Procedure

- Procedure kullanımı
  - Kodu modüler hale getirir, sonraki aşamalarda programda değişiklik yapılmasını kolaylaştırır
  - Kodun paralel olarak birden fazla kişi tarafından yazılmasına olanak sağlar
- Procedureler assembly dili seviyesinde nasıl gerçeklenir

Procedure

- Procedure çağrımı ve proceduredan dönüşün aşamaları
  - Save return address (geri dönüş adresini kaydet)
  - Procedure call (procedure çağrımı)
  - Execute procedure (procedure yürütümü)
  - Return (proceduredan dönüş)

Procedure Çağrımı

```
                .text
                .
                .
call:           jal proc
                .
                .
                done

proc:           # procedure code here
                .
                .
                jr   $31
```

procedure body

jal ve jr Komutları

- jal procedure_label
  - jal komutu, kendini izleyen komutun adresini $31'e ($ra) yazar
  - Etiketi procedure_label olan komuta dallanmayı sağlar
- jr $31
  - $31'in içeriği PC (program Counter) a yazılır ($31'in işaret ettiği komuta jump edilir (sıçranır))

Dynamic Storage Allocation (Dinamik Alan Tahsisi)

- Geri dönüş adresini bir registerda tutmak geçerli bir yöntem (eğer nested procedure çağrısı yoksa)
- Nested Procedure Call (NPC)
  - Procedure içinden procedure çağrımı
- Eğer NPC var ise
  - $31'e yazılan önceki değer üzerine tekrar yeni değer yazılır (overwrite)
  - Geri dönüş adresinin kaybolmasına sebep olur

Dinamik Alan Tahsisi

- Geri dönüş adresleri stackte (yığında) saklanır
- NPC a izin verir
- Stack dinamik olarak genişler (procedure çağrıldığında), dinamik olarak küçülür (procedure dan geri dönüldüğünde)
- Dolayısıyla stackin dinamik olarak tahsis edildiği söylenir (*dynamically allocated*)

Stack Tahsisi

- Bir çok bilgisayar, stacki programın çalışması esnasında oluşturulan environment (*runtime environment*) in bir parçası olarak gerçekler. Bu stacke *system stack* denir.
- Stack çok yaygın olarak kullanıldığından bazı makineler stacke efficient (hızlı) erişim için bazı destekler sağlar
- MIPS işlemcisi
  - Stacke itme yapıldığında (push) stack küçük bellek adresine doğru genişler
  - Stackin başlangıcının (bottom) adresini tutan bir registera sahip
  - *$29* (**$sp**) : stack pointer (stackin üzerinde ilk boş olan alanın adresini tutar)

push ve pop işlemleri

push işlemi:

```
        sw    $8, 0($sp)
        add   $sp, $sp, -4
```

veya

```
        add   $sp, $sp, -4
        sw    $8, 4($sp)
```

pop işlemi:

```
        add   $sp, $sp, 4
        lw    $8, 0($sp)
```

veya

```
        lw    $8, 4($sp)
        add   $sp, $sp, 4
```

Power function (recursive olarak)

```
            .text
            .
            li      $18,1           # $18 will contain the result
            move    $19, $17        # $19 is a counter, $17 contains the power
            jal     power
            .
power:      sw      $31, 0($sp)     # save return address
            add     $sp, $sp, -4    # by pushing it on the stack
      if:   add     $19, $19, -1
            blez    $19, endif
            jal     power           # recursive procedure call
endif:      mul     $18, $18, $16   # $16 contains base
            add     $sp, $sp, 4     # restore return address by pooping it off the
                                    # stack
            lw      $31, 0($sp)
return:     jr      $31
```

