Bilgisayar organizasyonu ve yığın çerçeveleri üzerinde teorik bir başlangıçtır burda anlatılanlar. Büyük ihtimalle kafa karıştırıcı bir ders olduğunu biliyorum, ama bunlar bazı ince şeyler.

Kaydediciler

Biz genel bir bilgisayar mimarisi varsayacağız , ve bu varsayacağımız bilgisayar mimarisi çoğu makineden farklı olacaktır fakat , bu hemen hemen tüm "uniprocessors" için örnek teşkil eder.

Makinemizde 8 tane genel amaçlı kaydediciler olduğu varsayacağız.Bunların hepsi 4 byte ve kullanıcı tarafından okunabilir ya da yazılabilir olabilir.İlk 5 kaydedicimizin ismi sırasıyla "**r0**, **r1**, **r2**, **r3**, **r4**"'dır.Bundan sonra gelen son 3 kaydedicimizin ismi özeldir:

- 6.kaydedicimizin ismi **sp** ve *"stack pointer"* diye çağrılır.(Yığın pointerı diyebilirz.)
- 7.kaydedicimizin ismi **fp** ve *"frame pointer"* diye çağrılır.
- 8.kaydedicimizin ismi **pc** ve **"***program counter"* diye çağrılır.(Program sayacı diyebilirz)

Buna ek olarak,bilgisayar her zaman aynı değerleri içeren 3 tane hem okur-yazar kaydediciye sahiptir:

- **g0,** değeri her zaman 0'dır.
- **g1,** değeri her zaman 1'dir.
- **gml,** değeri her zaman negatif sayılardır.

Son olarak,bilgisayar kullanıcı tarafından direkt bağlanamayan ayrıca 2 tane özel kaydediciye sahiptir.

- **IR** => Komut kaydedicisidir. Bu kaydedici şu anda yürütülmekte olan komutları tutar.
- **CSR** =>Durum kontrol kaydedicisidir.Şu an ki ve önceki icra edilebilir komutlar ile ilgili biligileri içerir.

Komut Dizisi(Instruction Cycle)

Bilgisayarınızın çalımasını tekrar tekrar çalışan komutlar oluşturur.Bu komut dizisi olarak bilinir. Komut dizisi 4 genel aşamadan oluşur:

- 1. Kod Çözümü (in **IR**)
- 2.Komut çalıştırma
- 3. Sonraki komutları belirleme
- 4. **IR** içine sonraki komutu yükleme

Komut nedir? Diğer her şey gibi, bunda da 0 ve 1 den oluşan bir dizi var. Biz tüm talimatları 32 bit (o naif bir varsayım var olmasına rağmen, bizim amacımız için çalışacak) olduğunu

varsayıyoruz.Komutlar,program hafızasının bir bölümünde depolanır ve Program counter kaydedicisi ile işaret edilen komut yürütme için IR içine yüklenen biridir. PC değeri 0x2040 içeriyorsa, diğer bir deyişle, daha sonra IR bellek adres 0x2040 başlayarak 4 byte içerdiği komut yürütülmektedir.*Assembly kod* is bir okunabilir kodlanan bir kodtur. Bir program çağırıldığında bi assembler tarafından 1 ve 0 'lardan oluşan bir assembly koduna çevrilir.If Eğer **gcc -S** ile çağırırsak, bu assembler içeren bir .s dosyası üretecek C programlama için. -S bayrağı olmazsa doğrudan üretilir.

(Bir Linux makine üzerinde deneyin - bu Solaris makineleri daha bunun gibi çok daha fazla assembler üretir).

Komutlar

1. Hafıza <-> Kaydedici Komutları:

ld mem -> %reg Hafızadan kaydediciye değer yükler.

st %reg -> mem Kaydediciden hafızaya veri depolar.

