Bu ders bilgisayar organizasyon & yığıt çerçevesinin devamıdır. Temel odak noktası yöntem çağrıları üzerindedir.

Aşağıdaki C koduna bakınız.

```
int a()
{
  return 1;
}

main()
{
  int i;

  i = a();
}
```

Bu, assembler'da aşağıdaki gibi derlenir.

```
    a:
        mov #1 -> %r0
        ret
    main:
        push #4
        jsr a
        st %r0 -> [fp]
        ret
```

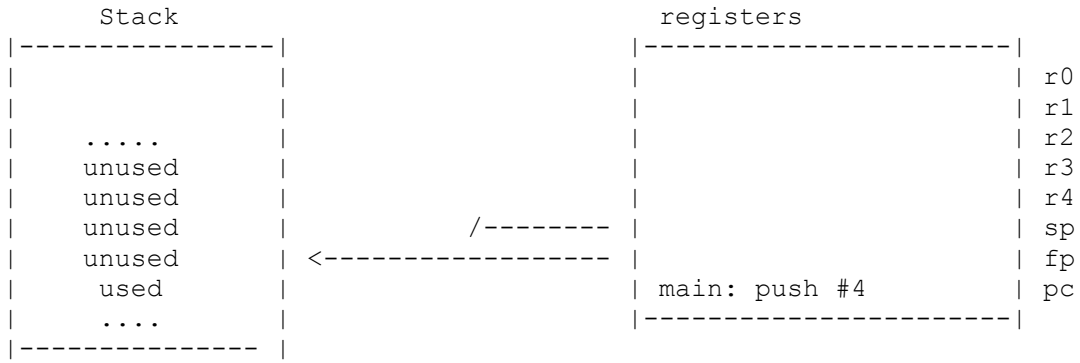
Böylece, bunların ikisi de doğrudur. **main()** öncelikle yığında bir değişken tahsis eder ve sonra altprogram a'ya geçiş anlamına gelen "**jsr a**" yı çağırır. a geri döndüğünde, kayıt **r0'** daki değeri, çerçeve göstergesi tarafından değişken noktaya tahsis edilen yerel değişken **i'** ye depolar. Daha sonra main() çıkar. A() da aynı zamanda doğrudur. r0' da depolanan 1 değerini döndürür ve daha sonra tekrar dönüyor.

Bu basit görünür fakat jsr ve ret çağırıldığında biraz daha aldatıcı olur. Bu her zaman olan bir şeydir.

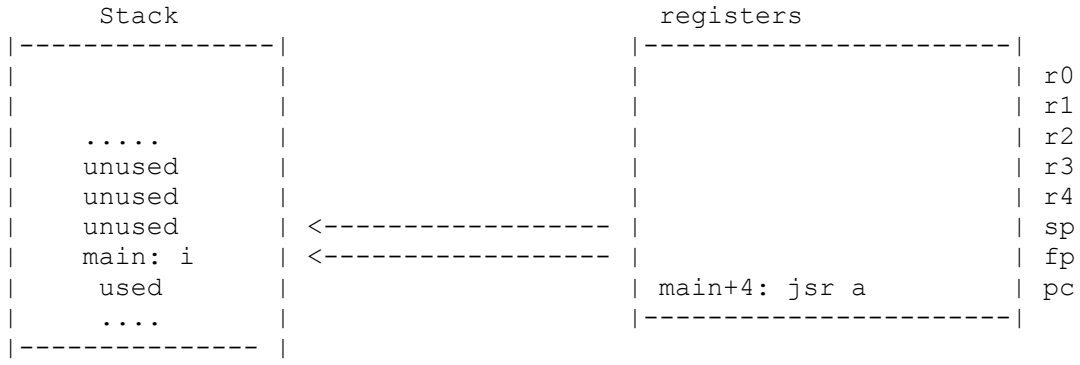
"jsr" çağırıldığında (pc+4) ve fp'nin son değeri yığının en üstüne depolanır. Daha sonra, fp, güncel sp ile değiştirilir ve pc adlandırılmış işlemin ilk talimatının yeriyle değiştirilir. Bu, bilgisayarın donanımı tarafından atomik bir şekilde yapılır. Jsr yürürlüğe girdikten sonra yeni yığın çerçevesindeyiz ve pc a()'yı yürütüyordur.