Activation Records

- Activation Record (stack frame)
  - Bir procedure çağrıldığında yeni bir environment (ortam) oluşturulur. Bu ortam:
    - Local değişkenler (yerel değişkenler)
    - Hesaplamada kullanılan ve ara değerler tutan değişkenler
    - Geri dönüş adresi
  - Ortamda tutulan değerler procedurenün yaşam süresi (lifetime) boyunca korunmalı (fakat procedure sonlandığında onlar da sonlanmalı)

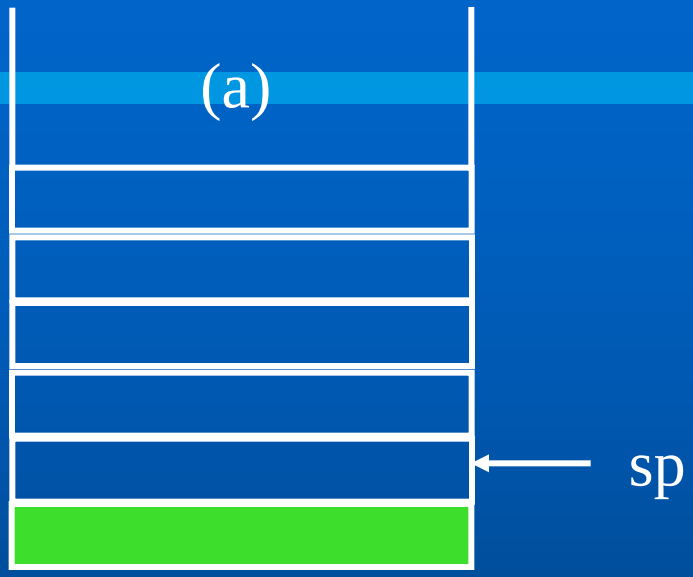
Activation Records

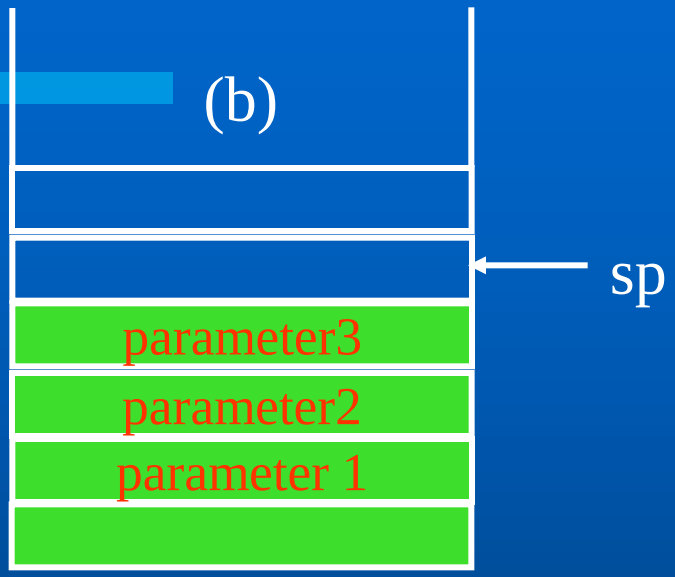
- Activation record, bir procedure hakkındaki bütün bilgileri içerir.
- NPC (nested procedure call) nin doğru olarak gerçeklenmesine yetecek kadar bilgi içerir
- Activation record, normal bir word un stacke push edilmesi ve stackten pop edilmesi gibi stack e push edilir veya stackten pop edilir
  - Tek farkı activation record un boyu word un boyundan daha büyüktür
  - Activation record un stacke push edilmesi (procedure çağrımı esnasında) ve stackten pop edilmesi (procedure dan donüşte), stack pointerinin değeri activation recordun boyuna uygun olarak değiştirilmeli.

