'un adresi)

Uzaklara Dallanma (Branching Away)

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

L1 etiketinin çok uzak bir adres olduğunu varsayın. Bu durumda iki komut kullanılarak çok uzak adreslere dallanma yapılabilir.

```
        bne  $s0, $s1, L2
        j       L1    # uzaklara dallanabilir. Niye?
L2:
```

MIPS'de Adresleme modları

1- Register adresleme : Burada operand register'dır.

2- Base veya yerdeğiştirme adresleme: Burada operand ana hafızadaki lokasyondur.Ki onun adresi; komutaki sabit ile registerin toplamından elde edilir.

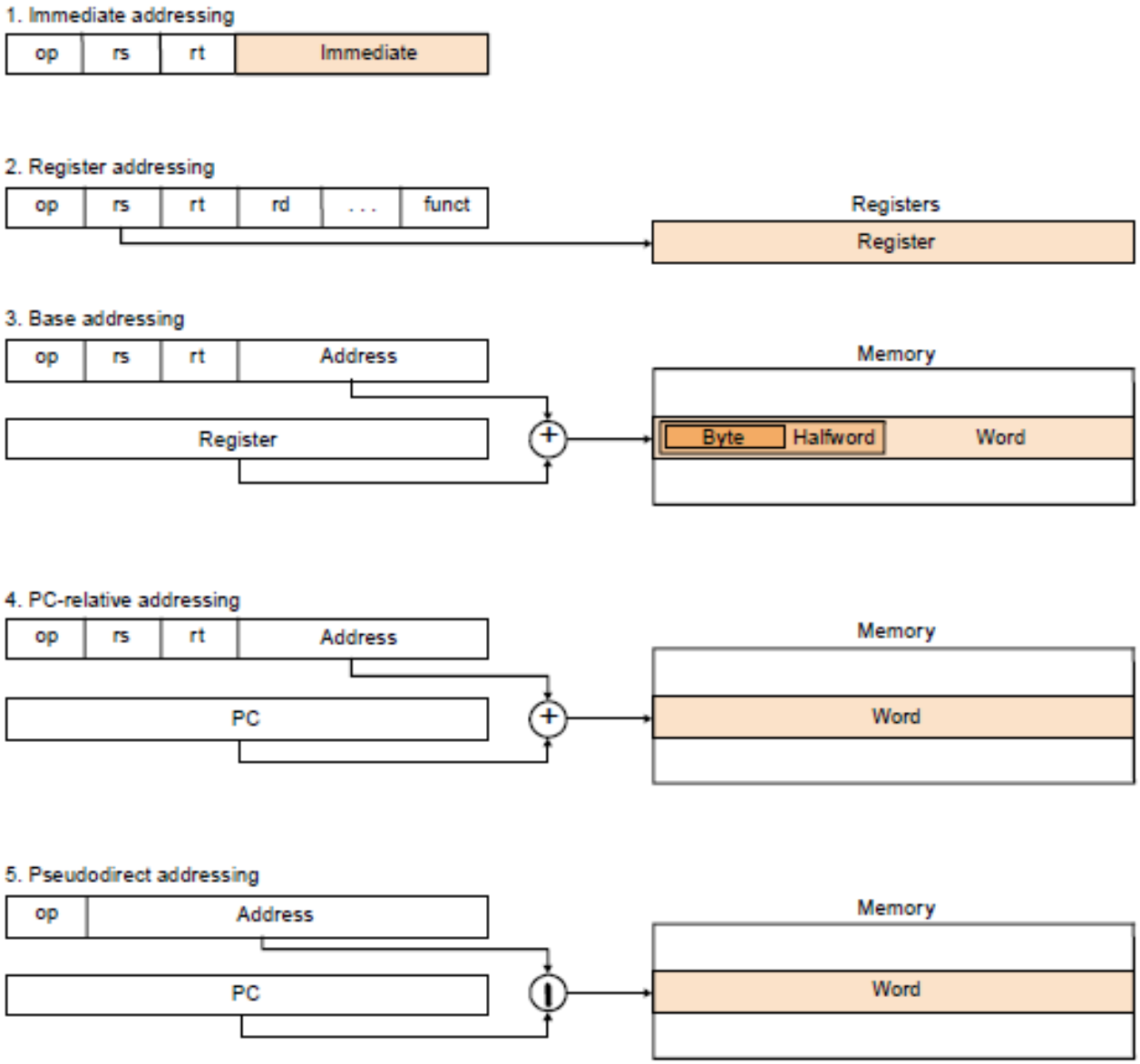
3- İmmediate adresleme: Buradaki operand komutun kendisindeki sabittir.