Adres hafızasına giden birkaç yol vardır :

st %r0 -> i Kaydedicinin r0 değerini hafızanın global olarak tanımlanan i bölgesine saklar.

st %r0 -> [r1] r1 kaydedicisinde ki değer için hafızada bir yer ayırır ve bunu hafızanın bi bölümünde r0'da saklar.

st %r0 -> [fp+4] Bir bellek bölgesi çerçeve işaretçisi değeri için bi adres ata, ve 4 byte o konuma sonra hafıza konumuna r0 değerini saklar ve bu bellek bölgesinden sonra hafızada 4 bytelık yer kaplayarak r0 değerini saklar.4'ten başka hiçbir değer kullanamazsın pozitif ya da negatif,sadece 4.Ancak ,bir kaydedici kullanamazsınız. (örneğin : st %r0 -> [fp+r2] bunu kullanamazsınız r2 kaydedicisi kullanılmış).Bu sadece frame pointer ile çalışır.Herhangi bir kaydedici ile çalışmaz.

st %r0 -> [sp]-- Hafızadan sp kaydedicisi değeri için bir adres ata,ve bu hafıza bölgesinde r0'ın değerini saklar ve daha sonra sp değerinden 4 çıkarır

st %r0 -> ++[sp] sp kaydedicisi değeri için bi adres atar.İlk olarak bu değere 4 ekler sonra r0'ın değerini bu bölgede saklar.

2. Kaydedici <-> Kaydedici Komutları

mov %reg -> %reg Kaydedicinin değerini baka kaydediciye kopyalar.

mov #val -> %reg kaydedici, ya da bir sabite değeri set'leyin

Kaydediciden kaydediye bütün aritmetik komutlar:

add %reg1, %reg2 -> %reg3 reg1 ve reg2 iyi toplayarak reg3'e yazar.
sub %reg1, %reg2 -> %reg3 reg1'den reg2yi çıkarır reg3'e yazar.
mul %reg1, %reg2 -> %reg3 reg1 ve reg2 yi çarpar reg3'e yazar.
idiv %reg1, %reg2 -> %reg3 reg1'i reg2'ye böler.bu bölme tamsayı bölmedir.reg3'e yazar.
imod %reg1, %reg2 -> %reg3 reg1 'i reg2'ye göre mod'unu alır.

Yığın işaretçisi(stack pointer) üzerinde toplama ve çıkarma yapan iki özel komut vardır:

push %reg

Bu push komutu yığından eleman çıkarır.

push #val

pop %reg Pop komutu ise yığına eleman ekler.
pop #val

3. Kontrol Komutları

jsr a a komutuyla başlayan altprogramı çağırır.
Ret altprogramdan geri döner.

Ayrıca “for” ve “if” durumları için uygulayabileceğiniz “compare” ve “branch” komutları da mevcut ancak şimdilik üzerinde durmayacağım.

Son olarak, aslında kod olmayan direktifler de vardır ama belirtilmelidir ki bu bellek değişkenler için tahsis edilmiştir. Örneğin;

.globl i i değişkeni için globals segmentinde 4 byte yer ayırır.

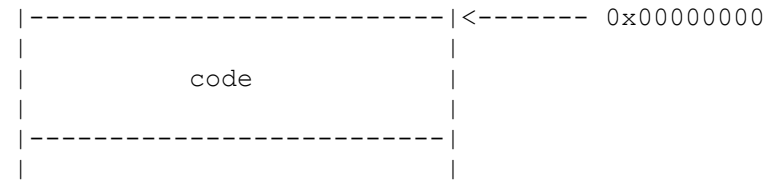
Program sayacı komut kayıt değeri yüklemek için nereye gidilmesi gerektiğini işaret eder. Normal komutlarda program sayacı 4 artırılır, böylece bir sonraki komut görülebilir. Kontrol komutlarında ise program sayacı makinenin alt programları çağırabileceği ve if-then durumlarını uygulayabileceği yeni bir değer alır.