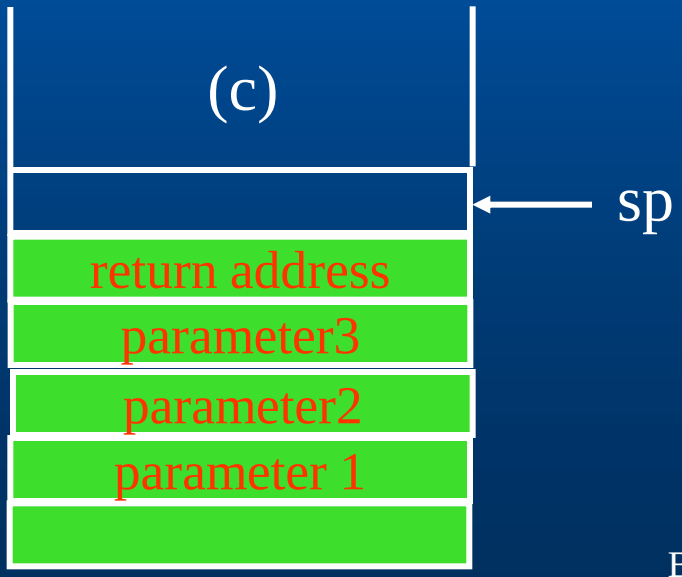
Parameter Passing (Parametre Aktarımı)

- Procedurelere alan tahsisi (activation record için) bir dizi push işlemleriyle sağlanır
  - Her bir parametrenin stacke itilmesi
  - Geri dönüş adresinin stacke itilmesi
- Calling (çağıran) procedure parametreleri stack e push eder
- Geri dönüş adresini ise called (çağrılan) procedure stack e push eder

Parametre Aktarımı







(a) Parametreleri push etmeden önce
(b)parametreler push edilmiş, fakat procedure call yapılmamış
(c) Procedure call ve dönüş adresinin saklanması

Parametre Aktarımı

parameter1->$8
parameter2->$12
parameter3->$6

```
        sw    $8, 0($sp)
        add   $sp, $sp, -4
        sw    $12, 0($sp)
        add   $sp, $sp, -4
        sw    $6, 0($sp)
        add   $sp, $sp, -4
        jal   proc

        .
        .
proc:   sw    $31, 0($sp)
        add   $sp, $sp, -4
        .
```

Eğer activation record tek varlık olarak düşünülürse

```
        add   $sp, $sp, -16
        sw    $8, 16($sp)
        sw    $12, 12($sp)
        sw    $6, 8($sp)
        jal   proc

        .
        .
proc:   sw    $31, 4($sp)
        .
```

Calling procedure içinde parametrelere erişim

- Load/store mimaride stack deki parametrelere erişim için onlar önce registerlara yüklenmeli

```
lw      $4, 16($sp)
lw      $5, 12($sp)
lw      $6, 8($sp)
```

- İkinci çözüm: parametreleri direkt olarak registerlar üzerinden aktarmak
- SPARC ve Berkeley RISC de register windows kavramı var

Proceduredan değer geri çevirme

- Proceduredan değer geri döndürmek parametre geçişine benzer. Farkı, bilgi akışı çağrılan proceduredan çağıran procedura doğru
- Nested procedurelarda değer stack üzerinden geri döndürülür
- Nested olmayan procedurelarda hız açısından değer bir registerla geri döndürülür

Registerların Kaydedilmesi

- Bir procedure, yerel değişkenler, parametreler ve geçici hesaplamalar için çok sayıda registera ihtiyaç duyabilir
- Çağıran prosedürün kullanmakta olduğu bir register, stacke itilip prosedürden dönüşte tekrar stackten yüklenmedikçe kullanılamaz
- Dolayısıyla bir register hem çağıran prosedüre ve hem de çağrılan prosedüre tarafından kullanılıyorsa, bu ilgili register activation record un bir parçası olmak zorunda (prosedüre çağrımı esnasında stacke itilmeli, ve prosedürden dönüşte tekrar orijinal değeri stackten geri alınmalı)

Registerlar Ne Zaman Kaydedilmeli?