4-PC-Relative adresleme: Burada adres, komuttaki sabit ile PC'deki değerin toplanmasıyla elde edilir.

5- Pseudodirect adresleme: Buradaki atlama adresi PC ağırlıklı bitleri ile birleştirilmiş komutunun 26 bit'idir.

Bunları şematik olarak gösterebiliriz.

Adresleme modları şeması



Makine dilinin dekodlanması

•Makine dilinden ,tersine giderek orijinal assemler komutlarnı oluşturmaktır.

•Örnek:

00af8020 hex

Makine dilindeki komutun assemler dili karşılığı nedir?

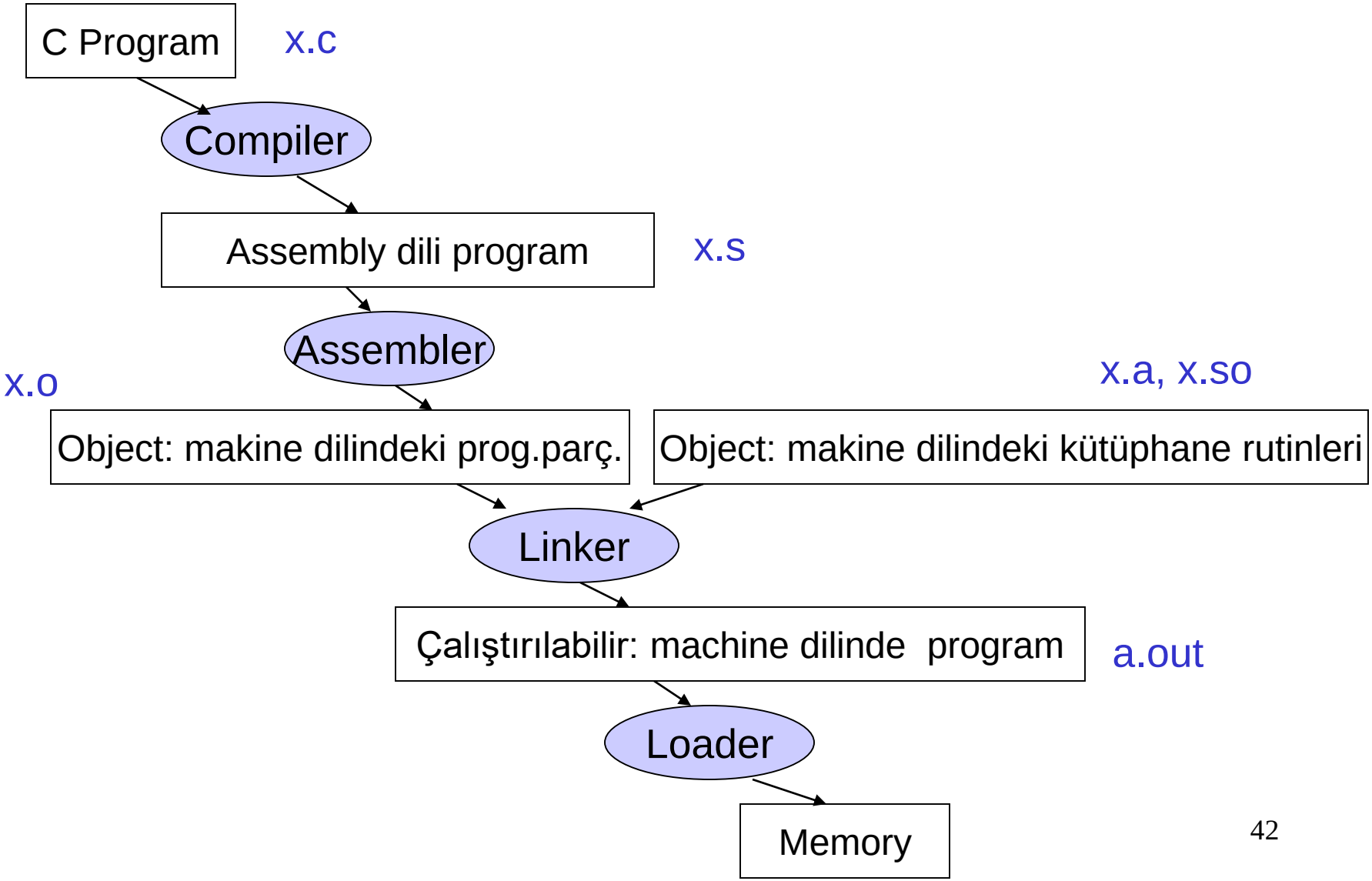
Çözüm:

```
(Bits:    31  28 26                                  5   2   0)
          0000 0000 1010 1111 1000 0000 0010 0000
```

