Assembler Dersi #3

Bu Assembler üzerine son derstir. Branch(dallanma), recursion vs. gibi konuların üzerinden geçeceğiz.

Branch(Dallanma) Talimatları

İf, for ve while yapıları için "karşılaştırma" ve "dallanma" talimatları vardır.

```
cmp %r0, %r1            Bu deyim; r0 ve r1 regester değerlerini
                        karşılaştırmak için kullanılır ve sonucu yansıtacak
                        kontrol durumu regesterını(CSR) ayarlar.
                        CSR (r0 == r1), (r0 < r1), (r0 > r1) olup
                        olmadığını saklayacaktır.

b l1                    Bu deyim; l1 etiketine doğrudan gitmek(dallanmak)
                        içindir. Bu durum PC l1 olana kadar sürer(PC+4).
                        Unutmayın ki bir "jsr" deyimi yapabildiğiniz gibi
                        "return" şeklinde bir dallanma yapamazsınız.

beq l1                  Bu deyim; Eğer CSR bu İki karşılaştırılmış
                        değerin eşit olduğunu gösteriyorsa l1 etiketine
                        gider(PC bu değere ayarlanır). Eğer bu iki
                        karşılaştırılmış değer eşit değil ise sonraki
                        ifade(PC+4) çalıştırılır.

ble l1                  (<=, <, >=, >, !=) şeklindedir.

blt l1

bge l1

bgt l1

bgt l1

Bu ölçüde; if, for, while gibi koşullu ifadeler açıktır ve nettir.
```

if (**cond**) {
  S1
} else {
  S2
}
S3
	koşullu tanımlama
	l1 için koşullu bir olumsuzlukta üzerine dallanma
	  S1
	  b l2
	l1:
	  S2
	l2:
	  S3

Örneğin;

```
int a(int i, int j)
{
  int k;

  if (i < j) {
    k = i;
```

```
  } else {
    k = j;
  }
  return k;
}
```

```
a:
   push #4
   ld [fp+12] -> %r0          / r0 içine i yükle
   ld [fp+16] -> %r1          / r1 içine j yükle

   cmp %r0, %r1               / olumsuz durumda karşılaştır ve dallan)
                               /(büyük veya eşit)

   bge l1

   ld [fp+12] -> %r0          / k = i
   st %r0 -> [fp]
   b l2

l1:
   ld [fp+16] -> %r0          / k = j
   st %r0 -> [fp]
l2:

   ld [fp] -> %r0             / return k
   ret
```

while (**cond**) {
  S1
}
S2
	l1:
	  koşullu tanımlama
	  l2 için koşullu bir olumsuzlukta üzerine dallanma
	  S1
	  b l1
	l2:
	  S2

for (**S1**; **cond**; **S2**) {
  S3
}
S4
	S1
	b l2
	l1:
	S2
	l2:
	koşullu tanımlama
	l3 için koşullu bir olumsuzlukta üzerine dallanma
	S3
	b l1
	l3:
	S4

Örneğin;

```
int a(int k)
{
  int i, j;

  j = 0;

  for (i = 0; i < k; i++) j += i;

  return j;
}
```

Derlendiğinde;

```
  push #8                       / i ve j için stack tahsisi

  st %g0 -> [fp-4]              / sıfıra j değerini ata

  st %g0 -> [fp]                / döngü başlat (S1)

  b l2


l1:

  ld [fp] -> %r0                / i++ (S2)
  add %r0, %g1 -> %r0
  st %r0 -> [fp]

l2:
  ld [fp] -> %r0                / Test et ve olumsuzlukta dallan.
  ld [fp+12] -> %r1
  cmp %r0, %r1
  bge l3

  ld [fp-4] -> %r0              / j += i  (S3)
  ld [fp] -> %r1
  add %r0, %r1 -> %r0
  st %r0 -> [fp-4]
  b l1

l3:
  ld [fp-4] -> %r0              / return j (S4)
  ret
```

Her zaman olduğu gibi bu kod büyük ölçüde optimize edilebilir.