- Çağrılan Prosedüre registerları kaydedebilir
- Çağıran Prosedüre registerları kaydedebilir

Çağrılan Prosedür Tarafından Registerların Kaydedilmesi

```
                .
                jal     procedure
                .
                .
procedure:
                sw      $31, 0($sp)         # push return address
                add     $sp, $sp, -4

                add     $sp, $sp, -12       # push register values and
                sw      $8, 12($sp)         # update stack pointer
                sw      $9, 8($sp)
                sw      $10, 4($sp)
                .
                # procedure's code here
                .

                lw      $10, 4($sp)         # restore register
                lw      $9, 8($sp)          # values and
                lw      $8, 12($sp)         # update stack pointer
                add     $sp, $sp, 12

                                            # restore return address
                lw      $31, 0($sp)
                jr      $31                 # return
```

Avantaj: Sadece çağrılan prosedüre tarafından kullanılan registerlar kaydedilir

Dezavantaj: Bazı registerlar gereksiz yere kaydedilir

Çağıran Prosedür Tarafından Registerların Kaydedilmesi

```
                .
                .
                .
                .
        add     $sp, $sp, -12       # push register values
        sw      $8, 12($sp)         # update stack pointer
        sw      $9, 8($sp)
        sw      $10, 4($sp)
        jal     procedure
        lw      $10, 4($sp)         # restore register values and
        lw      $9, 8($sp)          # update stack pointer
        lw      $8, 12($sp)
        add     $sp, $sp, 12
                .
                .

Procedure:  sw  $31, 0($sp)         # push return address
        add     $sp, $sp, -4

        # procedure's code here

        add     $sp, $sp, 4         # restore return address
        lw      $31, 0($sp)
        jr      $31                 # return
```

Avantaj: Sadece çağıran prosedürde kullanılan registerlar kaydedilir
Dezavantaj: Bazı registerlar gereksiz yere kaydedilir

MIPS RISC Register Kullanımı

- MIPS mimarisinde registerların kullanımı konusunda bir takım kurallara uyulması teşvik edilir
  - Bazı registerlar prosedür çağrıları arasında korunmalı (saved registers, $s0-$s8)
  - Saved olmayanlar geçici registerlar (temporary registers, $t0-$t9)

MIPS RISC Register Kullanımı

- Bir register tahsis edilirken bir seçim yapılmalı
  - Eğer prosedür, bir prosedür çağrısı içermezse temporary registerlar (kaydedilmeye gerek yok) kullanımı tercih edilir
  - Eğer prosedür, bir prosedür çağrısı içerirse temporary register (prosedür çağrılarında kaydetmeli) veya saved register (prosedürü çağrısı yapıldığında kaydedilmeli) arasında tercih yapılmalıdır

Register İsmi	Alternatif İsim	Kullanımı
$0		0 değeri
$1	$at	Assembler tarafından reserv edilmiş
$2 - $3	$v0 - $v1	İfade değerlendirilmesi ve fonksiyon sonuçları
$4 - $7	$a0 - $a3	İlk 4 parametre; prosedür çağrılarında korunmaz
$8 - $15	$t0 - $t7	temporary registers; prosedür çağrılarında korunmaz
$16 - $23	$s0 - $s7	Saved registers; prosedür çağrılarında korunurlar
$24 - $25	$t8 - $t9	temporary registers; prosedür çağrılarında korunmaz
$26 - $27	$k0 - $k1	İşletim sistemi için rezerv edilmiş
$28	$gp	Global işaretçi
$29	$sp	Stack işaretçisi
$30	$s8	Saved value; prosedür çağrılarında korunur
$31	$ra	Geri dönüş adresi
$f0 - $f2		Kayan noktalı prosedür sonuçları
$f4 - $f10		temporary registers; prosedür çağrılarında korunmaz
$f12 - $f14		İlk iki kayan noktalı param.; prosedür çağrı. korunmaz
$16 - $f18		temporary registers; prosedür çağrılarında korunmaz
$f20 - $f30		Saved registers; prosedür çağrılarında korunurlar

Assemblerlar Ne Yapar?

