Sistem Programlama

DR. ÖĞR. ÜYESİ ABDULLAH SEVİN

İçerik

a) Assembler

- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Assembler2/lecture.html
- http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Assembler3/lecture.html

Fonksiyon

- Bu ders, bilgisayar organizasyonu ve yığın-stack çerçevelerinin bir devamıdır ve fonksiyon çağrılarına odaklanmaktadır.
- Yanda bir fonksiyon içeren C kodu ve assembler'daki karşılığı verilmiştir.
- Bu fonk. çağrılarının her ikisi de basittir.
- Main() önce yığında bir değişken tahsis eder ve sonra "jsr a"yı çağırır, bu da alt program a'ya dallanmak anlamına gelir.
- a()'nın tek yaptığı 1 döndürmek -- bunu r0'ı 1'e ayarlayıp "ret"i çağırarak yapar.
- Kontrol main() fonksiyana döndüğünde, a'nın r0'daki dönüş değerini **i** için ayırdığı belleğe depolar.

p1.c

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

int main()
{
  int i;

  i = a();
}
```

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

Fonksiyon

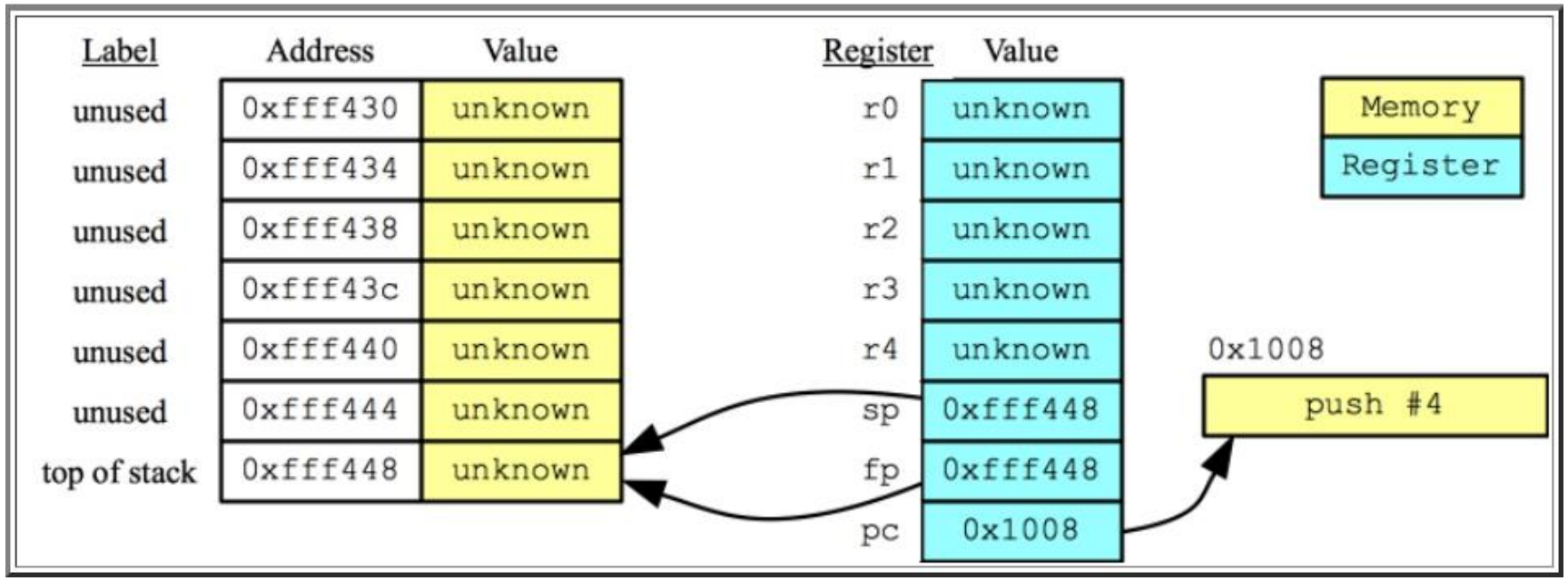
- Bu basit görünüyor; ancak, jsr ve ret çağrıldığında olanlar biraz daha karmaşıktır.
- Olan şu:"jsr" çağrıldığında, (pc+4 bir sonraki çalıştırılacak komut) ve fp'nin mevcut değeri yığına itilir-push.
- İtme-push nedeniyle, sp'nin değeri **jsr** çağrısından önceki değerden 8 daha az olacaktır.
- Daha sonra, **fp** geçerli **sp** olarak değiştirilir ve **pc** adı verilen fonksiyonun ilk komutunun konumu olarak değiştirilir.
- Bu, bilgisayarın donanımı tarafından atomik (kesintiye uğramadan) olarak yapılır.
- jsr devreye girdikten sonra, yeni bir yığın çerçevesindeyiz ve bilgisayar a()'yı yürütür.

Fonksiyon

- "**ret**" çağrıldığında, **sp** geçerli **fp** olarak değiştirilir.
- Ardından fp yığından çıkarılır: sp'nin değeri dört artırılır ve fp yığından okunur.
- Son olarak, pc yığından çıkarılır: sp'nin değeri tekrar dört artırılır ve pc yığından okunur.
- "jsr" gibi, bunların hepsi donanım tarafından atomik (kesintisiz) olarak yapılır.
- "ret" tamamlandığında, bilgisayar orijinal "jsr" komutundan sonraki komut olacak şekilde ayarlanır ve bu prosedürün yığın çerçevesi geri yüklenir.

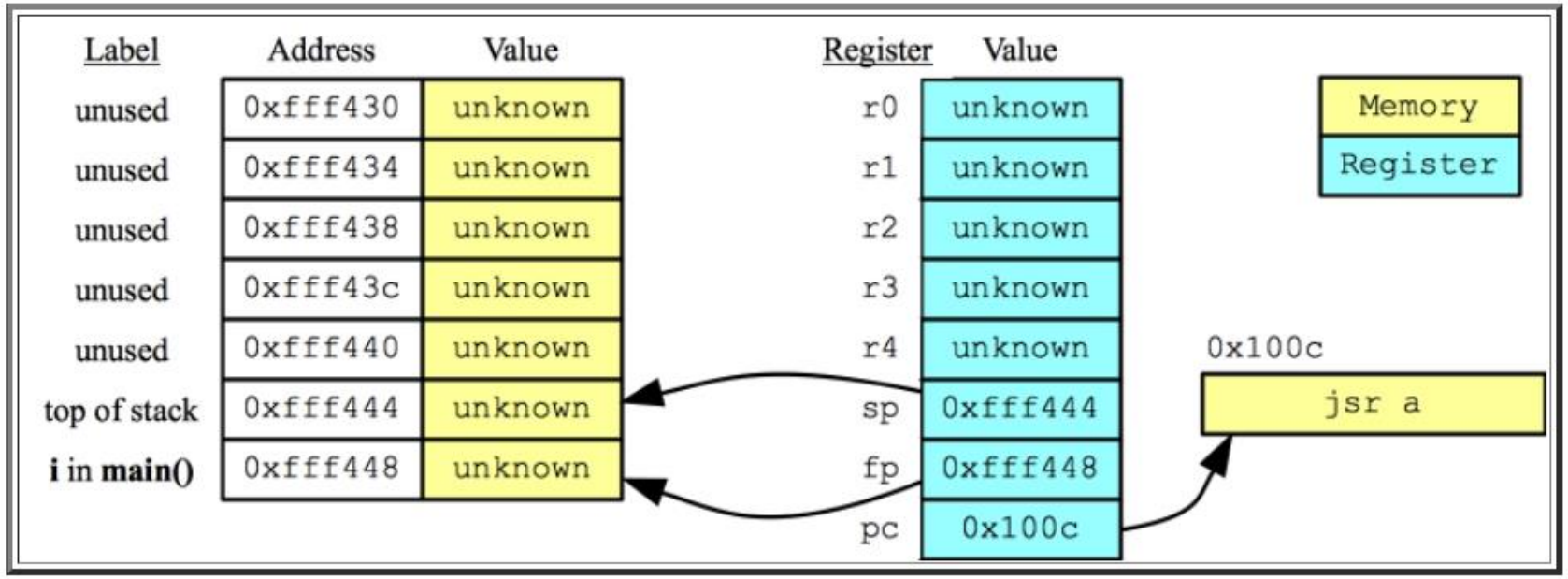
Fonksiyon

- Aşağıda jassem.tcl'yi p1.jas üzerinde çalıştırırsanız göreceğiniz şeyin bir çizimi bulunmaktadır. Programın başlangıcında, yığın ve kayıtlar aşağıdaki gibi görünür:



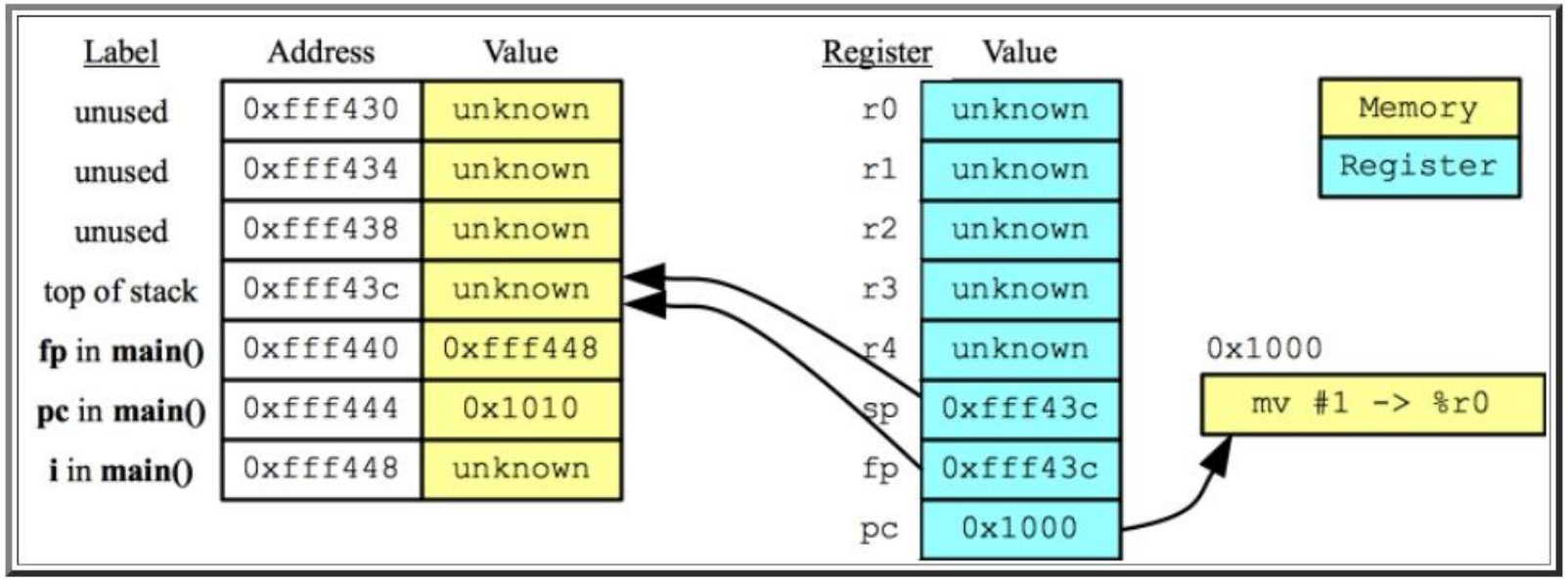
Fonksiyon

- (Bu arada -- jassem.tcl bilinmeyen değerlere ve kayıtlara **sıfır** atar. Burada değerlerin gerçekte ne olacağını bilmediğimizi göstermek için "unknown" koyuyoruz.)
- İlk olarak, **i** yerel değişkenini tahsis etmek için sp 4 azaltılır:



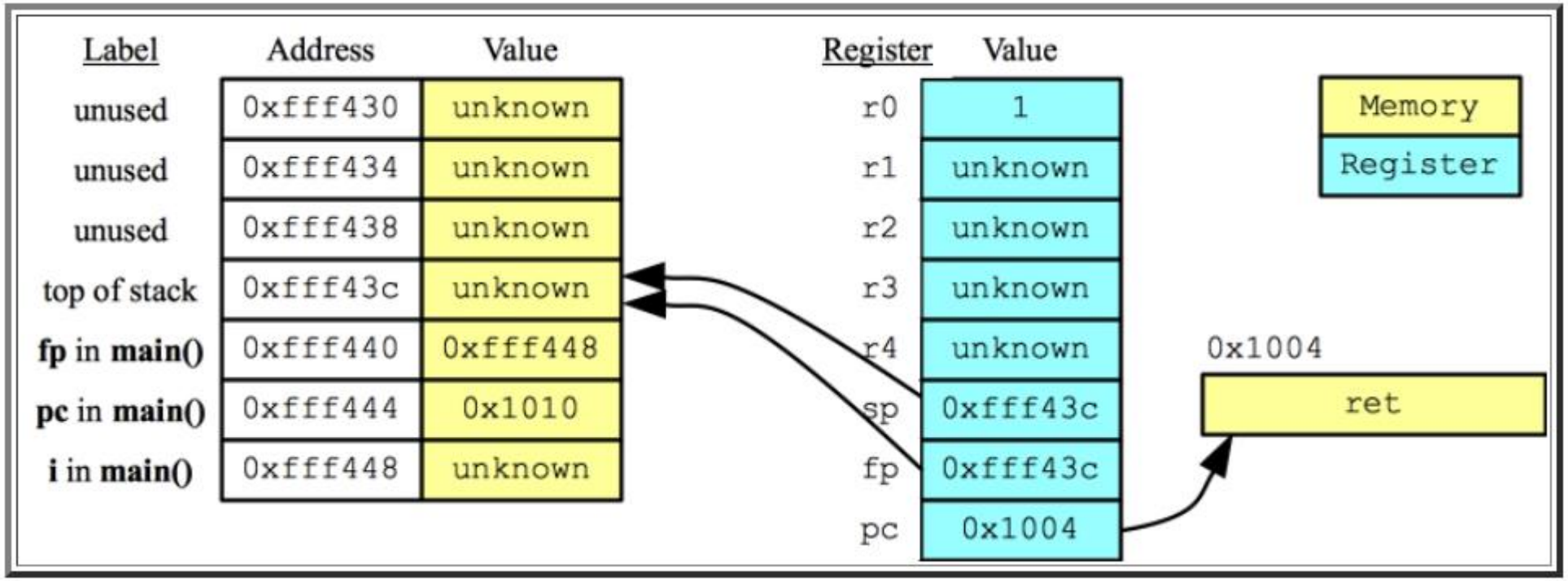
Fonksiyon

- Daha sonra **jsr** çağırılır. Bu, yığındaki (pc+4) ve fp'nin değerini iter-push ve fp'yi yeni sp'ye ve pc'yi a'ya ayarlar:



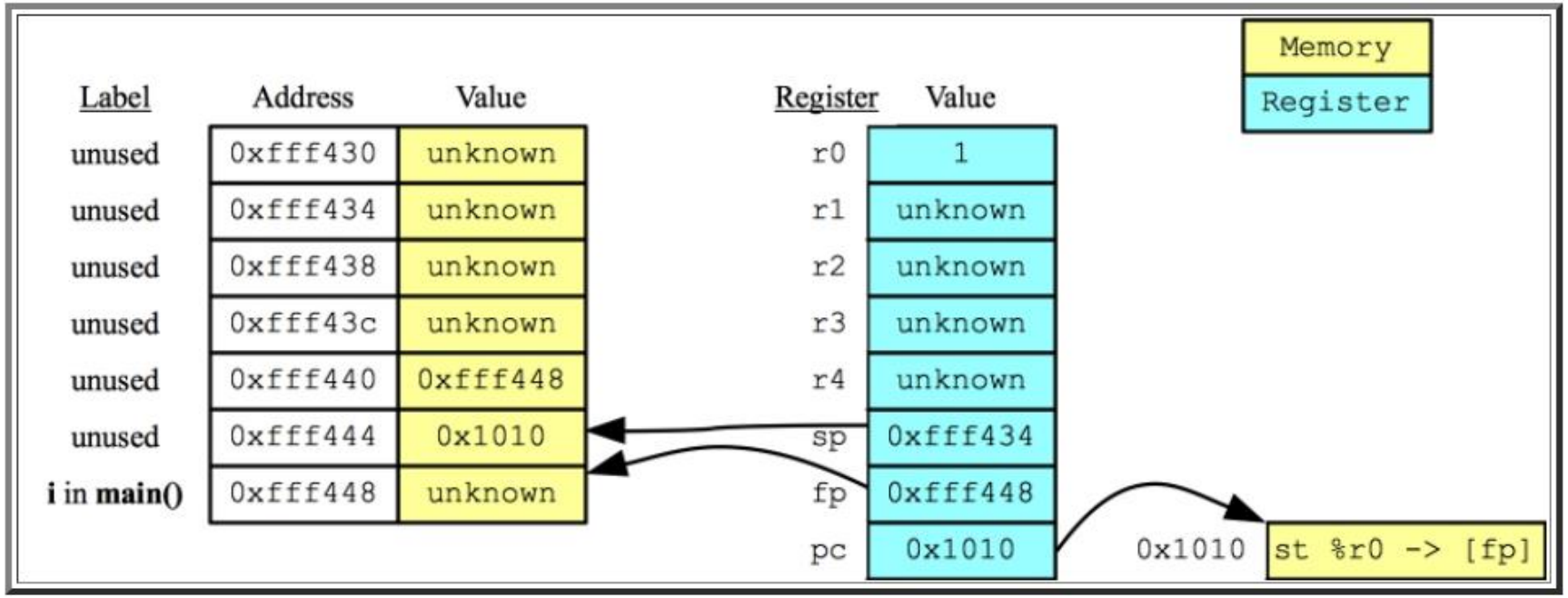
Fonksiyon

- Şimdi a için yeni bir yığın çerçevemiz olduğuna ve bilgisayarın a'yı yürüttüğüne dikkat edin. Yaptığı ilk şey 1'i r0'a yüklemek:



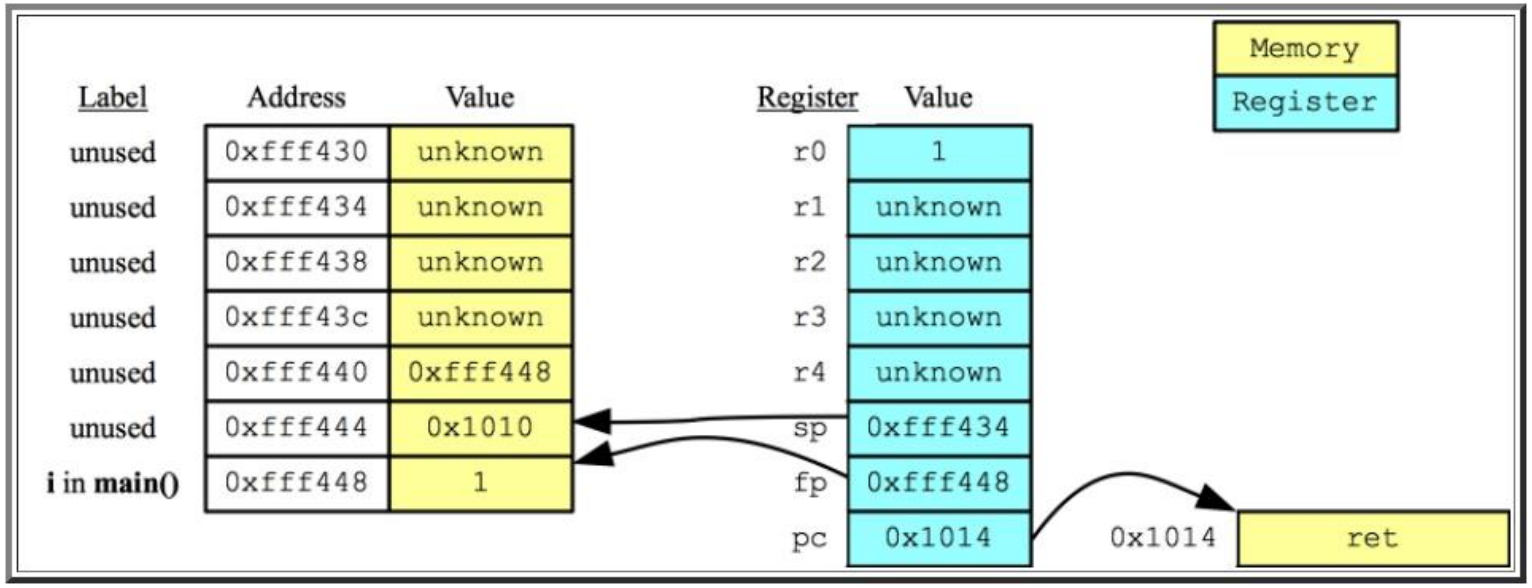
Fonksiyon

- Daha sonra "ret" diyoruz. "Ret", sp'yi fp'ye ayarlar ve sonra fp'yi ve bilgisayarı yığından çıkarır.
- Tamamlandığında, main()'in yığın çerçevesine geri dönüyoruz ve jsr'den sonraki komutu yürütüyoruz:



Fonksiyon

- "main içindeki fp" ve "main içindeki pc" değerlerinin değiştirilmediğini veya silinmediğini unutmayın.
- Sadece yığında kalırlar. Ancak "yığının üstünde" oldukları için referans alınmamalıdır.
- Şimdi "st %r0 -> [fp]" yürütülür ve makine durumu şöyle görünür:



Fonksiyon

- Şimdi main() bitti ve "ret"i çağırıyor.
- yığın, main() ret'i çağırdığında, kontrol işletim sistemine geri dönecek ve süreç sona erecek şekilde ayarlanmıştır.
- Program p1.jas'ta ve yukarıda gösterdiğim şeyi tam olarak görmelisiniz.
- Bir sonraki örnek, fonk. parametreleri ve yerel değişkenler içeren bir prosedürü göstermektedir.
- C kodu (p2.c): solda ve birleştirici (p2.jas): sağda:

Fonksiyon

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

  i++;
  j -= 2;
  k = i * j;
  return k;
}

int main()
{
  int i, j, k;

  i = 3;
  j = 4;
  k = a(j+1, i);
  return 0;
}
```

```
a:
    push #4                 / Allocate k, which will be [fp]

    ld [fp+12] -> %r0       / i++
    add %r0, %g1 -> %r0
    st %r0 -> [fp+12]

    ld [fp+16] -> %r0       / j -= 2
    mov #2 -> %r1
    sub %r0, %r1 -> %r0
    st %r0 -> [fp+16]

    ld [fp+12] -> %r0       / k = i * j
    ld [fp+16] -> %r1
    mul %r0, %r1 -> %r0
    st %r0 -> [fp]

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