Recursion(Özyineleme)

Şimdiye kadar recursif yordamlar bize yabancı değillerdi. Örneğin;

```
int fact(int i)
{
  if (i == 0) return 1;
  return fact(i-1)*i;
}
```

Derlendiğinde;

```
fact:
        ld [fp+12] -> %r0           / if deyimini gerçekleştir
        cmp %r0, %g0
        bne l1

        mov %g1 -> %r0
        ret

l1:
        ld [fp+12] -> %r0           / stack'e i-1 yerleştir
        add %r0, %gm1 -> %r0
        st %r0 -> [sp]--

        jsr fact                    / fact' a atla
```

```
        pop #4                      / stackten argüman çıkar

        ld [fp+12] -> %r1           / fact(i-1)*i türet
        mul %r0, %r1 -> %r0
        ret
```

Her özyinelemeli çağrı yeni bir stack çerçevesi ittirir. **fact(4)** üzerinden izlemek için **jassem.tcl** kullanabilirsiniz.

Bir Örnek daha;

Burada ayrıntılı olarak üzerine gitmeyeceğiz. Ama **bsort.c** sayfasına bakınız. Aşağıda 4 elemanlı bir dizinin bubble sort ile basit bir sıralaması bulunuyor:

```
void bsort(int *a, int size)
{
  int i, j, tmp;

  for (i = size-1; i > 0; i--) {
    for (j = 0; j < i; j++) {
      if (a[j] > a[j+1]) {
        tmp = a[j];
        a[j] = a[j+1];
        a[j+1] = tmp;
      }
    }
  }
}

main()
{
  int array[4];
  array[0] = 6;
  array[1] = 1;
  array[2] = 4;
  array[3] = 2;

  bsort(array, 4);
}
```

Burada birçok dizi işlemi var dolayısıyla assembly kodu uzun olacak. **bsort.jas** aşağıdadır:

```
bsort:
   push #12                 / i=fp-8, j=fp-4, tmp=fp
   st %r2 -> [sp]--         / r2 azalt

                            / döngü #1: etiketler f11, f12, f13
   ld [fp+16] -> %r0        / i = size-1
   add %r0, %gm1 -> %r0
   st %r0 -> [fp-8]
   b f12

f11:
   ld [fp-8] -> %r0         / i--
   add %r0, %gm1 -> %r0
   st %r0 -> [fp-8]

f12:
   ld [fp-8] -> %r0         / i > 0
   cmp %r0, %g0
```

```
   ble f13

                            / döngü #2: etiketler f21, f22, f23
   st %g0 -> [fp-4]         / j = 0
   b f22

f21:
   ld [fp-4] -> %r0         / j++
   add %r0, %g1 -> %r0
   st %r0 -> [fp-4]

f22:
   ld [fp-4] -> %r0
   ld [fp-8] -> %r1
   cmp %r0, %r1
   bge f23

                            / eğer (a[j] > a[j+1])

   ld [fp-4] -> %r0         / önce a[j] yi r0 regesterının içine koy
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0

   ld [fp-4] -> %r1         / a[j+1] yi r1 regesterının içine koy
   add %r1, %g1 -> %r1      / r0 a dokunmadan
   mov #4 -> %r2
   mul %r1, %r2 -> %r1
   ld [fp+12] -> %r2
   add %r1, %r2 -> %r1
   ld [r1] -> %r1

   cmp %r0, %r1
   ble i1

   ld [fp-4] -> %r0         / tmp = a[j]
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

   ld [fp-4] -> %r0         / a[j] = a[j+1]
   add %r0, %g1 -> %r0      / r0 içine a[j+1] yi yükle
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   ld [fp-4] -> %r1         / r1 içine &(a[j]) yi yükle
   mov #4 -> %r2
   mul %r1, %r2 -> %r1
   ld [fp+12] -> %r2
   add %r1, %r2 -> %r1
   st %r0 -> [r1]           / a[j] içine r0 ı hafızaya al.

   ld [fp] -> %r0           / a[j+1]  = tmp
   ld [fp-4] -> %r1
   add %r1, %g1 -> %r1
   mov #4 -> %r2
   mul %r1, %r2 -> %r1
   ld [fp+12] -> %r2
   add %r1, %r2 -> %r1
```

```
   st %r0 -> [r1]

i1:                          / if ifadesinin sonu

   b f21                     / #2 numaralı döngünün sonu
f23:

   b f11                     / #1 numaralı döngünün sonu
f13:
   ld ++[sp] -> %r2
   ret

main:
   push #16

   mov #-1 -> %r2          / Bu sadece boşalmasını göstermektedir

   mov #6 -> %r0
   st %r0 -> [fp-12]
   mov #1 -> %r0
   st %r0 -> [fp-8]
   mov #4 -> %r0
   st %r0 -> [fp-4]
   mov #2 -> %r0
   st %r0 -> [fp]

   mov #4 -> %r0
   st %r0 -> [sp]--
   mov #12 -> %r0
   sub %fp, %r0 -> %r0
   st %r0 -> [sp]--
   jsr bsort
   pop #8

   ret
```