- Assembly dili programı
  - Veri belirlemeleri
  - Komutlar
- Makine Kodu: İşlemcinin anlayıp, çalıştırabileceği koda denir
  - Binary (ikili) formda
- Assembler
  - Programı çalıştırmak için belleğin başlangıç durumu ne olmalı: Bu hem veriler hem de komutlar için geçerli
  - Temelde iki görevi var
    - Assembly dili programını makine koduna dönüştürmek
    - Sembolik etiketlerin adreslerinin hesaplanması

Makrolar

- Sık sık bir grup komutlar programın değişik kesimlerinde tekrarlanır
- Örnek: veriyi stake iten veya veriyi stackten çeken komut grubu
- Programcıya bir grup komutun belirlenmesine ve bu grubu bir anahtar sözcüğü ile ilişkilendirmeye dayanan mekanizmaya makro denir.
- Preprocessor: Makroları tanımladıkları komut gruplarına dönüştürme işlemini yapan programa preprocessor denir

- SAL soyut assembly dilidir
  - Direkt ikili gösterim karşılığı yoktur
- SAL komutları çoğu zaman çoklu MAL komutlarına karşı gelir
- Her ne kadar çoğu MAL komutlarının birebir gerçek assembly dili (TAL) karşılığı olsa da, MAL bazı soyut komutlara da sahiptir.
- TAL, MIPS RISC işlemcisinin gerçek assembly dilidir
  - SAL veya MAL programı önce TAL programına dönüştürüldükten sonra makine koduna dönüştürülebilir

TAL

- TAL aritmetiksel ve mantıksal komutlar
  - 3 operandlı komutlar; Oysa MAL'da 2 veya 3 operandlı komutlar kullanılır
  - TAL'da MAL'ın aksine sabitler kullanılır

TAL İvedi Komutları

Format	Etki
addi R_t,R_s, I	$R_t \leftarrow [R_s] + ([I_{15}]^{16} \,||\, I_{15..0})$
andi R_t,R_s, I	$R_t \leftarrow 0^{16} \,||\, ([R_s]_{15..0}$ AND $I_{15..0})$
lui R_t,I	$R_t \leftarrow I_{15..0} \,||\, 0^{16}$
ori R_t,R_s, I	$R_t \leftarrow [R_s]_{31..16} \,||\, ([R_s]_{15..0}$ OR $I_{15..0})$
xori R_t,R_s, I	$R_t \leftarrow [R_s]_{31..16} \,||\, ([R_s]_{15..0}$ XOR $I_{15..0})$

TAL mult, div Komutları

- mult R_s, R_t
  - MIPS RISC mimarisi HI ve LO diye bilinen iki özel registera sahip
  - Çarpma komutunda R_s ile R_t çarpılır
  - Çıkan sonucun düşük anlamlı 32 biti LO'ya, yüksek anlamlı 32 biti HI'ya konur
- div $9,$10
  - Bölüm LO registerine, kalan HI registerine konur

- MAL
  - mult $8,$9,$10
- TAL
  - mult $9,$10
  - mflo $8
- MAL
  - div $8,$9,$10
- TAL
  - div $9,$10
  - mflo $8
- MAL
  - rem $8,$9,$10
- TAL
  - div $9,$10
  - mfhi $8

Tamsayı Çarpma ve Bölme İşlemlerine Yönelik TAL Komutları

Format	Etkisi
mult R_s,R_t	HI || LO ←[R_s] *[R_t]
div R_s,R_t	LO ← [R_s] div [R_t]; HI ← [R_s] mod [R_t]
mfhi R_d	R_d ← HI
mthi R_s	HI ← R_s
mflo R_d	R_d ← LO
mtlo R_s	LO ← R_s