```
ain:
    push #12                / Allocate i, j, k.
                            / i is [fp-8], j is [fp-4], k is [fp]

    mov #3 -> %r0           / i = 3
    st %r0 -> [fp-8]
    mov #4 -> %r0           / j = 4
    st %r0 -> [fp-4]

    ld [fp-8] -> %r0        / Push i onto the stack
    st %r0 -> [sp]--

    ld [fp-4] -> %r0        / Push j+1 onto the stack
    add %r0, %g1 -> %r0
    st %r0 -> [sp]--

    jsr a                   / Call a(), then pop the arguments
    pop #8

    st %r0 -> [fp]          / Put the return value into k

    mov #0 -> %r0           / Return 0
    ret
```

[p2.c]

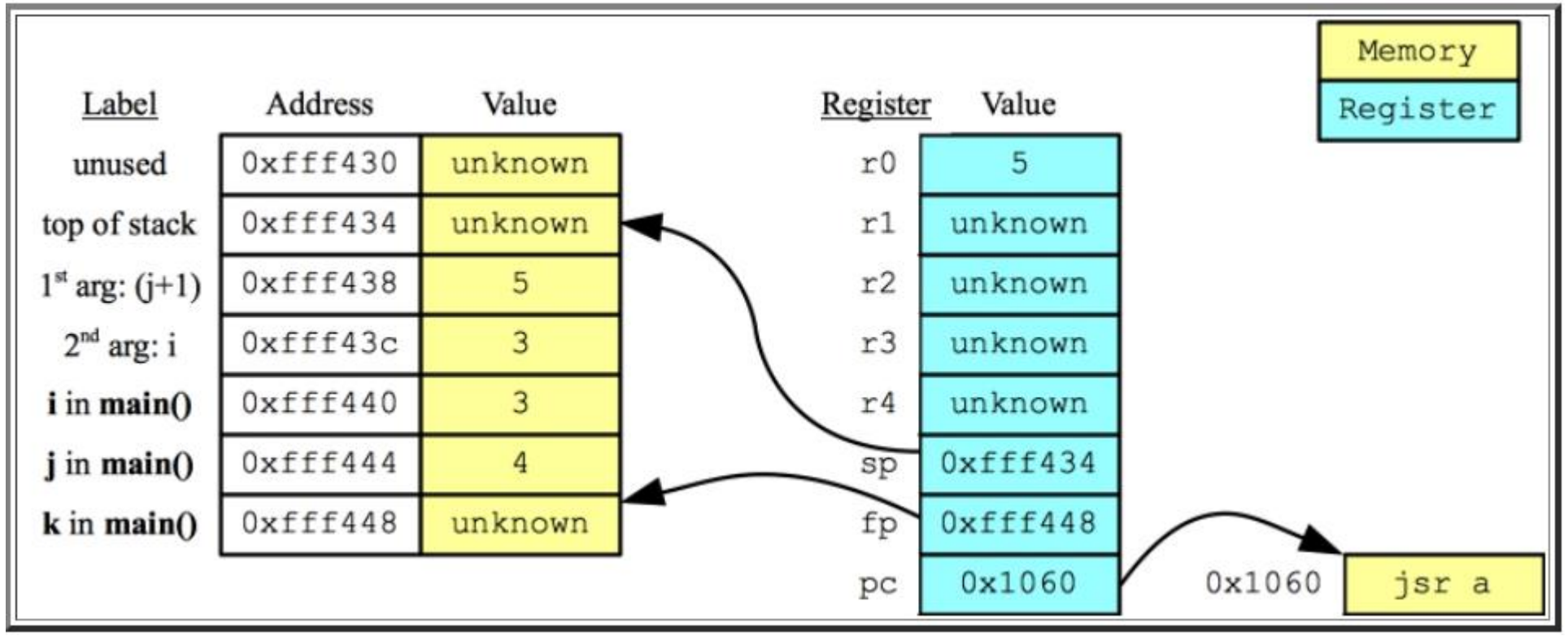
[p2.jas]

Fonksiyon

- Önce main() üzerine odaklanalım. Başlangıçta, yığın üzerinde i, j ve k'yi tahsis eden yığın işaretçisini 12 azaltır.
- Derleyicimiz bunları yığında bildirildikleri sıraya koyar, böylece yukarıdaki yorumlar şunu belirtir:
- i [fp-8]'de, j [fp-4]'te ve k [fp]'de. Ardından, i ve j'yi başlatıyoruz.
- Parametreli bir fonk. çağırdığınızda yaptığınız şey, parametreleri yığına ters sırada itmektir-push.
- Sonra jsr çağırılır. jsr çağrısı geri döndüğünde, o hafızayı yeniden kullanabilmek için argümanları yığından çıkarırız.
- Bu durumda, main() jsr'yi çağırdığında yığına bir göz atalım.
- Manuel çizim olsa da değerler ve adresler bunu jassem'de çalıştırdığınızda olanlarla eşleşiyor:

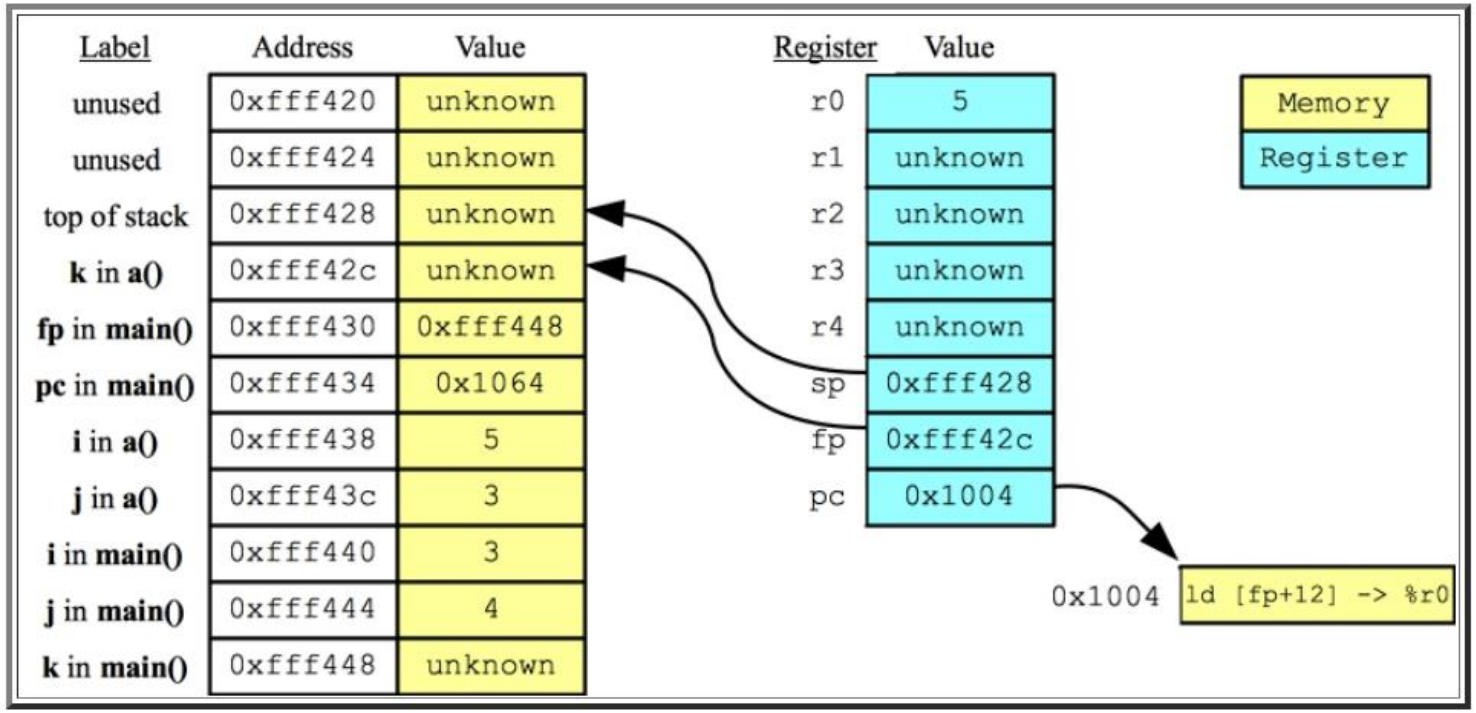
Fonksiyon

- Yığını etiketlemek iyi bir uygulama



Fonksiyon

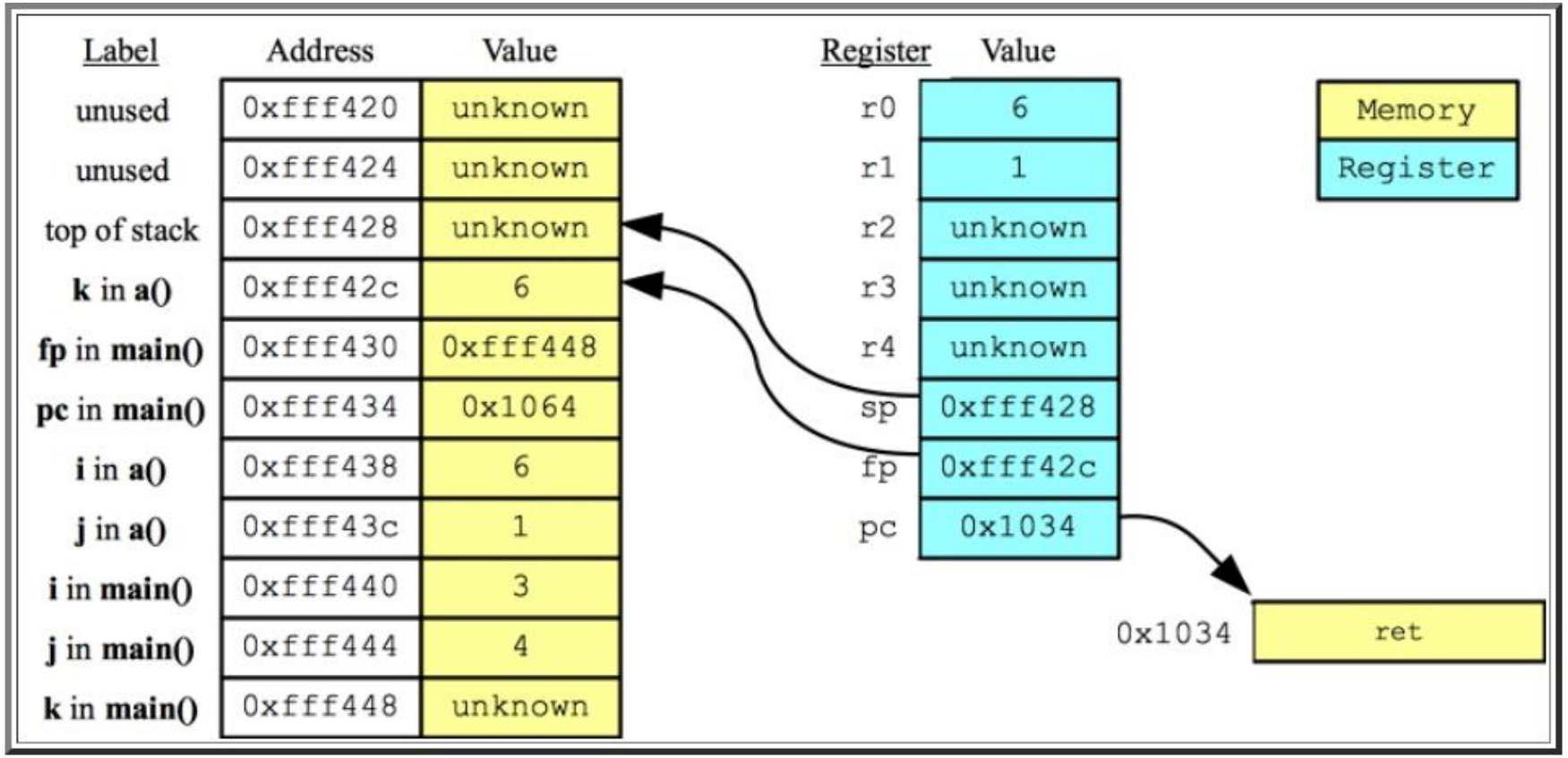
- Şimdi, jsr deyimi (pc+4)'ü yığına ve ardından fp'nin değerini iter-push.
- Daha sonra bilgisayarı a'daki ilk komuta ayarlar. Bu komut #4 değerini yığına iter, böylece k'ya tahsis edilir. Yığına aktarılan iki parametre"a() daki i" ve "a() daki j" olarak yeniden etiketledik:



Fonksiyon

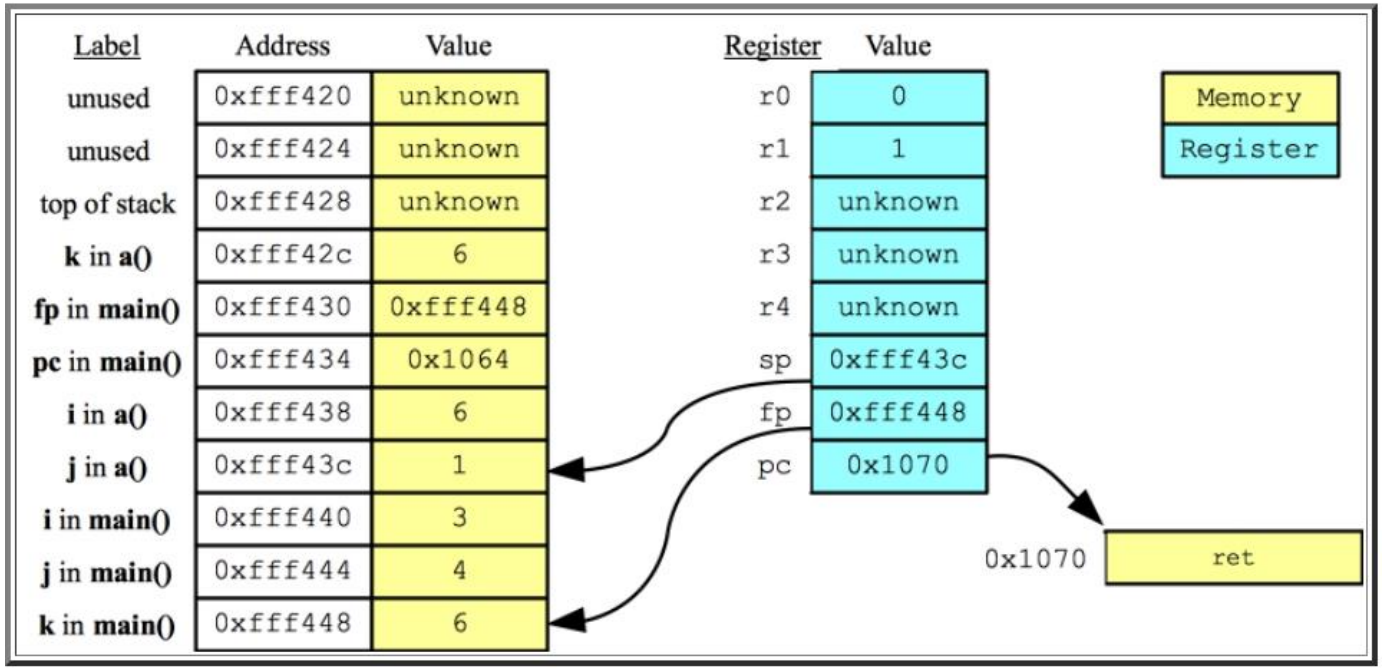
- Şimdi, a()'nın iki bağımsız değişkenini nasıl bulduğunu düşünelim.
- Eski fp'ın [fp+4]'te ve eski pc'ın [fp+8]'de olduğunu bilinmektedir. Parametreler ters sırada gönderildiğinden, ilk parametre [fp+12]'de bir sonraki olmalıdır.
- İkinci parametre [fp+16]'dadır. Üçüncü bir parametre olsaydı, [fp+20]'de olurdu ve bu böyle devam ederdi.
- Şimdi, a()'dan döndüğü andaki duruma bakalım. Bu noktada, a()'nın i, j ve k değişkenlerinin sırasıyla 6, 1 ve 6'ya eşit olduğunu görmelisiniz. k'nin değeri r0 kayıtçısına-register yüklendi, çünkü dönüş değerleri bir prosedürden diğerine bu şekilde aktarılır.

Fonksiyon


Label
Address
Value
unused
0xfff420
unknown
unused
0xfff424
unknown
top of stack
0xfff428
unknown
k in a()
0xfff42c
6
fp in main()
0xfff430
0xfff448
pc in main()
0xfff434
0x1064
i in a()
0xfff438
6
j in a()
0xfff43c
1
i in main()
0xfff440
3
j in main()
0xfff444
4
k in main()
0xfff448
unknown
Register
Value
r0
6
r1
1
r2
unknown
r3
unknown
r4
unknown
sp
0xfff428
fp
0xfff42c
pc
0x1034
Memory
Register
0x1034
ret

Fonksiyon

- Son olarak, bilgisayar 0x1064'e ayarlandığında ve fp tekrar 0xfff448'e ayarlandığında, main()'e geri dönmüş oluyoruz.
- Yığın işaretçisi sekiz bayt açılır ve dönüş değeri k'de depolanır. main() döndüğünde, sistem aşağıdaki gibi görünür:



Fonksiyon

- Bu örnekteki tüm ana noktaları gözden geçirelim;
- **jsr deyimi** dört şey yapar: (pc+4)'ü yığına iter-push, ardından fp'ı, ardından fp'ı sp'ın geçerli değerine ayarlar ve pc'ı verilen prosedürün ilk komutuna ayarlar.
- **ret deyimi**, jsr'nin kurduğu şeyi geri alır: sp'yi fp'nin geçerli değerine ayarlar, ardından fp'yi ve pc'yi yığından çıkarır. Bir yordamın argümanları varsa, jsr çağrısını yapmadan önce bunları yığına ters sırada göndermeniz gerekir.
- Böylece, bir prosedür çağrıldığında, ilk parametre [fp+12] konumundadır. İkinci parametre [fp+16] konumundadır. Üçüncü parametre[fp+20] konumundadır. Ve benzeri…

Fonksiyon

- Bir prosedür çağırana bir değer döndürdüğünde, değeri r0'da saklar.
- Argümanları jsr'yi çağırmadan önce ittiyseniz-push, o zaman jsr çağrısı döndüğünde yapmanız gereken ilk şey bu argümanları yığından çıkarmaktır.
- Şu anda adresleri yığın işaretçisinden küçük veya ona eşit olan değerleri kullanmıyor olsanız da, bu adreslerin değerleri olacaktır ve bu değerler deterministik olabilir.
- Örneğin, son resimde, yığın işaretçisinin üzerinde birçok değer var ve bunların 0xfff42c ile 0xfff43c arasında ne olduğunu biliyoruz çünkü önceki komutların ne yaptığını biliyoruz.

Fonksiyon

- Her prosedürün, parametreleri ve yerel değişkenleri için kendi hafıza konumlarını nasıl elde ettiğini görebilmeniz gerekir.
- main() ve a()'nın her biri i, j ve k kullansa da, bunlar için kendi ayrı bellek konumları vardı.
- Ayrıca, bir prosedür yürütüldüğünde belleğin yerel değişkenler ve parametreler için nasıl ayrıldığını ve prosedür tamamlandığında nasıl kaybolduğunu görmelisiniz.
- Her zaman olduğu gibi, jassem.tcl'yi bu program üzerinde kendi başınıza çalıştırmanızı ve neler olup bittiğini anladığınızdan emin olmanız gerekiyor.

Soru-1

1. Assembly kodu;