"ret" çağırıldığında sp, güncel fp ile değiştirilir. Daha sonra fp yığından ayrılır: bu en üst yığın değerinde olmak için ayarlanır ve sp 4 tarafından azaltılır. Son olarak, pc yığından ayrılır: bu en üst yığın değerinde olmak için ayarlanır ve sp yine 4 tarafından azaltılır. Aynı "jsr" için olduğu gibi bu da donanım tarafından atomik bir şekilde yapılır. "ret" sonlandırdığında pc, orijinal "jsr" ve restore edilen bu işlemin yığın çerçevesinden sonraki ilk talimatı olarak ayarlanır.

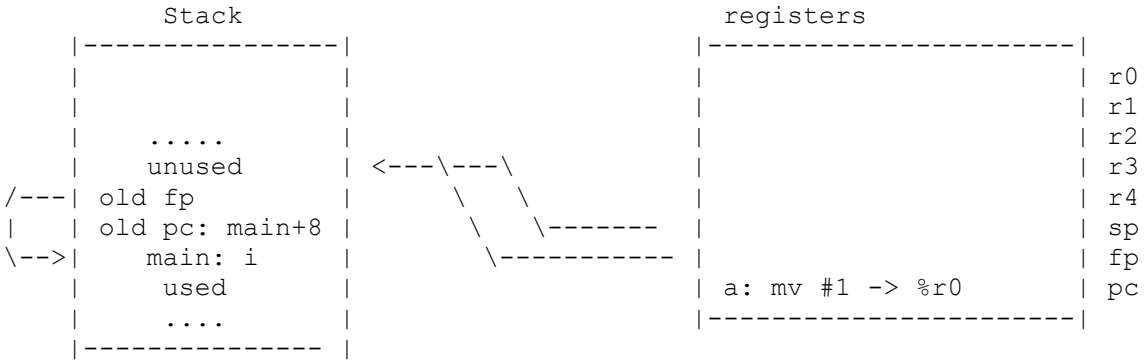
Buna resimli olarak bakalım (bunu kendiniz yapmanız için jassem.tcl kullanmanız gerekir). Başlangıçta yığın ve kaydediciler aşağıdaki gibidir:



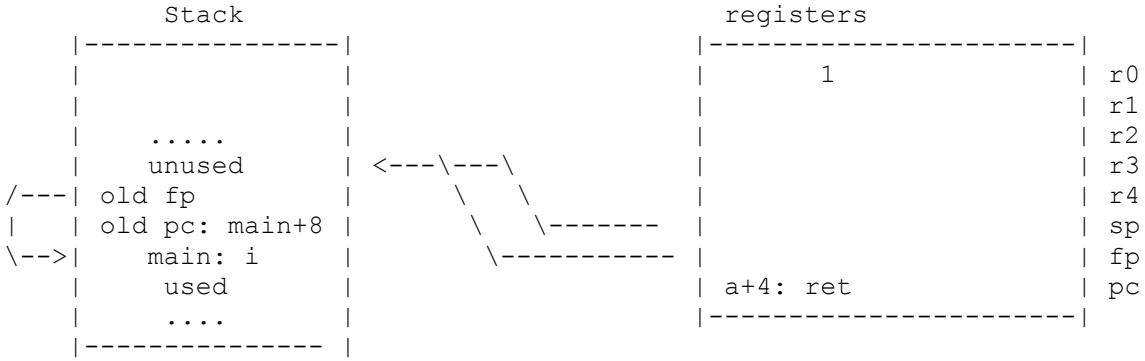
Öncelikle sp, 4 tarafından yerel değişken i sağlayabilmek için azaltılır:



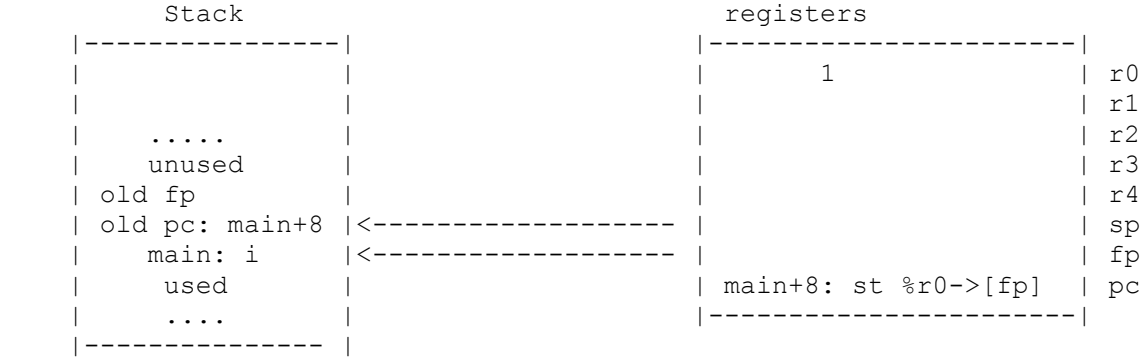
Şimdi jsr çağırılır. Bu pc+4 ve fp değerini yığının tepesine iter ve fp'yi yeni sp'ye; pc'yi a'ya ayarlar:



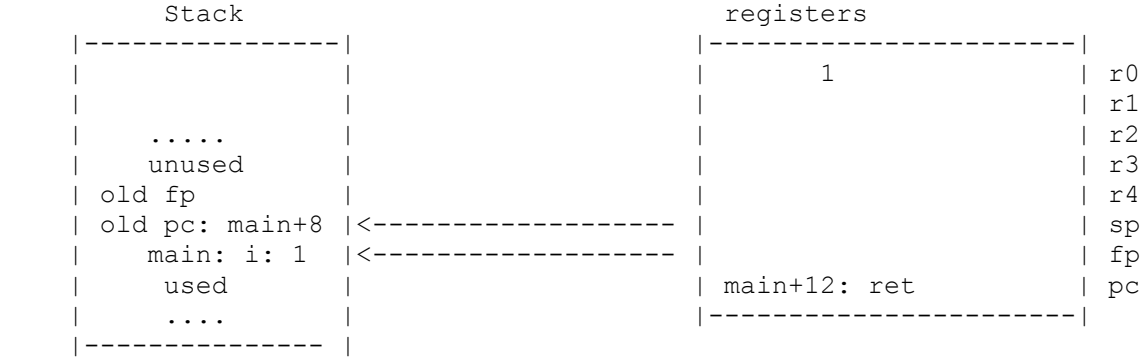
Şimdi a için yeni yığın çerçevemiz olduğunu ve pc'nin a'yı yürüttüğüne dikkat edin. Yaptığı ilk şey 1 'i r0' a yüklemektir.



Ve daha sonra “ret” çağırılır. “ret” sp’yi fp’ye ayarlar ve fp ve pc’yi yığından ayırır. Tamamlandığında yeniden main() yığın çerçevesindeyiz ve “jsr”den sonraki talimatı yürütmekteyiz:



Not: Eski **“fp”** ve eski **“pc”** değişmemiş oldu. Ancak onlar “yukarıdaki yığında” olduklarından, belli referans almaları gerekir.



Şimdi main() dışarıdadır ve “ret” i çağırır. Ne yaptığını tahmin edebilirsiniz- - main() “ret”i çağırdığında kontrol işletim sistemine dönebilsin ve işlem devam etsin diye yığın ayarlanır.

Şimdi bu Jassem ile incelediğinizden emin olun. bu program pla.jas içindedir ve yukarıda gösterdiklerimi ve tüm hafıza adreslerini tam olarak görmelisiniz.

Bu bir sonraki örnek bağımsız ve yerel değişkenlerle bir işlemi göstermektedir:

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

  j = i+1;
  return j;
}

main()
{
  int i;

  i = a(5);
}
```

Bu, aşağıdaki gibi bir kod ile derlenir:

```
a:
        push #4
        ld [fp+12] -> %r0
        add %r0, %g1 -> %r0
        st %r0 -> [fp]
        ld [fp] -> %r0
        ret
    main:
        push #4
        mov #5 -> %r0
        st %r0 -> [sp]--
        jsr a
        pop #4
        st %r0 -> [fp]
        ret
```

Bu örnek ve sonuncusu arasındaki tek gerçek farklılık a() bağımsız değişkenidir. A'nın yığın göstergesini artırarak yerel değişkeni j'yi yığında nasıl tahsis ettiği net olmalıdır. O zaman j çerçeve göstergesi tarafından işaretlenmiş yerleşim olarak bildirilir. Bağımsız değişkenler, çağırma işlemiyle onları ters sıraya iterek ve jsr çağırarak geçilir. İşlem bağımsız değişkenleri nasıl ilişkilendireceğini bilir- - hafıza yerleşiminde fp'nin 12 byte önünde başlarlar. Neden? [fp] çerçevenin başlangıcına işaret eder. [fp+4] eski çerçeve göstergesini ve [fp+8] işlem bittiğinde pc'nin geri dönüşüne işaret eder. Böylece bağımsız değişkenler jsr'yi çağırmadan önce yığına direk olarak itilirler ve daha sonra [fp+12] de başlarlar.

Bu program p2.jas'tadır ve jassem'i kullanarak izini sürmelisiniz. Main'in kendi bağımsız değişkenini yığına nasıl itiiğini, a'nın nasıl bağımsız değişkeni bulduğunu ve jsr/ret'te ne olduğunu anladığınızdan emin olun.

Kayıt ortaya çıkarma

Bir işlemin güncel değerini düşünmeden kayıt kullanabileceği veya bir işlemin bir kaydı kullanmadan önce onu yığında koruması gerektiği karar verilmesi gereken önemli bir şeydir. Bu önemlidir, çünkü örneğin ana rutin kayıt r3'ü kullanır sonra "jsr a" yı çağırır ve daha sonrasında r3 aynı değere sahip olmayı bekler. Sonra a() ve a()'nın çağırdığı herhangi bir

işlem r3'ü kullanmadığından veya r3'ün değerini kullanmadan önce koruduğunu ve tamamlandığında düzelttiğinden emin olmalıdır.

Çağrı işleminden önce kaydın değerini korumak ve sonrasında onu düzeltmek hareketi "ortaya çıkarma" olarak adlandırılır. Farklı makineler ve derleyiciler ortaya çıkarmayı farklı yollarda ele alırlar. Örneğin; eski CISC yapıları bazen çağrı işleminin bir parçası olan ortaya çıkarma maskesine sahipti. Bu, hangi kayıtların ortaya çıkarılması gerektiğini özelleştirdi ve makine aslında ortaya çıkarmayı sizin için yaptı.

Bizim makinemizde yaptığımız tipik bir ortaya çıkarma çözümüdür. İşlemler değerlerini düşünmeden r0 ve r1 kullanabilir. Ancak r2 ve r4 bir işlem tarafından kullanılırsa ortaya çıkarılmalıdır.

Burada bir örnek var:

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

  k = (i+2)*(j-5);
  return k;
}
```

Eğer bunun hakkında düşünürseniz, sadece r0 ve r1'i kullanarak bu aritmetiği yapmanın bir yolu yoktur. Bu yüzden işlemin başında r2'yi yığının üzerine atmalısınız ve geri dönmeden önce düzeltmelisiniz.

```
a:
    push #4
    st %r2 -> [sp]--     / %r2 atıldı

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0
    ld [fp+16] -> %r1
    mov #5 -> %r2
    sub %r1, %r2 -> %r1
    mult %r0, %r1 -> %r0
    st %r0 -> [fp]
    ld [fp] -> %r0

    ld ++[sp] -> %r2    / %r2 geri alındı
    ret
```

Yerel değişkeni ayırdıktan sonra r2'yi yığının üzerine çıkarmak zorunda olduğunuza dikkat edin. Yoksa k fp'de olmayacaktır. Bunu düşünün.

Göstergeli bazı kodlar

Göstergeler biraz dikkat gerektirir- - göstergelerle ilgili benim tavsiyem yavaş gitmeniz ve tam olarak düşünmenizdir. İşte göstergelerin farklı çeşitleri ve onlarla başa çıkabileceğiniz yöntemler:

Basit gösterge elde etme. Örneğin;

```
a(int *p)
{
  return *p
}
```

İşte basit bir şekilde göstergenin değerini bir kaydediciye geçirebilirsiniz ve sonra kaydı elde edebilirsiniz. İşte yukarıdaki örnek:

```
a:
    ld [fp+12] -> %r0    / ro'a p'deki alındı
    ld [r0] -> %r0       / refererans olmayan r0
    ret
```