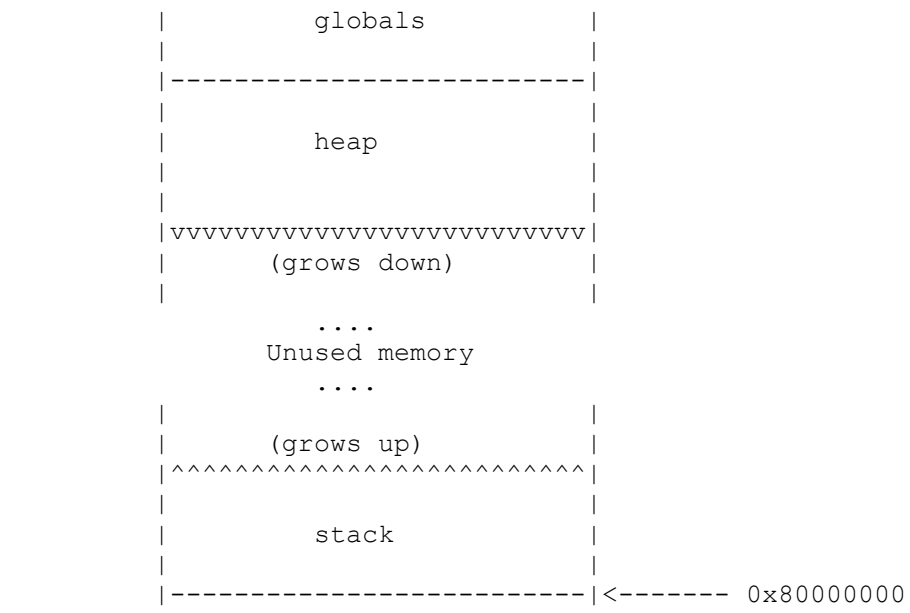
Adres Alanı

Her program hafızasında bir adres alanı görür. Tipik bir adres alanı 4 parçaya bölünmüştür; kod, globaller, yığın ve yığıt. Kodlar herhangi bir şey tutmaz ancak komutlar tutar. Globaller, global değişkenlerin saklandığı yerde saklanırlar. Yığın için ise malloc fonksiyonu yer tahsis eder. Yığıt yerel değişkenler ve argümanlar gibi geçici depolamalar içindir.

Genellikle “process treats memory” bayt olarak büyük boyutlu diziler gibi davranır, fakat baytlar 4 baytlık birimler halinde mantıksal olarak düzenlenir. Biz bu bellek boyutunun 0x80000000(hex) olduğunu varsayalım. Sparcs gibi bazı makineler bunu 0x100000000 şeklinde alır fakat biz burada küçük boyutları ele alacağız. Kod 0 adresinden başlar(genelde daha büyük adreslerden başlar ancak biz daha basit olması açısından 0’ı kullanacağız). Globaller kodları, yığın ise globalleri takip eder. Unutulmamalı ki program çalışırken yığın ve yığıt büyüyüp küçülebilirken globaller sabit kalır. Yığıt 0x8000000(aslında 0x7fffffff) adresinden başlayarak arkadan öne doğru büyür. Yığın ve yığıt arasındaki kullanılmayan bellek:

Programın Adres Alanı:





Basit Derlenmiş Kod

C derleyicisi C kodu alır ve talimatlar haline çevirir.Bunu ne yaptığımızı ve bu çevirinin nasıl çalıştığını aşağıdaki dersler gösteriyor. Çeviri ile üretilen toplayıcı kodu, makine talimatları ve direktiflerinden oluşur.Tercüme çok uygun.Örneğin, aşağıdaki kod:

```
int i;
int j;
main()
{
 i = 1;
 j = 2;
 j = i + j;
}
```

Aşağıdaki assembler kodu derlenerek:

```
        .globl i
        .globl j
  main:
        mov #1 -> %r0           / i = 1
        st  %r0 -> i
        mov #2  -> %r0          / j = 2
        st  %r0 -> j
        ld  i   -> %r0          / j = i + j
        ld  j   -> %r1
        add %r0,%r1  -> %r1
     st  %r1 -> j
     ret
```

Bu kod oldukça basittir. C de her komuta, assembler da karşılık gelen talimatlar dizisi vardır. Derleyici akıllı olmadıkça, verimsiz kod üretecek.Örneğin muhtemelen onu görebilirsiniz:

```
        .globl i
        .globl j
  main:
        mov #1  -> %r0
        mov #2  -> %r1
        add %r0,%r1  -> %r1
     st  %r1 -> j
        st  %r0 -> i
     ret
```

İyi ye yakın çalışır ve daha az talimatlara sahiptir.Eğer 0 bayrağı ile gcc optimize etmek için çalıştırırsan kodunun onun daha az talimatları olur.Ancak, normalde, gcc sadece basit, optimize olmayan kod üretir.