```
int a(int i)
{
  int s;
  s=i*i;
return s;
}

int main()
{
  int i,j;
  i=5;
  j = a(i);

return 0;
}
```

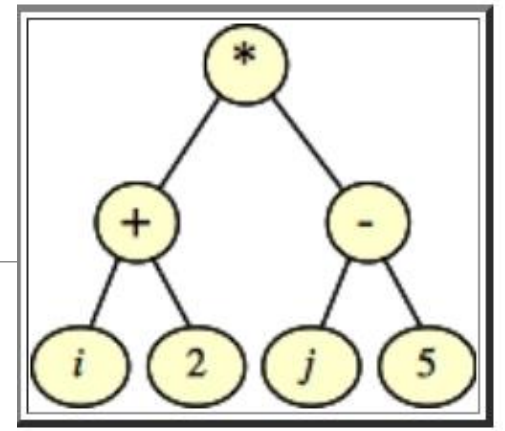
Kayıtçı kaydırma-Register Spilling

- Karar verilmesi gereken önemli bir şey, bir fonksiyonun mevcut değeri hakkında endişelenmeden bir kayıtçıyı-register kullanıp kullanamayacağı (a()'nın r0 ile yaptığı gibi) veya bir fonksiyonun kullanmadan önce kayıtçıyı-register yığına kaydetmesi gerekip gerekmediğidir.
- Bu önemlidir, çünkü örneğin, ana rutinin r3 kayıtçısını kullandığını, ardından "jsr a"yı çağırdığını ve daha sonra r3'ün aynı değere sahip olmasını beklediğini varsayalım.
- Daha sonra a() ve a()'nın çağırdığı tüm fonksiyonlar, r3'ü kullanmadığınızdan emin olmalı veya kullanmadan önce r3'ün değerini kaydetmeli ve bittiğinde onu geri yüklemelidir.
- **Bir fonksiyon çağrısının gövdesinden önce bir kayıtçının değerini kaydetme ve daha sonra geri yükleme işlemine spilling denir.**

Register Spilling

- Farklı makineler ve derleyiciler, Spilling'i farklı şekillerde işler.
- Örneğin, eski **CISC mimarilerinde** bazen bir fonksiyon çağrısının parçası olacak bir spill maskesi bulunur. Bu, hangi kayıtçıların spill edilmesi gerektiğini belirlerdi ve makine aslında sizin için spill işlemini yapardı.
- Makinemizde yaptığımız tipik bir spilling çözümü: Fonksiyonlar, değerleri hakkında endişe duymadan r0 ve r1'i kullanabilir. Ancak, bir fonksiyon bunları kullanıyorsa, r2'den r4'e kadar olan kayıtçıların spill-kaydırılması gerekir .

Register Spilling



- Aritmetik ifadeleri Assembler'da derlemek için onları ağaçlara dönüştürmek faydalıdır. Örneğin;
- Ağacı değerlendirmek için post-order gezinme (en alt sol düğümden yukarı doğru ilerleyerek gezinir.) yapmanız gerekir (veya kenarların yukarıyı gösterdiğini düşünüyorsanız, ağacın topolojik bir sıralamasını yapmanız gerekir).
- Aritmetik, kayıtçı bazında yapılmalıdır, bu nedenle bu düğümlerin her biri bir kayıtçıda olmalıdır.
- Siz (derleyici), komut sıralaması ve ardından, değerlerine artık ihtiyacınız olmadığından emin olmadığınız sürece kayıtçıları yeniden kullanmamak için kayıtlara düğüm ataması yapmalısınız.

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

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

int main()
{
  int i;

  i = a(44, 22);
}
```

Register Spilling

- Örneğin yukarıdaki ifadede önce (i+2) hesabını yaptığınızı ve sonucu r0'da tuttuğunuzu varsayalım.
- O zaman (j-5)’ hesaplamak için r0'ı kullanamazsınız. Bu nedenle r2'yi kullanmak zorunda kalacaksınız ve **r2'yi kullandığınız için yığının üzerine spill** etmek zorunda kalacaksınız.
- Bunu fonksiyonun başında yapıyoruz. **En sonunda, onu yığından geri okuyarak** "unspill" ediyoruz.
- Kod, aşağıda yeniden üretilen spill1.jas dosyasındadır. Bunun üzerinden geçmek için jassem.tcl'yi kullanabilirsiniz.

Register Spilling

- Yerel değişkeni tahsis ettikten sonra r2'yi yığına atmanız gerektiğini unutmayın. Aksi takdirde, k [fp]'de olmayacaktır. Bunu düşünün.

```
a:
    push #4              / Allocate k
    st %r2 -> [sp]--     / Spill r2

    ld [fp+12] -> %r0
    mov #2 -> %r1
    add %r0, %r1 -> %r0  / Calculate (i+2) and put the result in r0

    ld [fp+16] -> %r1
    mov #5 -> %r2
    sub %r1, %r2 -> %r1  / Calculate (j-5) and put the result in r1

    mul %r0, %r1 -> %r0
    st %r0 -> [fp]       / Do k = r0 * r1

    ld [fp] -> %r0
    ld ++[sp] -> %r2     / Unspill r2
    ret
```

```
main:

    push #4              / Allocate i

    mov #22 -> %r0       / Push arguments onto the stack in reverse order
    st %r0 -> [sp]--
    mov #44 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8               / Always pop the arguments off the stack after jsr

    st %r0 -> [fp]
    ret
```

Register Spilling

- a() öğesinin tamamen aynı olduğunu varsayalım. Tek fark, a()'yı iki kez çağırmamız ve dönüş değerlerini toplamamızdır.
- Bunu bir dakikalığına düşünün -- a()'ya yapılan ilk çağrının dönüş değerini nerede saklamanız gerekir?
- Bunu r0 veya r1'de saklayamazsınız çünkü bir fonksiyon çağrısı yapmak onları yok edecektir (bunu varsaymalıyız).
- Bu nedenle, r2 gibi daha yüksek bir kayıtçıda saklamanız gerekir. Sorun çıkmaz, çünkü a(), r2'nin değerinin değişmemesini sağlayacaktır.
- İşte main() kodu (spill2.jas içinde). main()'in r2'yi de spill ettiğini fark edeceksiniz, çünkü herhangi bir fonksiyon r2, r3 veya r4 kullanıyorsa, bunları da spill etmesi gerekir.