Dallanma Komutları

- Tüm MAL dallanma komutları TAL'da mevcut değildir
- TAL'da mevcut dalalnma komutları
  - bltz, bgez, blez, bgtz, beq, bne
- Diğer MAL komutları ya ters karşılaştırma, veya değer çıkarılarak sonucun sıfırla karşılaştırılması şekline dönüştürülür
  - blt $11,$12,repeat_loop MAL komutunun TAL karşılığı iki komuta karşı gelip, bunlar:
  - sub $1, $11,$12
  - bltz $1,repeat_loop

la Komutu

- MAL la (load address) komutuna sahip, fakat TAL'da la komutu yok
  - MAL komutu
    - la R, label
  - TAL komutları olarak
    - lui R, high_address //load upper immediate
    - ori R,R, low_address //or immediate

Flag = 0x00030248

I/O Komutları

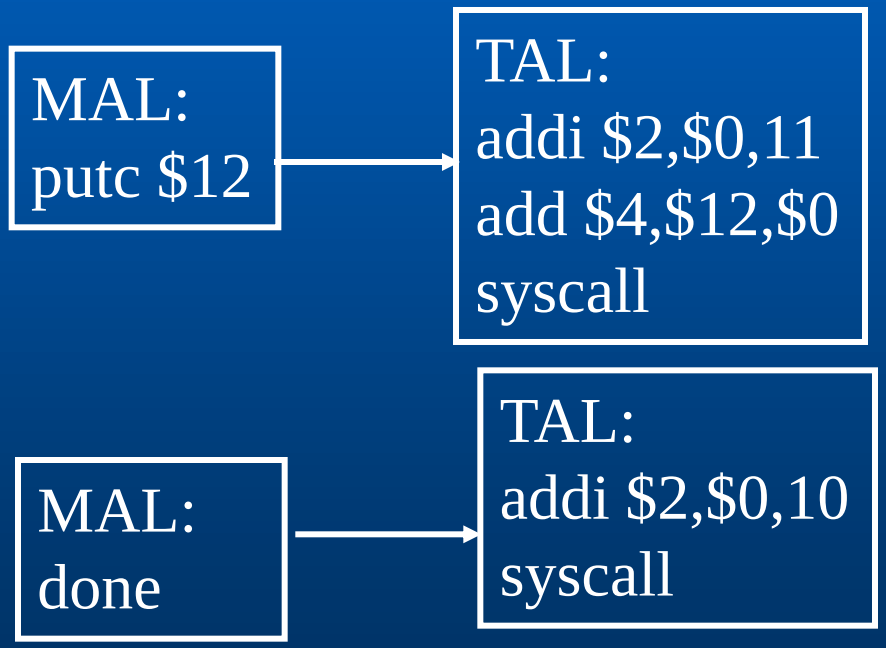
- putc ve getc MAL komutları TAL'da mevcut değildir.
- I/O komutlarının gerçekleştirilmesi görevi OS (işletim sistemine) verilmiştir
- syscall (sistem çağrısı) kullanımı ile bu komutlar gerçekleştirilir
  - Hangi görevi gerçekleştirileceği bir parametre ile belirlenir
  - $2 registeri hangi fonksiyonun gerçekleştireceğini belirler

I/O Komutları

Fonksiyon	$2 değeri
put	1
puts	4
get	5
done	10
putc	11
getc	12

I/O Komutları

- MAL putc
  - Syscall ekrana gönderilecek karakterin $4 registerinde olmasını bekler