Jas ile yürütme biraz ağır olur. Linuxte hızlı olabilir fakat Windowsta öyle gözükmüyor.

Dünyada tcl/tk kodlamada en verimli olanı bu değildir. Unutmayın ki assembly kod çevirisini ve assembly işleyişini anlamak gerekir. Bu kod büyük ölçüde verimsizdir ve bazı akıllıca kullanımlar ile daha hızlı yapılabilir.

Delay Slots(Gecikme Yuvaları)

Herhangi bir makineden assembly okunması zor olabilir. Fakat assembly haritalarının nasıl çalıştığını tek bir tanımlanmış sınıfla anlayabilirsiniz. Sparc işlemciler için muhtemelen özgün gecikme yuvalarının bir noktasında karışıklık var. “Pipelinning” işlemcileri hızlandırmak için bir teknik vardır. Yani Bir sonraki komut yürütmeye başlamadan önce CPU yürütme bitirinceye kadar mevcut talimatı bitirmez. Genellikle bu kadar karmaşa içermez. Ancak jsr, ret ve b talimatlarında bir sorun vardır:

Bu talimatlar PC değiştirmek gibi, şöyle ki bir sonraki talimat yürütülür olmamalıdır. Ama işlemcinin bir hattı üzerinde zamana göre talimat yapılır.

Sparc üzerine çözümümüz jsr, ret ve b talimatları yürütüldükten sonra after talimatı ile değiştirilen PC değeri gider. Bu talimattan bir sonraki jsr, ret veya b gelmesine gecikme yuvası(delay slot) denir.

Jsr semantiğinin değişmesi gerektiğini de unutmayalım. Öyle ki ret çağırıldığında yığına PC+8 atılması şarttır. After talimatı ile gecikme talimatı verilir.

Bu yuvaya doğru kullanıldığından emin olmak için compiler da yazarken garantiye almak gerekir. Örneğin derleyici optimizasyonu olmadan, çoğu derleyiciler jsr, ret veya b den sonra noop ekler.

Örneğin:

```
a(int i)
{
  return b(i+1)+1;
}
```

Derlendiğinde;

```
a:
  ld [fp+12] -> %r0            / Yığına i+1 atar
  add %r0, %g1 -> %r0
  st %r0 -> [sp]--

  jsr b                        / b prosedürünü çağır
  noop                         / gecikme yuvası(delay slot)
  pop #4

  add %r0, %g1 -> %r0          / r0 içine b(i+1)+1 yükle

  ret                          / return
  noop                         / gecikme yuvası(delay slot)
```

Optimize edilmiş derleyici eğer jsr b yada ret çalıştırıldıktan sonra kod okumayı zorlaştıracak *AFTER* talimatlarını hatırlamak zorunda olduğu için gecikme yuvası kullanacaktır.Ayrıca JSR çağrısından sonra after komutuyla altprograma komut döndürür. Yukarıdaki prosüdür derlenmiş şekilde jsr ve ret ifadelerinin gecikme yuvası kullandığına bir örnektir.

```
a:
  ld [fp+12] -> %r0            / Yığına i+1 atar
  add %r0, %g1 -> %r0

  jsr b                        / b prosedürünü çağırır
  st %r0 -> [sp]--
  pop #4

  ret                          / return
  add %r0, %g1 -> %r0          / r0 içine b(i+1)+1 yükle.dönüş aslında
                               / oluşmadan önce çalıştırılır
```