Register Spilling

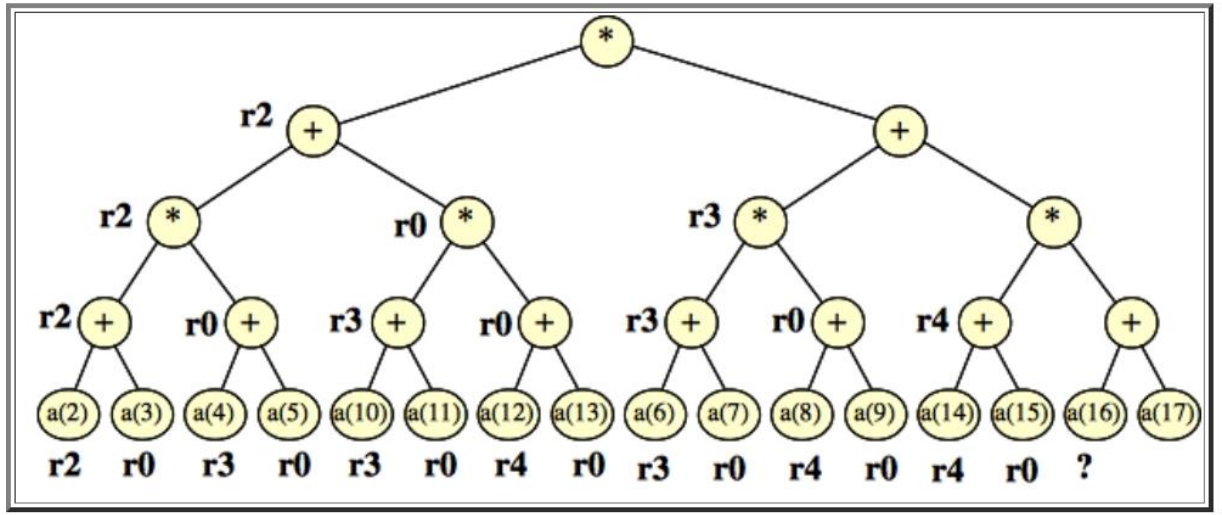
```
int a(int i, int j)
{
  int k;

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

int main()
{
  int i;

  i = (a(10, 20) + a(30, 40));
}
```

```
main:

    push #4                 / Allocate i
    st %r2 -> [sp]--        / Spill r2

    mov #20 -> %r0          / Call a(10, 20) and store the result in r2
    st %r0 -> [sp]--
    mov #10 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    mov %r0 -> %r2

    mov #40 -> %r0          / Call a(30, 40) and add the result to r2
    st %r0 -> [sp]--
    mov #30 -> %r0
    st %r0 -> [sp]--
    jsr a
    pop #8
    add %r0, %r2 -> %r0
    st %r0 -> [fp]

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

Register Spilling

- Kayıtçılarınız bittiğinde ne yaparsınız?

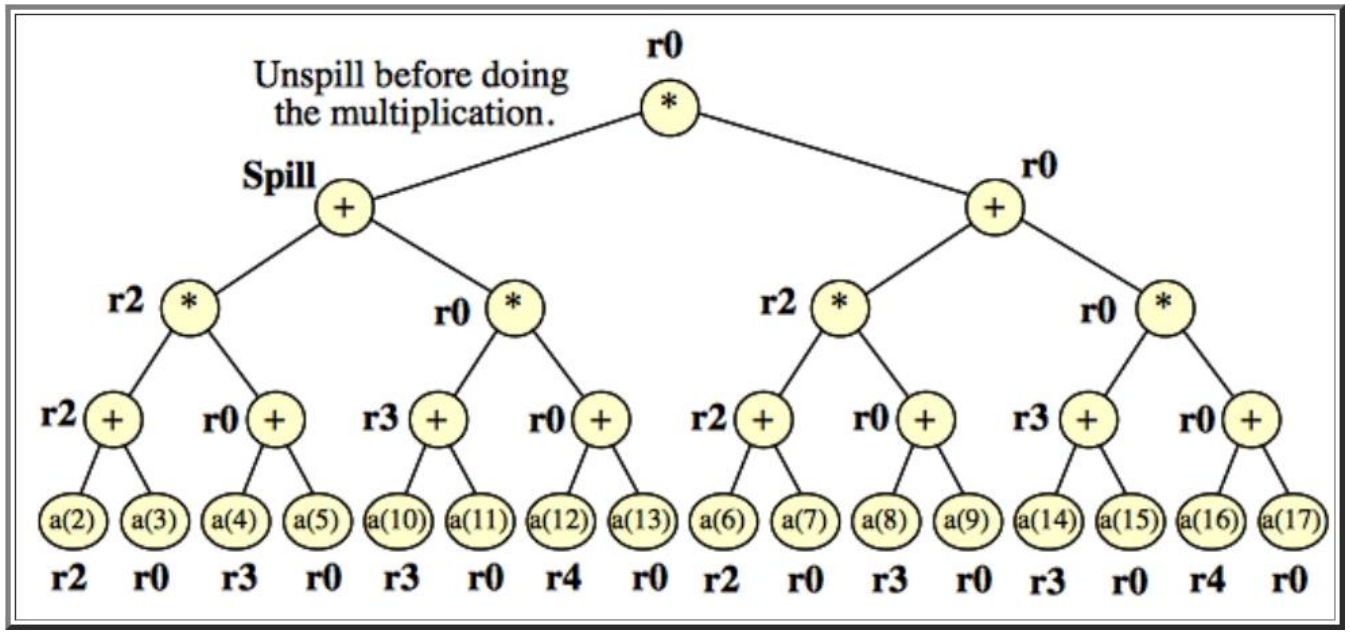
```
int a(int i)
{
  return i+5;
}

int main()
{
  int i;

  i = ( (a(2)+a(3)) * (a(4)+a(5)) + (a(10)+a(11)) * (a(12)+a(13)) ) *
      ( (a(6)+a(7)) * (a(8)+a(9)) + (a(14)+a(15)) * (a(16)+a(17)) );
}
```



- Soldan sağa doğru sıralı hesaplama yaparsanız kullanabileceğiniz kayıtçıları (etiketledi) görebilirsiniz. Kayıtçılarımızın tükendiğini göreceksiniz!

Register Spilling

- Aşağıda bununla nasıl başa çıkacağınızı gösteriyoruz -- "Spill" olarak gösterilen ara değeri spill etmelisiniz.
- Bu, r2'yi tekrar kullanmanıza izin verir ve artık kayıtçılarınız bitmez. Son çarpma işlemini yapmadan önce, değeri bir kayıtçıya boşaltırız.



Register Spilling

- Bunun için assembly kodu: spill3.jas'tadır. Okumak o kadar da zor değil. İşte kritik kod: Toplamın sonucunu (yukarıdaki resimde "spill" ile) yığının üzerine spill ederiz.
- a(13) çağrısıyla başlıyorum. Tamamlandığında, çarpma ve toplama işlemini gerçekleştirirsiniz ve ardından toplama işleminin sonucunu yığına spill ederiz.
- Sonra denklemin sağ tarafı üzerinde çalışmaya başlarsınız (a(6) ile başlayarak):

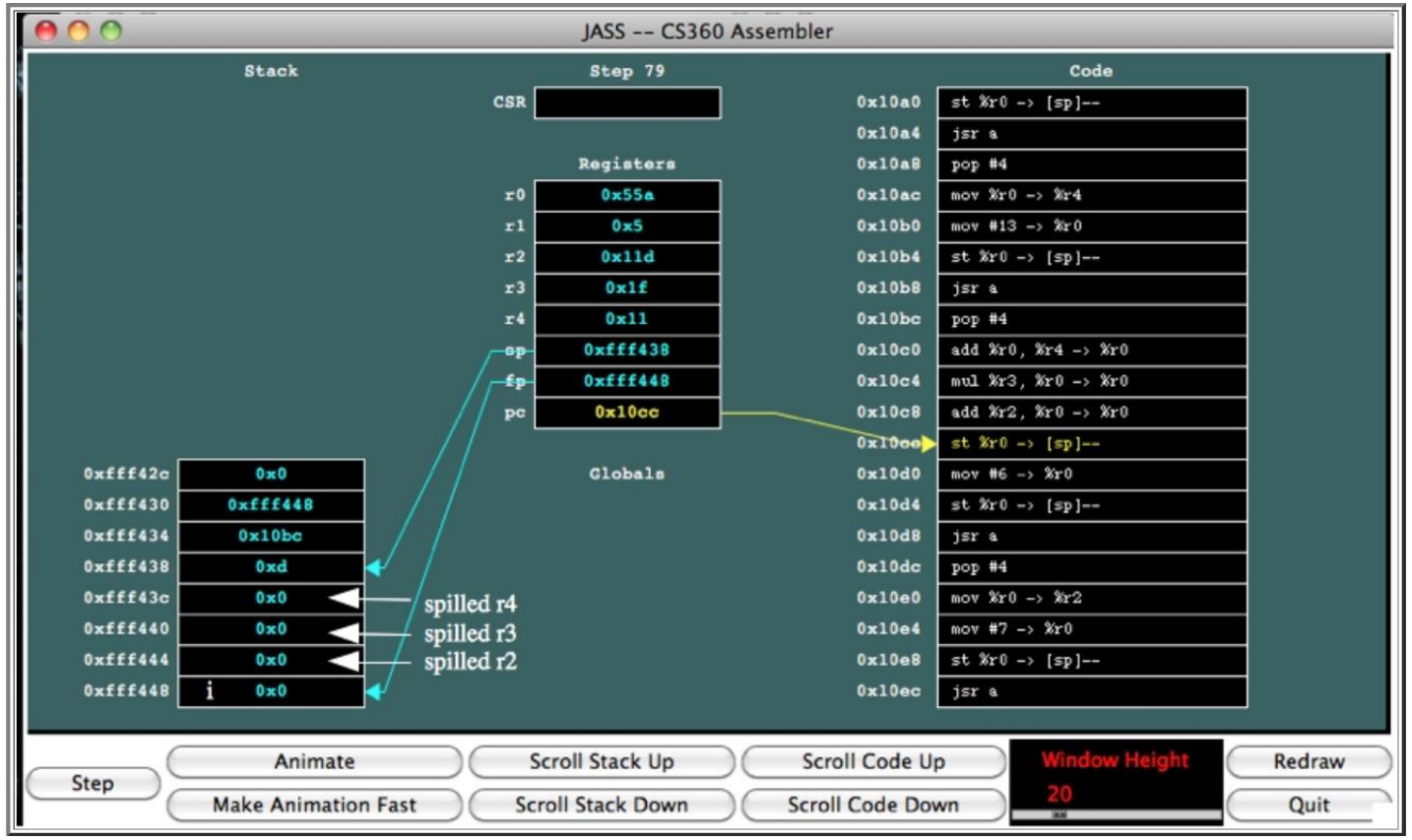
```
...
mov #13 -> %r0
st %r0 -> [sp]--
jsr a
pop #4
add %r0, %r4 -> %r0