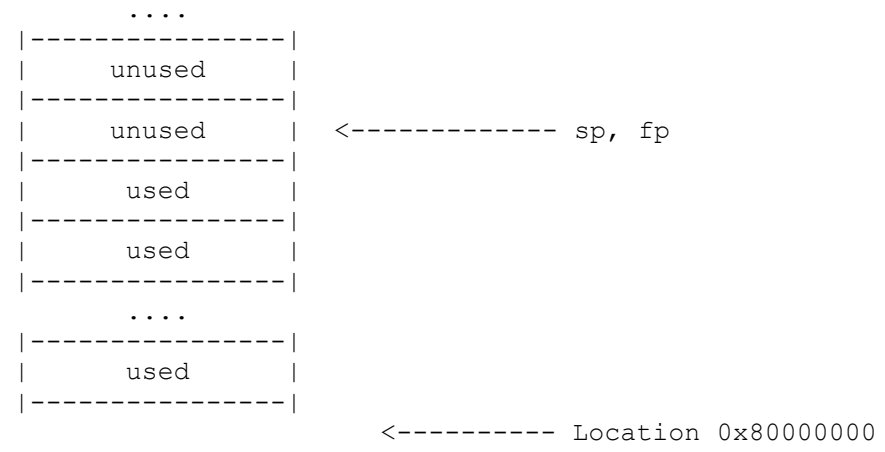
Şimdi aşağıdaki kodu çalıştırmak için varsayalım:

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

  i = 1;
  j = 2;
  j = i + j;
}
```

I ve j yerel değişkenler olduğundan,geçici depolamadan gelmelidirler: yığından gelmelidir. Yığın nasıl çalışır ? Bu sp ve fp kaydedicisi tarafından yönetilir. sp ve fp atanmış yığın üzerinde bir çerçeve olarak bilinen nedir? Çerçevenin alt fp noktaları ve üst sp noktaları. Yığın işaretçisi yukardaki tüm bellek konumlarını kullanılmayan olarak kabul eder. Böylece mevcut yığın çerçevesi içine bellek konumları koyarak sp eksiltiminden yeni geçici hafıza alabiliriz.