Her Zaman İşimiz Kolay Olmayabilir

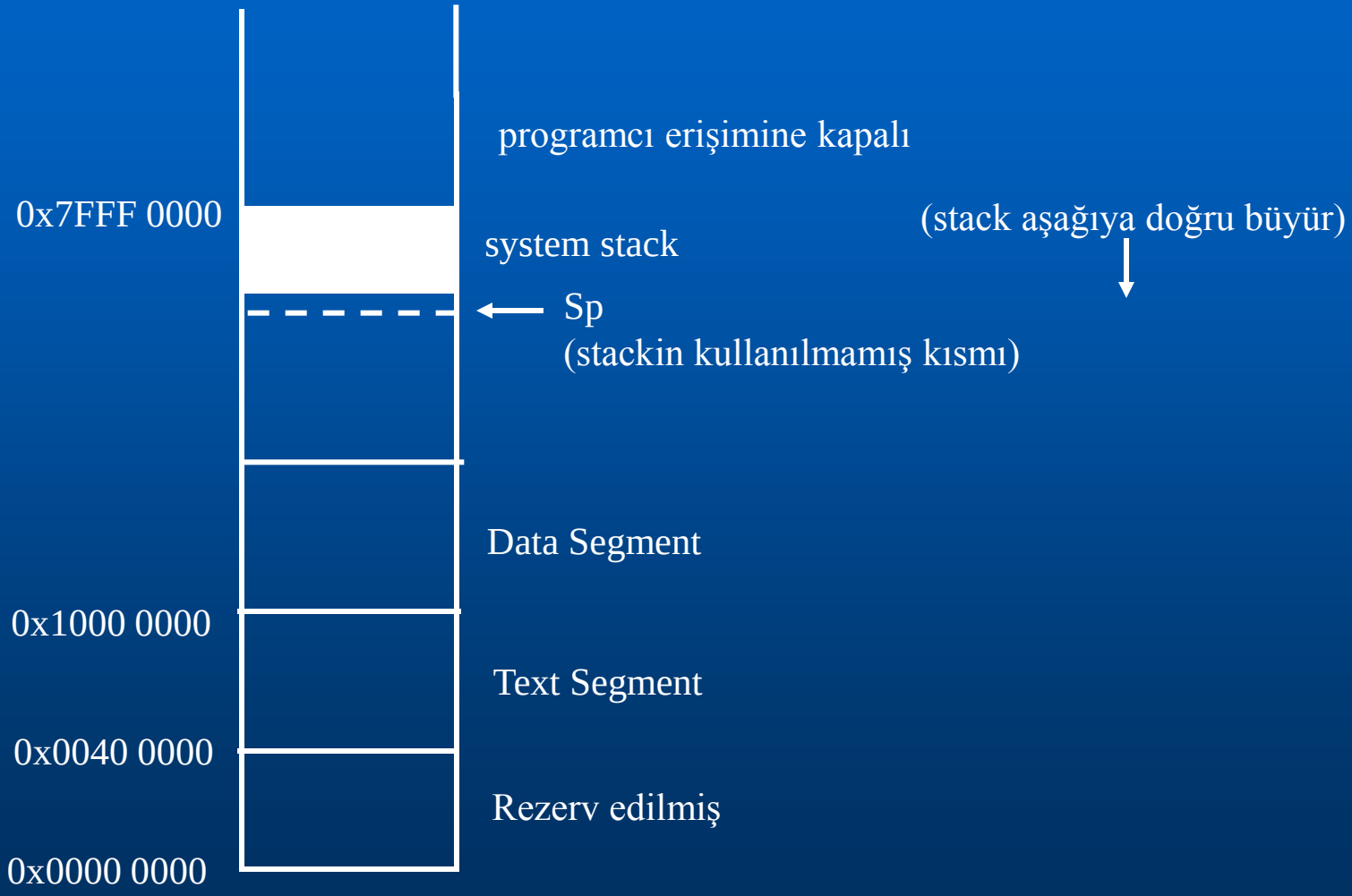
Taşma varsa: $11 = 4,000,000,000
$12 = -4,000,000,000

MAL:
```
blt $11,$12,repeat_loop
```
→
TAL:
```
sub $1, $11,$12
bltz  $1, repeat_loop
```
Hatalı olur

MAL:
```
blt $11,$12,repeat_loop
```
→
TAL:
```
subu $1, $11,$12
bltz  $1, repeat_loop
```
Hatasız