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

```
add $s0,$a1,$t7
```

Bir Programın Başlatılması



Assembler'in Rolü

- Pseudo(sözde) kodları gerçek donanım komutlarına dönüştürmek.
  – Pseudo-kodlar assembly programa kolayca dönüştürür
  örnekler: "move", "blt", 32-bit immediate operandlar gibi.

- Assembly kodlar machine kodlarına dönüştürülür.
Ayrı object file (x.o) her bir C dosyası için oluşturulur
(x.c) komut etiketleri için gerçek değerleri hesaplar.
Dış referanslar ve hata ayıklama bilgileri hakkında bilgi sağlamak için

Linker’in Rolü

- patch iç ve dış kaynaklar
- veri ve komut etiketlerinin adreslerini hesapla
- hafızada veri ve kod modüllerini organize et.

• Bazı kütüphaneler(DLLs) dinamik olarak bağlanır(linked) Dummy rutinleri yürütülebilir nokatalara– Bu dummy rutinleri dinamik linker ve yükleme cağırır. Bu yüzden onlar doğru rutinlere atlama için yürütülebilir güncellenebilir.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,100`	$s1 = $s2 + 100	Used to add constants
Data transfer	load word	`lw $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Word from memory to regis
	store word	`sw $s1,100($s2)`	Memory[$s2 + 100] = $s1	Word from register to mem
	load half	`lh $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[$s2 + 100] = $s1	Halfword register to memory
	load byte	`lb $s1,100($s2)`	$s1 = Memory[$s2 + 100]	Byte from memory to registe
	store byte	`sb $s1,100($s2)`	Memory[$s2 + 100] = $s1	Byte from register to memor
	load upper immed.	`lui $s1,100`	$s1 = 100 * 2^{16}	Loads constant in upper 16
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-b
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-b
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 |$s3)	Three reg. operands; bit-by-b
	and immediate	`andi $s1,$s2,100`	$s1 = $s2 & 100	Bit-by-bit AND reg with const
	or immediate	`ori $s1,$s2,100`	$s1 = $s2 | 100	Bit-by-bit OR reg with constar
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branc
	branch on not equal	`bne $s1,$s2,25`	if ($s1 != $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for bec
	set less than immediate	`slti $s1,$s2,100`	if ($s2 < 100) $s1 = 1; else $s1 = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure retur
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call