Gösterge aritmetiği dahil edildiğinde bunda bir dönüş olur- - o zaman gösterilen nesnenin boyutunu çoğaltmayı hatırlamalısınız. Örneğin:

```
int a(int *p)
{
  return *(p+2)
}
```

Derlenir:

```
a:
   ld [fp+12] -> %r0
   mov #8 -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   ret
```

Dizi referanstan ayırma. Dizi referanstan ayırma gösterge referanstan ayırma gibidir. Nesnenin boyutuyla dizi indeksini çoğaltırsınız ve sonra onu dizinin en üstüne koyarsınız. Daha sonra bu değeri referanstan ayırırsınız. Örneğin aşağıdaki kod parçasına bakınız.

```
a(int *p)
{
  int i;

  i = p[0];
  i = p[5];
  i = p[i];
}
```

Bunlar dizi referanstan ayırmanın üç çeşididir. Birincisinde; p'yi r0'a basitçe yükleriz ve onu referanstan ayırırız. İkincisinde; p'yi yükler, 20 ekler ve referanstan ayırırız. Üçüncüsünde, i'yi 4 ile çoğaltırız, buna p'yi ekler ve daha sonra referanstan ayırırız.

```
a:
   push #4

   ld [fp+12] -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]
```

```
   ld [fp+12] -> %r0
   mov #20 -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

   ld [fp] -> %r0
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

   ret
```

Eğer derleyici değerleri zaten biliyorsa çoğaltma ve eklemeyi onu kodun içine koymaktan çok derleme zamanında yapar. Bu yüzden bir kayda 5, bir başkasına 4 hareket ettirmek ve onları çoğaltmak yerine r1'e 20 hareket ettirebiliriz.

Bu kavram bir dizinin yerel bir değişken olduğunu bildirdiğinizde de iş başındadır:

```
main()
{
  int a[5];

  a[2] = 3;
}
```

"**push #20**" yi çağırarak a'yı ayırabiliriz. Derleyici derleme zamanında şunu bilir:

- **a[0]** will be at **[fp-16]**
- **a[1]** will be at **[fp-12]**
- **a[2]** will be at **[fp-8]**
- **a[3]** will be at **[fp-4]**
- **a[4]** will be at **[fp]**

Bu yüzden yukarıdaki program şöyle olur:

```
main:
   push #20
   mov #3 -> %r0
   st %r0 -> [fp-8]
   ret
```

Ancak derleyici sabit göstermelerle uğraşmıyorsa referanstan ayırmadan önce çoğaltma ve dizinin en üstüne ekleme yapmalıdır. Örneğin;

```
int a(int i)
{
  int b[5];

  return b[i];
}
```

Şuna derlenir:

```
a:
   push #20
   ld [fp+12] -> %r0       / + ile i'yi çarp ve ro'a koy
   mov #4 -> %r1
   mul %r0, %r1 -> %r0

   mov #-16 -> r1          / dizinin parçalarını r1'e koy
   add %r1, %fp -> %r1

   add %r1, %r0 -> %r0     / dizinin parçalarına 4*i'yi ekle

   ld [r0] -> %r0          / refererans değil ve döndür
   ret
```

Adresler benzerdir- - göstergeyi referanstan ayırmazsınız:

```
int *a(int p)
{
  return &p;
}
```

Şuna derlenir: a: mov #12 -> %r0 add %r0, %fp -> %r0 ret çalışan ikili dolaylamalar da açıktır- - sadece şu şekilde düşünmelisiniz:

```
int a(int **arr, int i, int j)
{
  return a[i][j];
}
```

Şuna derlenir:

```
a:
   st %r2 -> [sp]--         / r2 ortaya çıktı çünkü ona ihtiyacınız olacak

   ld [fp+16] -> %r0
   mov #4 -> %r1
   mul %r0, %r1 -> %r0
   ld [fp+12] -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0          / a[i] şimdi r0'da

   ld [fp+20] -> %r1
   mov #4 -> %r2
   mul %r1, %r2 -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0          / a[i][j] şimdi r0'da
   ld ++[sp] -> %r2
   ret
```