MAL:
```
blt $11,$12,repeat_loop
```
→
TAL:
```
slt $1, $11,$12 // set less than
bne  $1, $0, repeat_loop
```
Hatasız

MIPS RISC İşlemcisi için Bellek Tahsisi



Makine Kodu Üretimi

- Assembler belleğin görüntüsünü tanımlamalı (image of memory)
  - Programın yürütümü için bellekte yer alacak başlangıç değerlerini atamak için veri yapıları oluşturur
  - İlgili program için belleğin başlangıç değerlerini temsil etmek için iki dizi kullanılır
    - Biri veriler için Data Segment, diğeri komutlar için Text Segment
    - Her bir dizi elemanı, ilgili segmentin başlangıcından ilgili uzaklıktaki bir bellek elemanına karşı gelir
  - Görüntü oluşturulduktan sonra, görüntüyü tanımlayan diziler ilgili bilgilerle birlikte bir dosyaya yerleştirilir
  - Bu dosyayı loader programı kullanarak program yürütümü başlangıcında belleğin uygun başlangıç durumuna getirilmesini sağlar

Branch Offset Hesabı

- TAL, bir branch komutunda hedef adresi belirlemek için bir etikete gereksinim duyar
- Makine kodunda hedef adres kesin adres değil de branch komutunun adresine bağıl olmalı (PC-relative)
- Branch komutları için hedef adrese yönelik offset hesabı aşağıdaki formülle yapılır
  - Offset = hedef komutun adresi – (branch komutu adresi +4)
  - PC, taken brancha ait offset PC'ye eklenmeden önce güncellendiğinden (4 artırıldığından), bu formüle 4 eklenmiştir

Branch Offsetleri

- Üç önemli husus
  - Offset komuta byte offsetten ziyade word offset olarak yerleştirilir. Bu mümkün çünkü komutlar word-aligned olarak belleğe yerleştirilirler
    - Komut adreslerinin son iki bitleri 0'dır.
  - Offset negatif olabilir. Eğer hedef adres branch komutundan daha önceki bir komuta aitse, offset negatif olacaktır
  - Branch komutlarında offsetlerin yerleştirilmesi için 16 bitlik bir alan ayrılır.
    - Dallanma aralığı
      - Branch komutu adresi ± 2^{15} komut
      - 2^{15} komut aralığı, 2^{17} byte aralığı yapar

Branch Offsetleri

- PC-relative adreslemenin yararı
  - Hedef adresi belirlemek için sadece 16 bit alan branch komututlarında mevcutken, PC-relative adresleme ile hemen hemen tüm branch komutlarının üstesinden gelebiliriz
  - Nadir de olsa 2^{15} komut aralığının dışına çıkmak istersek:

```
bgtz    $5, long_jump
```

```
        blez    $5, continue
        j       long_jump
continue:
```

Jump Hedef Hesabı

- 32-bit adresleri 32 bit uzunluklu komutlara gömemeyiz
- jump komutunun iki değişik şekli mevcut
  - j komutu: adres komutun içine gömülü
  - jr (jump register) komutu: hedef adresi tutan bir register komut içerisinde kodlanır
  - j komutu, 6-bit opcode alana sahip, dolayısıyla adres 26 bitlik alana gömülmeli
    - Komutlar word-alignment olduklarından komut adreslerinin son iki bitleri daima 0'dır
    - j komutu kısıtlaması: jump sonucu PC'nin en anlamlı 4 biti değişmemeli
    - Aşağı yukarı 67 milyon aralığındaki komutlar j komutunun hedefi olabilir
    - Eğer bu aralığın dışındaki bir yere jump edilmek istenilirse jr komutu kullanılabilir.
  - Hedef adresin en anlamlı 4 biti ve en az anlamlı iki biti atılarak elde edilen 26 bitlik değer j komutuna gömülerek jump edilecek adres belirlenir.