mul %r3, %r0 -> %r0   / Multiplication, then Addition, then spill
add %r2, %r0 -> %r0
st %r0 -> [sp]--

mov #6 -> %r0          / a(6)+a(7)
st %r0 -> [sp]--
jsr a
pop #4
mov %r0 -> %r2
...
```

Register Spilling

- Sonunda, a(17) ile işiniz bittiğinde, çarpma ve toplama işlemini yaparsınız. Bir çarpma işleminiz daha var, ancak onun işleneni yığına spill edilen işlenendir. Unspill ederek ve çarpma işlemini gerçekleştirirsiniz.

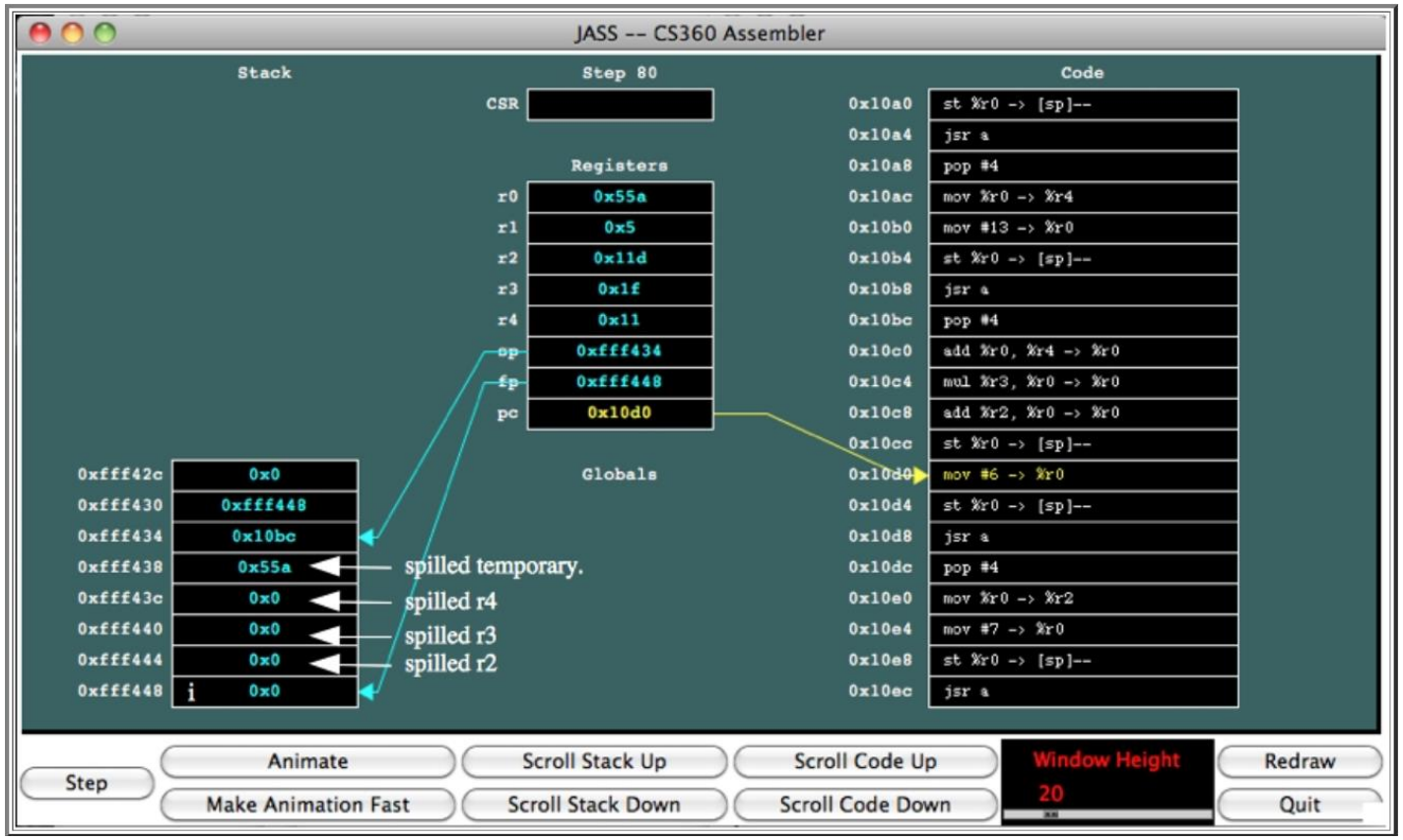
```
...
mov #17 -> %r0
st %r0 -> [sp]--
jsr a
pop #4
add %r0, %r4 -> %r0

mul %r3, %r0 -> %r0    / Multiplication, then addition, then unspill and multiply
add %r2, %r0 -> %r0
ld ++[sp] -> %r1
mul %r0, %r1 -> %r0

st %r0 -> [fp]         / Store the result into i

ld ++[sp] -> %r4       / Unspill before returning
ld ++[sp] -> %r3
ld ++[sp] -> %r2
ret
```

Register Spilling



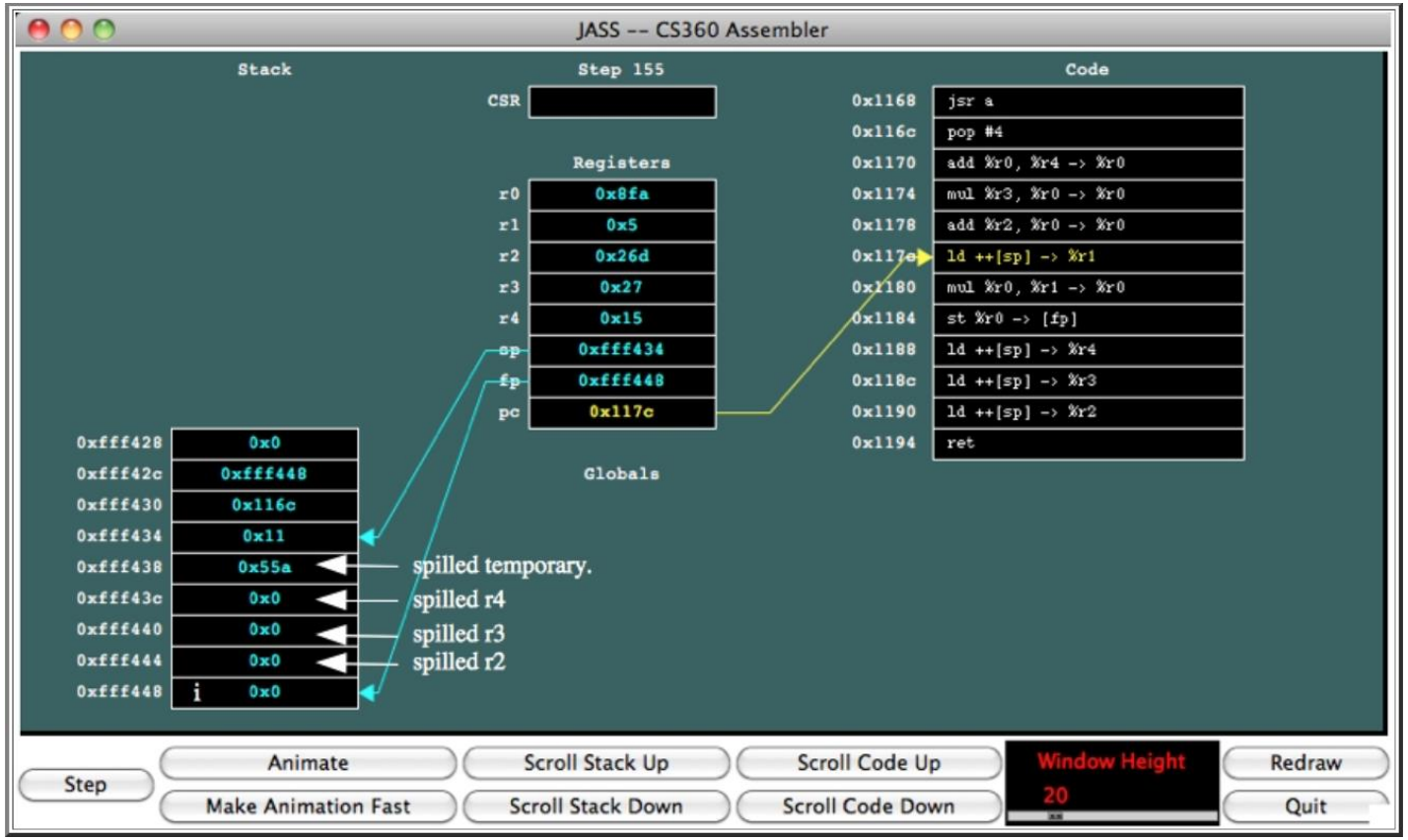
Kendiniz tekrar kontrol edebilirsiniz
-- a(i) sadece i'ye 5 ekler, yani:
r4 (12+5) = 17 = 0x11'e eşit olmalıdır.
r3, (10+5+11+5) = 31 = 0x1f'ye eşit olmalıdır.
r2, (2+5+3+5)*(4+5+5+5) = 15*19 = 285 = 0x11d

Register Spilling



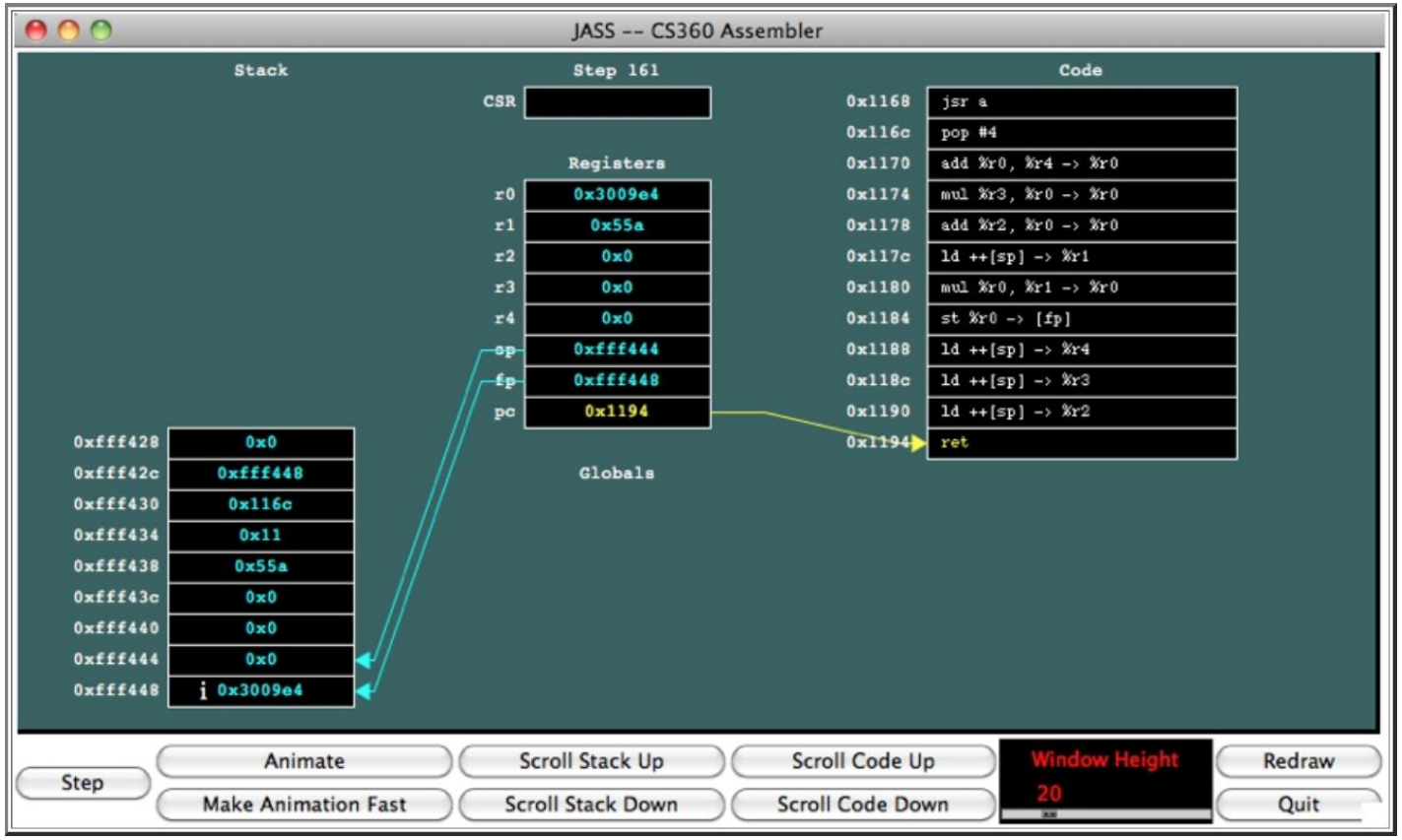
Spill’den sonra, 0x55a yığına gider:

Register Spilling



"Unspill" adım adım devam ediyoruz:

Register Spilling



Ve sonunda i, 0x3009e4 = 3148620 olarak ayarlandı.

Pointers

- Basit İşaretçi referans kaldırma.
- Çerçeve işaretçisinden erişilecek olan üç yerel değişken vardır:
- **i** (fp-8) konumunda . Bu nedenle, &i eşittir (fp-8) ve i'nin değeri [fp-8]'e yüklenecek ve saklanacaktır.
- Benzer şekilde, &j eşittir (fp-4) ve j'nin değeri [fp-4]'e yüklenecek ve saklanacaktır.
- Son olarak, &jp, fp'ye eşittir ve jp'nin değeri yüklenecek ve [fp]'de saklanacaktır.
- *jp istiyorsanız, [fp]'yi bir kayıtçıya yüklemeniz ve kayıtçının refenransını kaldırmanız gerekir.

pointer1.c:

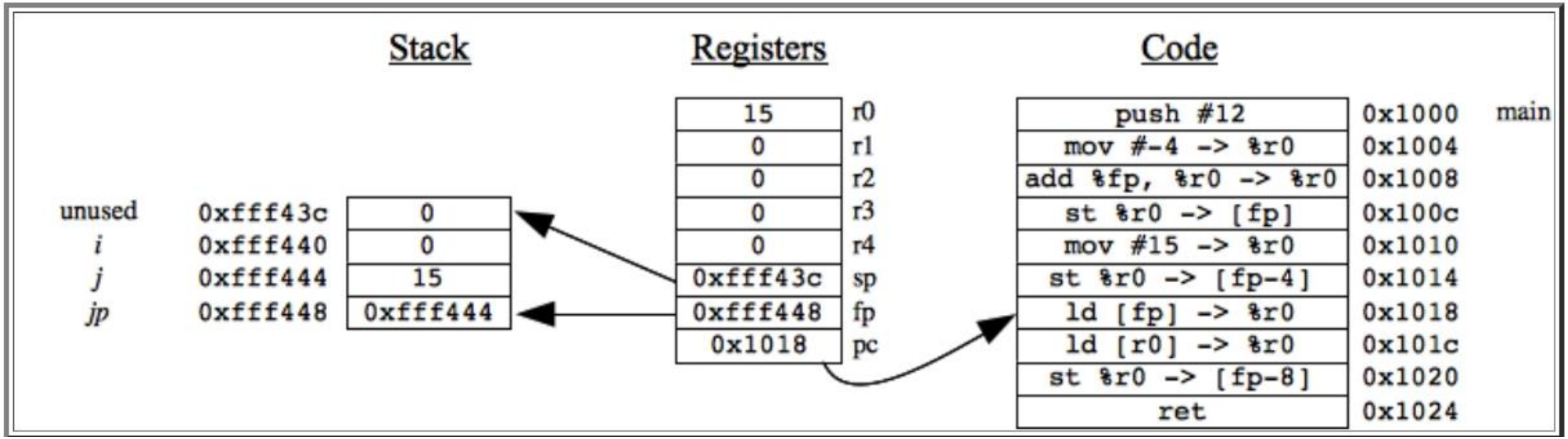
```
int main()
{
  int i, j, *jp;