Son olarak, işte göstergelerle kodun iki ek parçası. Birincisi (pimp.c' ye bakın) anlaşılır gösterge ve dizi işletimi yapar:

```
main()
{
  int *a, a2[3], i;
```

```
  i = 6;
  a = &i;
  a2[1] = i+2;
  *a = 200;
  *(a2+2) = i+5;
}
```

Şimdi bunu derlemek için öncelikle tüm yerellerin nerede olacağını anlamanız gerekir. Aşağıdaki kodda onları belirtilen yerlere koyacağım:

- **i**, **fp]** olacak.
- **a2[0]**, **a2[1]** ve **a2[2]**, **[fp-12], [fp-8] ve [fp-4]** olacak.
- **a**, **[fp-16]** olacak.

Bunun şu anlama geldiğine dikkat edin:

- &i, fp olacak
- A2, fp-12 olacak
- *a, **[[fp-12]] olacak.** Bunu çeviricide gerçekten yapamayacağınıza dikkat edin. Bunun yerine [fp-12]'yi bir kayda yüklersiniz ve bu kaydı referanstan ayırırsınız.
- A2+2, fp-4 olacak. Unutma—bu gösterge aritmetiğidir.

İşte çevirici:

```
main:
        push #20             / yerel değişkenler tahsis edilir
        st %r2 -> [sp]--     / r2 açığa çıkarılır

        mv #6 -> %r0         / i = 6
        st %r0 -> [fp]

        st %fp -> [fp-16]    / a = &i

        ld [fp] -> %r0      / a2[1] = i+2
        mv #2 -> %r1
        add %r0, %r1 -> %r0
        st %r0 -> [fp-8]

        mv #200 -> %r0       / *a = 200
        ld [fp-16] -> %r1
        st %r0 -> [r1]

        ld [fp] -> %r0      / *(a2+2) = i+5
        mv #5 -> %r1
        add %r0, %r1 -> %r0
        mv #-12 -> %r1
        add %fp, %r1 -> %r1
        mv #8 -> %r2
        add %r1, %r2 -> %r1
        st %r0 -> [r1]

        ld ++[sp] -> %r2      / r2 açıktan alınır
        ret
```

P3.c' ye bakınız:

```
int *a(int *x)
{
  x[0] += x[2];
  return x+1;
}

main()
{
  int array[3];
  int *ip;

  array[0] = 8;
  array[1] = 9;
  array[2] = 10;

  ip = a(array);
  *ip = *ip+1;
}
```

Çalıştırmayı bitirdiğinizde ip'nin dizin[1] ögesini işaret etmesi ve dizi ögelerinin aşağıdaki değerlere sahip olması gerektiğine kendinizi inandırın:

- **array[0]** is 18
- **array[1]** is 10
- **array[2]** is 10

Şimdi, bunu çeviriciye derlemek biraz şaşırtıcıdır. Öncelikle işte a:

```
a:
    st %r2 -> [sp]--      / r2 açığa alınır
    st %r3 -> [sp]--      / r3 açığa alınır
    ld [fp+12] -> %r0     / x'i r0'a yükle
    ld [r0] -> %r1        / x[0]'ı r1'e yükle
    ld [fp+12] -> %r2     / x[2]'yi r2'ye yükle
    mov #8 -> %r3
    add %r2, %r3 -> %r2
    ld [r2] -> %r2
    add %r1, %r2 -> %r2   / ekle ve sonucu r2'ye koy
    st %r2 -> [r0]        / r2'yi x[0]'ın içine depola

    ld [fp+12] -> %r0     / x+1 döner
    mov #4 -> %r1
    add %r0, %r1 -> %r0

    ld ++[sp] -> %r3      / r3'ü onar
    ld ++[sp] -> %r2      / r2'yi onar
    ret
```

A, r2 ve r3'ü kullanır bu yüzden yaptığı ilk şey onları yığına atmaktır. Daha sonra ip[0]'ı r1'e ve ip[2]'yi r2'ye yükleriz. Sonrasında bunları ekler ve sonucu ip[0]'a tekrar depolarız.

İkinci talimat ip'ye 4 ekler (gösterge aritmetiği) ve bu değere geri döner. Daha sonra atılan kayıtları ve geri dönüşleri yeniden düzenler.