Örneğin bir prosedür ilk çağırıldığında bu iki kaydedici yığında aynı yere işaret eder. Çerçeve boş kabul edilir.

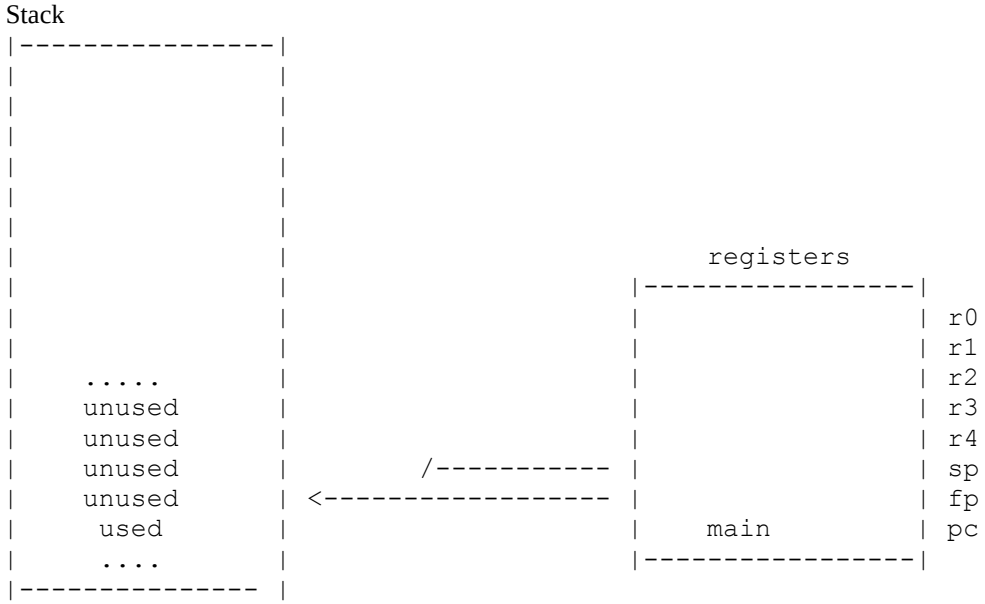


İki yerel değişken i ve j için 8. tarafından oda tahsis etmek, yığın işaretçisini bir azaltırız.Bu, mevcut yığın çerçevesini 4-baytlık miktarlarda ikiye ayırır. Geleneksel olarak alttaki j yi ve üstteki i yi arayacağız.Bunuderleyici tanımlar.Biz kolayca alttaki i ve üstteki j yi çağırmış olabilirdik:

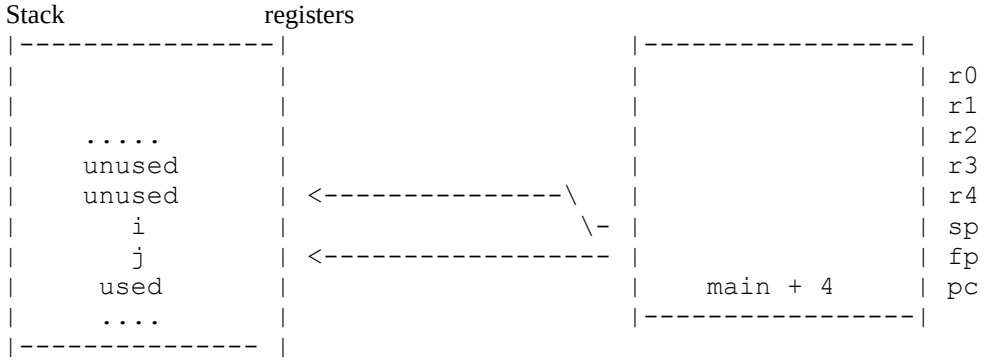
Şimdi, ana kod için hemen önceki kod gibi yalnızca global değişkenler olarak i ve j ye erişim yerine çerçeve işaretçi için uzaklıklar olarak onlara erişiriz.