  jp = &j;
  j = 15;
  i = *jp;
}
```

pointer1.jas:

```
main:
    push #12                / Allocate the three locals

    mov #-4 -> %r0          / jp = &j.
    add %fp, %r0 -> %r0
    st %r0 -> [fp]

    mov #15 -> %r0          / j = 15
    st %r0 -> [fp-4]

    ld [fp] -> %r0          / i = *jp
    ld [r0] -> %r0
    st %r0 -> [fp-8]

    ret
```

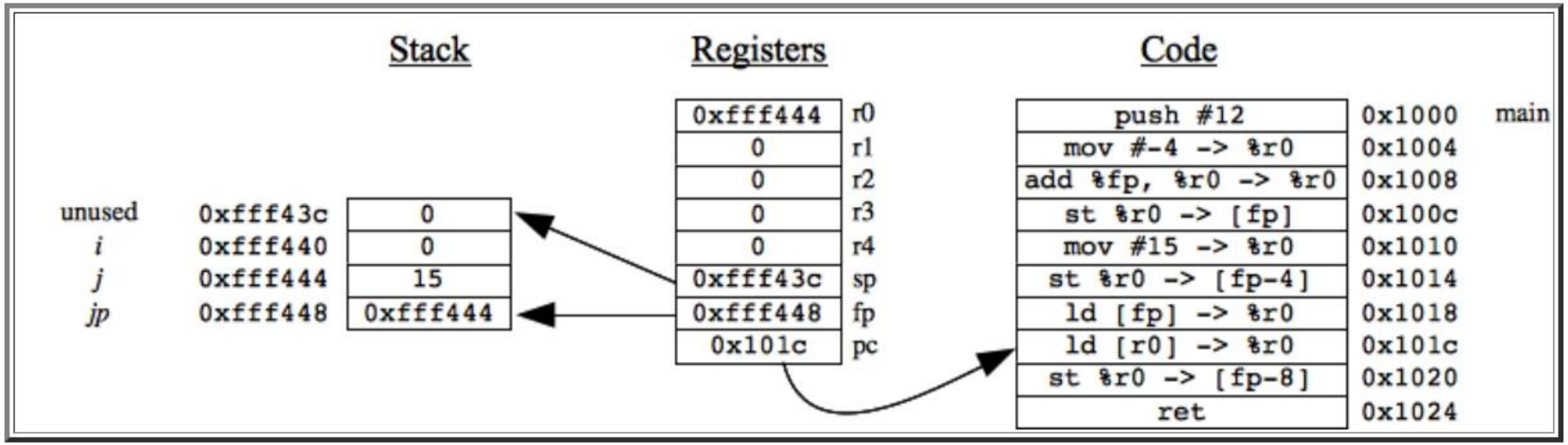
Pointers

- İşte programın önemli bir parçası olduğunu düşündüğüm şeyin bir kopyası -- "i = *jp" yapmaya başladığınızda:



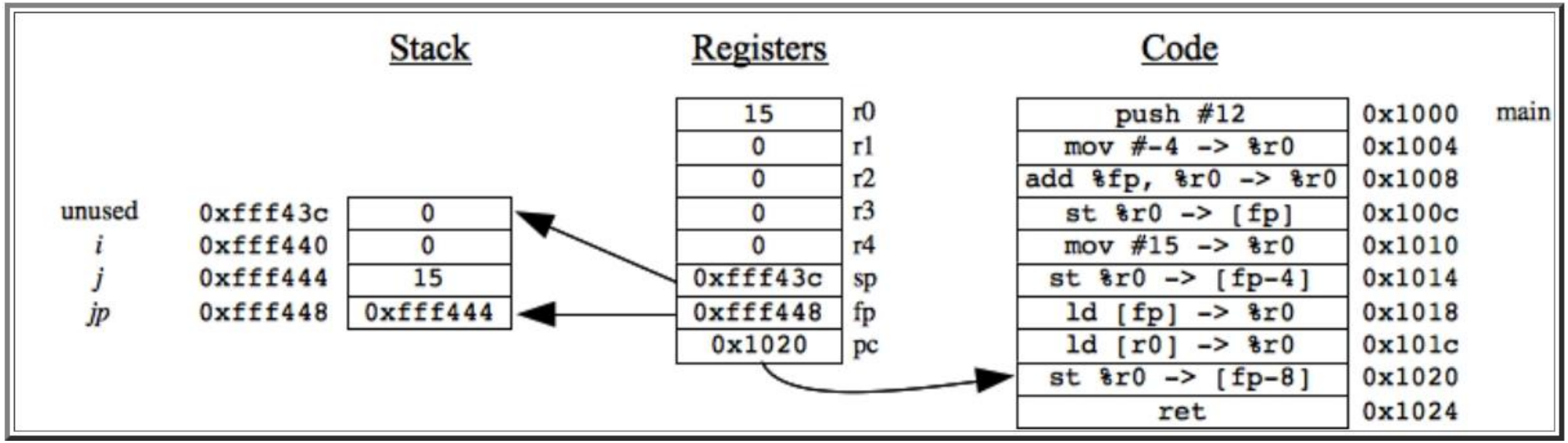
Pointers

- &j'yi manuel olarak 0xfff444 olarak hesaplamak zorunda kaldık. Bu değeri jp = [fp] içinde görebilirsiniz. *jp'yi hesaplamak için önce jp'yi r0'a yüklüyoruz:



Pointers

- Ve sonra [r0] yüklüyoruz: bu, 0xfff444'teki değeri alıyor, yani 15 (jassem'de bunu 0xf olarak görürsünüz, çünkü jassem her şeyi onaltılık-hexadecimal olarak yapar):



- Son olarak, 15, **i**'de depolanır (konum 0xfff440).

Pointers

- main()'de **&i** (fp-4)'tür ve i'nin değeri [fp-4]'tür.
- main()'de **&j**, fp'dir ve j'nin değeri [fp]'dir.
- a()'da &p (fp+12)'dir ve p'nin değeri [fp+12]'dir.
- *p elde etmek için [fp+12]'yi bir kayda yüklemeniz ve kayıtçının referansını kaldırmanız gerekir.

pointer2.c:

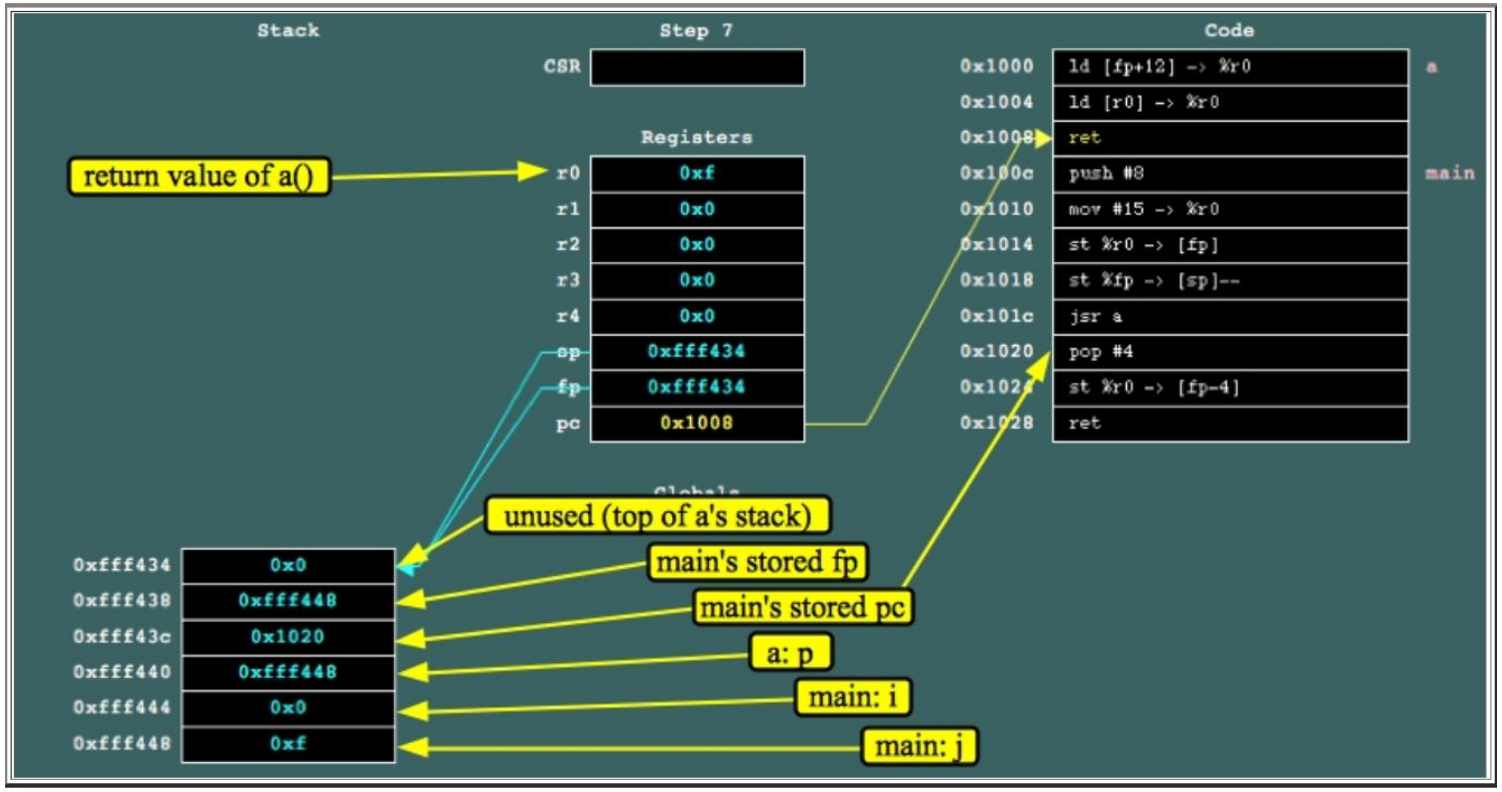
```
int a(int *p)
{
  return *p;
}