İşte main:

```
main:
    push #16           // Dizi = fp-8, ip = fp-12
    mov #8 -> %r0
    st %r0 -> [fp-8]
    mov #9 -> %r0
    st %r0 -> [fp-4]
    mov #10 -> %r0
    st %r0 -> [fp]

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

    ld [fp-12] -> %r0
    ld [r0] -> %r1
    add %g1, %r1 -> %r1
    st %r1 -> [r0]
```

Öncelikle main 16 byte değerinde yerel değişken ayırır. Derleyici dizinin fp-8'de başlayacağına karar verir. Bu, dizi[0]'ın fp-8'de olacağı anlamına gelir. Dizi[1] fp-4 ve dizi[2] fp'de alacaktır. Bunlar verildiğinde üç dizi değerinin yerleşimi açıktır.

Daha sonra dizi (fp-8)'i yığına koyar ve a'yı çağırırız. Sonucu ip'ye depolarız. Son olarak *ip'ye bir tane ekleriz.

Daha önceki gibi, bunu jassem.tcl ile takip edin ve her şeyi anladığınızdan emin olun.

Bu son bölüm aşağıdaki main için kodu göstermektedir:

```
main(int argc, char **argv)
{
  char *s;

  s = malloc(atoi(argv[1]));

  read(0, s, 10);

  write(1, s, strlen(s));

}
```

Main'in herhangi bir işleme çok benzediğine dikkat edin. Bağımsız değişkenlerinin fp+12 ile başlayan ters dizilişte yığının üzerine itildiğini varsayar. Ayrıca sistem çağırmasına rağmen okur ve yazar. Fark, programın talimatlarında biten read() ve write() kodlarının aslında işletim sistemine sistem çağrıları yapmasıdır.

Aşağıdaki main() için çeviricidir. Anlaşılır olmalıdır. Strlen() değerinin geri dönüşü, atoi() değerinin geri dönüşünde olduğu gibi write() a 3. bağımsız değişken olarak yığına itilir.

Argv[1]'in nasıl bulunduğuna da dikkat ediniz. Argv diziye bir göstergedir. Böylece argv değerinden sonra argv[1] 4 byte değerindedir.

```
main:
        push #4                 / tahsis s

        ld [fp+16] -> %r0       / argv'yi r0'a koy
        mov #4 -> %r1
        add %r1, %r0 -> %r0
        ld [r0] -> %r1          / argv[1]'yi r1'e koy
        st %r1 -> [sp]--        / r1'i yığıta at
        jsr atoi                / atoi(argv[1])'i çağır
        pop #4

        st %r0 -> [sp]--        / geri dönüş değerini yığıta at
        jsr malloc              / malloc'u çağır
        pop #4

        st %r0, -> [fp]         / geri dönüş değerini s içinde depola

                                / argümanları okumak için yığıta at
        mov #10 -> %r0          / 10
        st %r0 -> [sp]--
        ld [fp] -> %r0           / s
        st %r0 -> [sp]--
        mov #0 -> %r0           / 0
        st %r0 -> [sp]--
        jsr read                / read'i çağır
        pop #12

        ld [fp] -> %r0           / argument'i strlen içine at
        st %r0 -> [sp]--
        jsr strlen              / strlen'i çağır
        pop #4

                                 / argümanları yazmak için yığıta at
        st %r0 -> [sp]--        / strlen(s)
        ld [fp] -> %r0           / s
        st %r0 -> [sp]--
        mov #1 -> %r0           / 1
        st %r0 -> [sp]--
        jsr write               / write'ı çağır
        pop #12

        ret
```

Her talimattan sonra yığını göstermeyeceğim. Kendiniz takip etmelisiniz. Ne yazık ki read jassem'de gerçekleştirilmediğinden onu burada kullanamazsınız. Bağımsız değişkenleri yığından çıkarmak için yapılan her işlem çağrısından sonra yığının nasıl ayarlanması gerektiğine dikkat edin. Derleyici ayarlandığında bazı işletimleri koruyabilirsiniz. Örneğin; "jsr atoi" den sonra sadece şunu yapmalısınız:

```
        st %r0 -> [sp+4]
        jsr malloc
```

Ancak ayarlanmamış derleyiciler etkisiz de olsa kolay okunan kodlar üretirler.