```
main:
        push #8             / This allocates i and j
        mov #1   -> %r0
        st  %r0 -> [fp-4]       / Set i to 1
        mov #2  -> %r0
        st  %r0 -s> [fp]        / Set j to 2
        ld  [fp-4]  -> %r0
        ld  [fp]    -> %r1
  add %r0,%r1 -> %r1       / Add i and j and put the result
        st  %r1 -> [fp]         / back into j
        ret
```

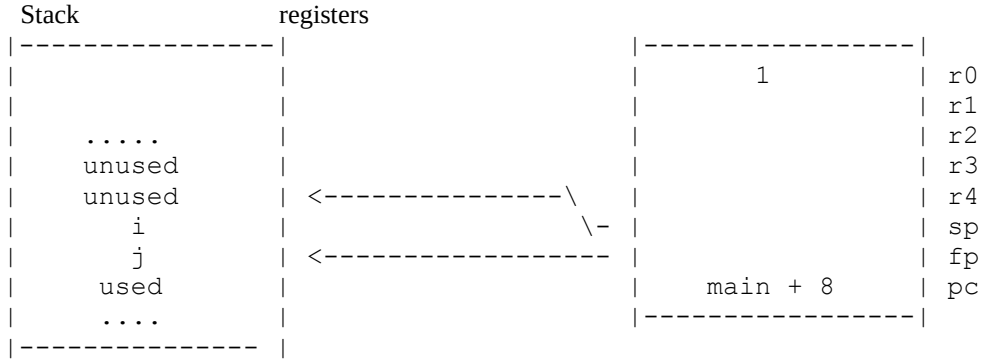
Ana fonksiyon yürütüldüğünde ne olur görelim.



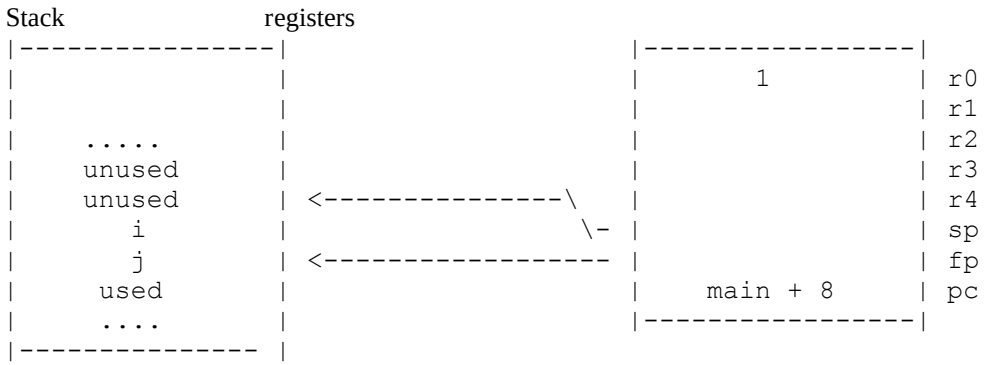
Not olarak bu fp ve sp noktası boş yığın çerçevesinin tabanını işaret eder.main rutin başlangıcında pc noktalarıdır.Bu yönerge push#8. Bu yürütme yapıldığında şu oluşur:



Alan i ve j için geçerli yığın çerçeve üzerinde tahsis edilmiştir ve pc arttırılır.Bu talimatları işaret eder mov 1 -> %r0 Bu da, aşağıdaki makinehaline koyar.

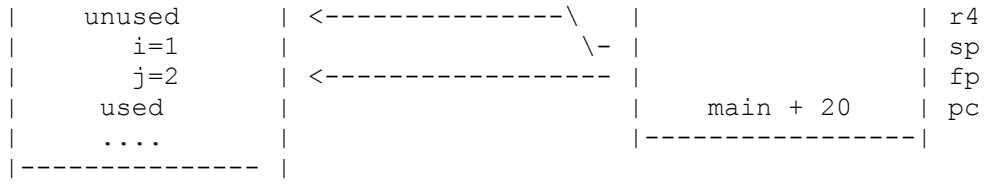


Şimdi, pc noktaları "st %r0 -> [fp-4] yapıldığında, i için konum değeri olarak 1 ayarlanır.



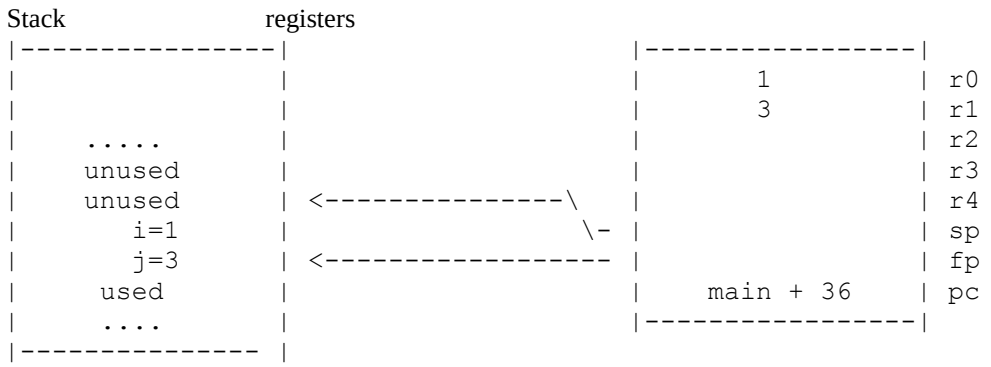
Sonra, sonraki iki talimat makinenin durumu olacaktır:

mov 2 -> %r0
st %r0 -> [fp]



Son olarak son 4 talimatı yap:

```
ld  [fp-4]  -> %r0
ld  [fp]    -> %r1
add %r0,%r1 -> %r1
st  %r1     -> [fp]
```



Jassem - Görsel Assembler

Assembler anlamanıza yardımcı olmak için size assembler kodu programları yükleyenve onlarıadım adım sağlayan basit görsel assembler yazdık. Bu grafik tcl/tk kodlama dilinde yazılmıştır.

/ blugreen/homes/plank/cs360/bin/jassem.tcl ya da

http://www.cs.utk.edu/~plank/plank/classes/cs360/360/bin/jassem.tcl adresinden yerel olarak bu dosyayı alabilirsiniz.

Tcl / tk ile ilgili güzel bir nokta, Unix, Windows ve Macintosh üzerinde çalışmasıdır. Kullandığımız makinelerde jassem.tcl kullanmak için sadece programı çalıştırın:

UNIX> wish ~plank/cs360/bin/jassem.tcl

İstek tüm makinelerin üzerine monte edilmelidir.

Farklı olarak Windows veya Macintosh makinede jassem.tcl kullanmak için tcl / tk yüklemeniz gerekir.

Bu cod ücretsiz olarak www.scriptics.com adresinden alabilirsiniz.

Şimdi program adım adım gidebilirsiniz ve işleme boyunca herşeye bakabilirsiniz.Baştan sona gittiğiniz gibi her adımda anladığınızdan emin olun. Bu çok faydalı bir araçtır.

Ek

Bir sınıftaki 360 öğreciden gelen sorular üzerine birçok öğrencinin bu soruları sorguladığını öğrendiğimden beri cevapları herkese yayınlıyorum.

1. Sahip olduğumuz bir işlemci sürekli tek işlemcili mi olduğunu yada anlamam gereken başka birşey var mi ?

Bu doğru. Tek bir işlemci (birçok işlemciye sahip olan paralel bir işlemcinin aksine)

2. Kaydedicide hafıza durumu

st %r0 --> [r1]

Notlar bize "bir bellek konumu için bir işaretçi olarak r1 kaydedicisinin değerini işlediğini" göstrdi. Biz şimdiye kadar Pc hariç, fp ve sp nin yaptığının, "hafızada bir kayıt nereyi işaret ediyor" olduğunu biliyoruz.

Siz pc nin koda ve fp/sp nin yığında hafızaya iki noktada işaret ettiğini farzedebilirsiniz. Hatta bu varsayımlarla karmaşık şeyler yapıyorsanız bazı sistemlerde ihlaller olabilir (o kısma girmeteceğiz). Aksi halde, bir kaydın belli bir bellek kesimini işaret ettiğini kabul edemeyiz. r1 işaretçisi coda, globallere,alt yığın veya yığıta işaret edebilir.

3.Her işlem adres alanının 0x80000000 olduğunu kabul eder mi? Eğer öyle ise bu byte numarası bizim için çok büyük olur.Bizde tam 2048 Mb alan var oda hydras. Örneğin;RAM de 96Mb alana sahip onu 2048 yapıcaz,

8*(16^7)/(1024^2)

Yaptığım varsayım doğrumu? Bazıları doğru gelmiyor.

Evet, bellek 2Gb lık bir dizi olduğunu varsayar.Fakat her 2Gb tı kullanılmazÖzellikle,yığın ve yığın arası adresleri kullanılmayan ve adres alanların büyük kısmını oluştururlar.Bir işlemcei 2Gblık RAM dan az olsa bile sistem sanki her 2Gblık işleme bakmak için kuruldu.Bu "sanal bellek" diye adlandırılır ve CS560 hakkında bazı şeyler öğreneceğiz.Birkaç hafta içinde bellek arayüzünün nasıl sınırlı olduğunu göreceğiz. Özellikle de genel olarak kod ve globals bölümün megabayt alanı küçüktür.Makinemizdeişletim sistemi yığının 8M den daha buyuk olmasına izin vermez(tipi "limit" ve "yığıta"bak) ve yığın 96Mb tan daha müyümeye izin vermez.Eğer oyle olduğuna inanmıyorsanız bunu deneyebilirziniz;

Yığın Testi;

```
main(int argc, char **argv)
{
  char *s;
  s = (char *) malloc(atoi(argv[1]));

  if (s == NULL) { perror("malloc"); exit(1); }
  printf("malloc %d worked\n", atoi(argv[1]));
}

(This is on hydra1a)

UNIX> test1 90000000
malloc 90000000 worked
UNIX> test1 100000000
```

```
malloc 100000000 worked
UNIX> test1 110000000
malloc: Not enough memory
UNIX>
```

This (test2.c) tests the stack:

```
#include
main()
{
  char s[9000000];
  printf("s = 0x%x\n", s);
  printf("%d\n", *s);
}
```

(again on hydra1a)

```
UNIX> limit stacksize
stacksize       8192 kbytes
UNIX> test2
Segmentation fault (core dumped)
UNIX>
(If I change the 9000000 to 8000000, I get):
UNIX> test2
s = 0xef85da08
0
UNIX>
```