int main()
{
  int i, j;

  j = 15;
  i = a(&j);
}
```

pointer2.jas:

```
a:
    ld [fp+12] -> %r0       / get p's value
    ld [r0] -> %r0          / dereference it
    ret

main:
    push #8

    mov #15 -> %r0          / j = 15
    st %r0 -> [fp]

    st %fp -> [sp]--        / push &j on the stack
    jsr a                   / and call a()
    pop #4
    st %r0 -> [fp-4]

    ret
```

Pointers

- Yığındaki her değeri tanımlayabildiğinizden emin olmamız gerekli. Bunu, kod a() için ret deyimindeyken yapıyoruz:



Dizi Referans kaldırma

- Dizi başvurusunu kaldırma, işaretçi başvurusunu kaldırmaya çok benzer.
- Dizi indeksini öğenin boyutuyla çarparsınız, ardından onu dizinin en üstüne eklersiniz.
- Ardından bu değeri kaldırın. Örneğin, pointer3.c'ye bakın:

pointer3.c:

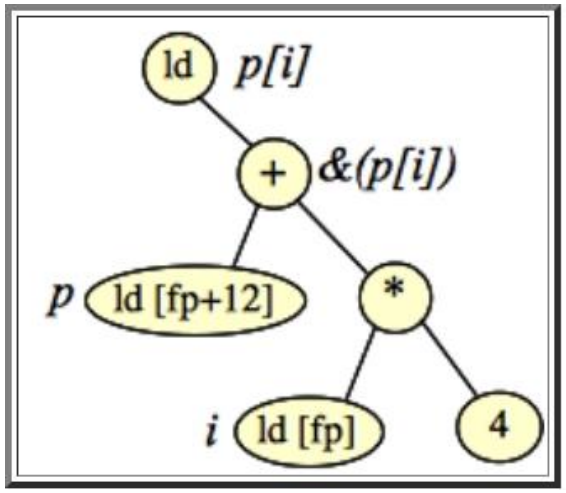
```
void a(int *p)
{
  int i;

  i = p[0];
  i = p[3];
  i = p[i];
}

int main()
{
  int array[5];

  array[0] = 10;
  array[1] = 11;
  array[2] = 12;
  array[3] = 2;
  array[4] = 15;

  a(array);
}
```

Dizi

- Şimdilik main() hakkında endişelenmeyelim. Sadece belleği ayarlamak için main()'i kullanıyoruz böylelikle **a fonksiyonunun** assembler'ını takip edebilirsiniz.
- Yine, Assmbler'da bulmayı kolaylaştırmak için değişkenlerimizin adreslerini ve değerlerini bulalım:
- &i fp'dir ve i'nin değeri [fp]'dir.
- &p (fp+12)'dir ve p'nin değeri [fp+12]'dir.
- p[0]'ı elde etmek için, [fp+12]'yi yüklememiz ve referansını kaldırmamız gerekiyor.
- p[3]'ü elde etmek için, [fp+12]'yi yüklememiz, ardından buna 12 eklememiz (3 * bir tamsayının boyutu) ve ardından sonucu kaldırmamız gerekir.
- p[i]'yi elde etmek için aşağıdaki değeri hesaplamamız gerekir:[fp+12] + 4 * [fp]

Dizi

- Ve sonra referansı kaldırabiliriz. Devam edelim ve referanssızlaştırmayı denkleme koyalım:
  - [[fp+12] + 4 * [fp]]
- Son derste denklemlerle yaptığımız gibi, bunu bir ağaca çevirelim:



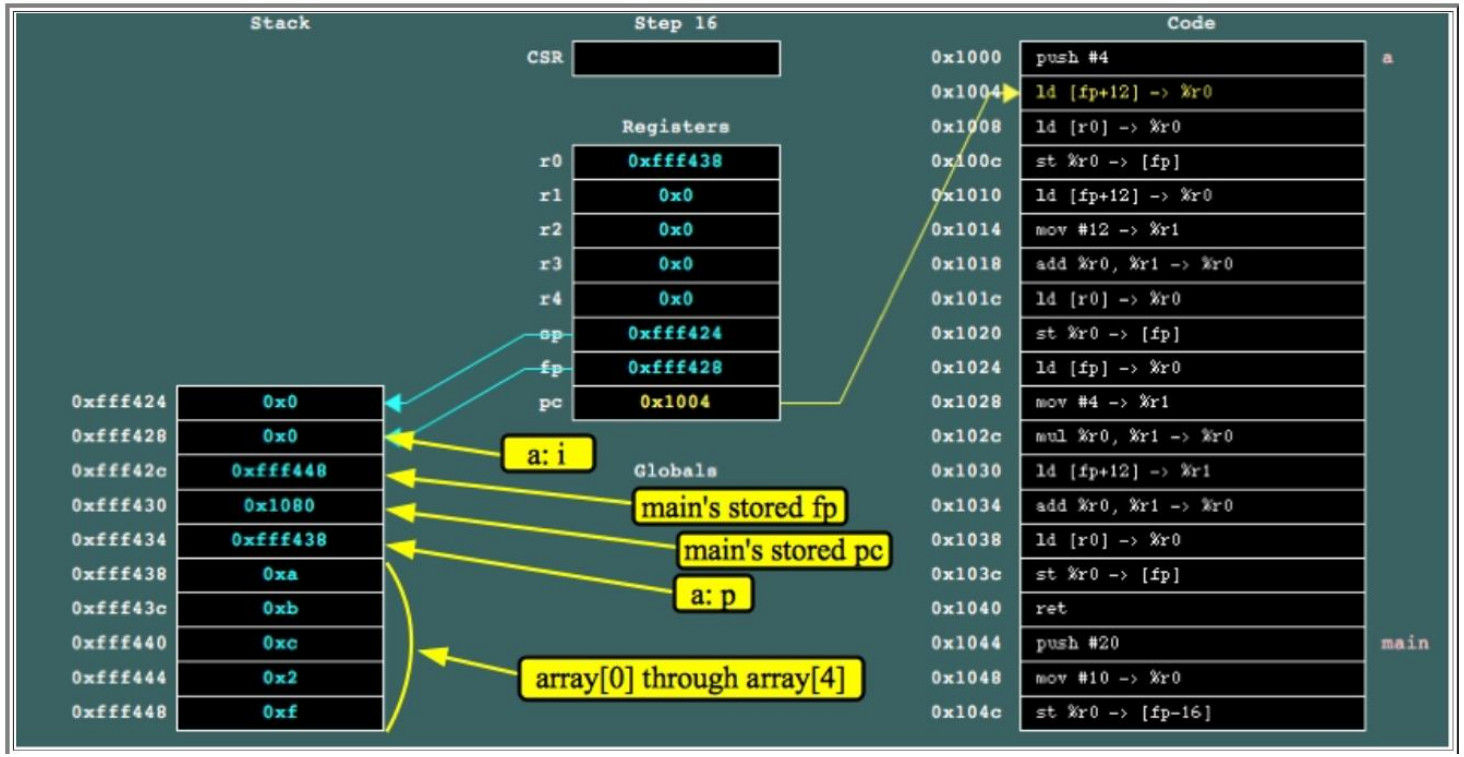
Dizi

- &(p[i])'yi nasıl oluşturduğumuzu ve sonra onu nasıl kaldırdığımızı görebilmeniz için düğümleri etiketledik . İşte derleyici (pointer3.jas'ta):

```
a:
   push #4

   ld [fp+12] -> %r0      / i = p[0]
   ld [r0] -> %r0
   st %r0 -> [fp]

   ld [fp+12] -> %r0      / i = p[3]
   mov #12 -> %r1
   add %r0, %r1 -> %r0
   ld [r0] -> %r0
   st %r0 -> [fp]

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

   ret
```

Dizi

- Yine, şimdilik main()'i görmezden geliyoruz. "Push #4"ten hemen sonra a() çalıştırana kadar jassem'de ilerleyelim. Yine, yığındaki her şeyi tanımlamak yararlıdır:



Dizi

- p[0]'ın 10'a (0xa), p[3]'ün 2'ye eşit olduğunu ve i'nin 2'ye ayarlandığı için p[i]'nin 12'ye (0xc) eşit olduğunu görmelisiniz.
- Şimdi main() hakkında düşünelim. Yapacağı ilk şey, dizinin beş tamsayısını tahsis etmek için push #20'yi çağırmak olacaktır. Bundan sonra, derleyici şunu bilir:
- &(dizi[0]) (fp-16) ve dizi[0]'nin değeri [fp-16]'dır.
- &(dizi[1]) (fp-12)'dir ve dizi[1]'in değeri [fp-12]'dir.
- &(dizi[2]) (fp-8) ve dizi[2]'nin değeri [fp-8]'dir.
- &(dizi[3]) (fp-4) ve dizi[3]'ün değeri [fp-4]'tür.
- &(dizi[4]) fp'dir ve dizi[4]'ün değeri [fp]'dir.

Dizi

- Dikkat edeceksiniz -- "array" için ayrılan bir bellek yoktur.
- Derleyici, "array"in (fp-16)'ya eşit olduğunu bilir -- bu, dizi[0] için bir işaretçidir.
- Bu bilgiyle donanmış olarak, dizinin öğelerini ayarlamak basittir.
- **a(array)** çağırmak biraz daha zordur, ama işte pointer3.jas'ın geri kalanı:
- a(array)'yi çağırmak için (fp-16)'yı hesaplamalı ve bunu yığına göndermeliyiz.
- Bu, "mov #-16 -> %r0" ile başlayan üç satırda yapılır.

```
main:
   push #20

   mov #10 -> %r0          / Store the values of array
   st %r0 -> [fp-16]
   mov #11 -> %r0
   st %r0 -> [fp-12]
   mov #12 -> %r0
   st %r0 -> [fp-8]
   mov #2 -> %r0
   st %r0 -> [fp-4]
   mov #15 -> %r0
   st %r0 -> [fp]

   mov #-16 -> %r0          / Push array onto the stack
   add %fp, %r0 -> %r0
   st %r0 -> [sp]--
   jsr a                    / call a
   pop #4
   ret
```