6502 Makine Dili Kursu

Bilgem ÇAKIR

6502 Makine Dili Kursu

Bilgem ÇAKIR

Yayımlanma 2005

İçindekiler

Giriş

Başlamadan Önce

Selamlar. Pekçok yerde benzer kaynaklar olmasına rağmen bir makine dili yazı dizisi de ben yazayım dedim. Amacım 6510 makine dili kullanarak nasıl program yazılabileceğini anlatmak olacak. Burada bence önemli olan komutlarla ilgili her detayı bilmenizden ziyade, bir problemi nasıl küçük parçalara bölüp assembler komutlarının yapabileceği basitlikte birimler halinde ifade edebilmeyi öğrenmenizdir.

Bu yazıyı takip edebilmeniz için bilmeniz gereken bazı konular var. Genel olarak CPU, register, program ve bellek kavramlarını tanıyor olmanız lazım. Bunun için Bilgisayar Mimarisine Giriş başlıklı yazıma bakabilir ya da pekçok başka kaynaktan yararlanabilirsiniz. Bu konuda hali hazırda bir yazım olduğu için burada tekrar anlatmayacağım.

İkinci bilmeniz gereken konu ise ikilik ve onaltılık sayı sistemleridir. Bunu da pekçok ilköğretim matematik kitabından araştırabilirsiniz. Bu sayı sistemlerini zaten bildiğinizi varsayarak bu yazıyı yazacağım.

Üçüncü bileceğinizi varsaydığım konu da mantık AND(ve), OR(veya) ve EOR(türkçesini bilmiyorum) operasyonlarıdır. Bunları da Lise 1 Matematik kitabından öğrenebilirsiniz.

Makine Dili nedir:

Bilgisayar Mimarileri belleğe yerleştirilmiş art arda komutları işletip bu komutların istekleri doğrultusunda giriş ve çıkış cihazlarını yöneterek bir takım işleri becerirler. Bilgisayar mimarisine giriş yazısında bahsettiğim gibi işte CPUnun bellekten bayt bayt alıp okuduğu ve değerlendirdiği komutların tümü o CPUnun makine dilini oluşturur. Dolayısıyla genelde her farklı CPU mimarisinin farklı makine dili vardır. Bu yazıda ben size 6510 makine dilini anlatacağım.

Zaten her işlemcide değişen diller yüzünden bir CPU için makine dilinde yazılan program başka CPUda çalışmaz. Bu yüzden de benzer programları değişik CPUlarda tekrar tekrar yazmamak için daha yüksek seviyeli programlama dilleri (mesela Pascal, C, BASIC vs) geliştirilmiştir. Bu dillerde yazılan programlar ya önce bir araç tarafından hedef CPUnun makine diline çevrilir (bu araca compiler denir) ya da o dildeki her komut bir araç tarafından birer birer okunup çalıştırma sırasında o komutun işini yapan küçük makine dili rutinleri çalıştırılır (bu araca da interpreter denir).

Genel olarak Makine dili komutları, son derece basit işler yapabilen ve bellekte de çok az yer tutan komutlardır. Çoğunlukla her komut şu işlemlerden birini yapar.

- bir registera bellekteki bir adresten değer yükleme (LDA, LDX, LDY...)
- bir registerdaki değeri bellekteki bir adrese saklama (STA, STX, STY...)
- Koşullu veya koşulsuz olarak bellekte bir yere atlama (BNE, BEQ, JMP...)
- Bir registerdaki değer ile bellekteki bir değeri kullanarak aritmetik veya mantıksal bir işlem yapma (ADC, SBC, AND, ORA, CMP...)

İşte bu basit komutları kullanarak dünyadaki en karmaşık programlar bile yazılabilir. Bunu mümkün kılan çok önemli bir kavram vardır.

Modüler, Yapısal, Fonksiyonel Programlama

Bu kavramı tek bir kelimede anlatmanın yolunu bulamadım. Hangi programlama dili kullanılırsa kullanılsın, büyük sistemleri kurabilmeyi sağlayan temel felsefe şudur. Problem daha küçük 4-5 alt parçaya bölünür ve her bir alt parça problem iyice tanımlanır. Şimdi elimizde her bir alt problemi çözebilen birer alt sistem olsaydı, bu alt sistemleri nasıl birbiriyle etkileştirmeliyiz ki ana problemi çözen bir sistem elde edelim. Yani alt sistemleri hangi sırayla çalıştırmalıyız, her bir alt sisteme ne girdiler vermeliyiz ve hangi alt sistemin çıktısını hangi alt sisteme vermeliyiz gibi sorular sorarak bir tasarım adımı atmış oluruz.

Böylece problemin bir kısmını çözmüş oluruz. Bu sefer elimizde her biri ilk problemimizden daha basit olan birkaç tane alt problemimiz olur. Her bir alt problemi kendi içinde daha da küçük birkaç parçaya bölüp onları çözen sistemler olduğunu varsayarak onların etkileşimini planlarız. Böylece her adımda problemleri çözülmesi daha kolay olan ve sayıca artan küçük problemlere dönüştürerek devam ederiz. Ta ki elimizdeki problemler o kadar küçülür ki artık bu problemleri çözen alt sistemlere bire bir karşılık gelen programlama dili komutları bulabiliriz. İşte bu noktada programı yazmaya başlarız.

İşte herhangi bir teknik dalda bir problemi çözebilmenin yaklaşımı budur. Büyük sistemleri iç içe alt sistemler halinde düşünüp alt sistemlerin birbiriyle etkileşimini tasarlayabildiğiniz zaman büyük sistemler inşa edilebilir.

Makine Dilinin Avantaj ve Dezavantajları

Makine dilinin çok önemli bazı avantajları vardır.

- Donanımın üzerinde ne olduğunu en ince ayrıntısına kadar bilebilme imkanı sağlar. Bu sayede yazdığınız hangi alt sistem bellekte nereye yerleşti. tam olarak ne hızda çalışacak vs gibi sorulara kesin cevaplar verebilirsiniz.
- *HIZ*. Yüksek seviyeli bir dil kullanırken kullandığınız makine diline çevirici aracın ne kadar performanslı olduğunu bilemezseniz. Ama şundan emin olabilirsiniz ki hiç bir otomatik çevirici makine dilinde tasarlanmış bir program kadar hızlı

çalışan bir çevrim elde edemez. Özellikle C64 üzerinde demo veya oyun programlarken makine dili en mantıklı seçim olacaktır.

- *BELLEK*. Makine dili programları bayt bayt belleğe dizilir. Bu yüzden mesela BASIC programlarına göre çok az yer kaplarlar.

Makine Dilinin en büyük dezavantajı komutların basit işler yapabilmesidir. Bu yüzden makine dilinde tasarım yapan bir kişi bir problemi çok daha küçük parçalara bölüp hatasız olarak inşa edebilirse ancak başarılı olur. Oysa daha yüksek seviyeli dilleri kullanan birinin aynı problemi çözmek için yazacağı program daha kolay ve kısa olabilir.

6510 Programlama Ortamı

6510 çipinin mimarisi elbette bu çipin makine diline yön verir. Bu mimarinin en temel bileşenlerinden bahsedeceğim. Bu esnada onların kullanıldıkları yerlere değinirken tanımadığınız veya anlamadığınız terimler geçebilir. şimdilik buna takılmayın. kulak aşinalığı olsun.

6510un en önemli alt birimleri şunlardır:

- Akümülatör register: Bu register aritmetik ve mantıksal işlemlere sokulacak değerleri saklamak ve işlem sonuçlarını elde tutmak için kullanılır.
- X register: Bu register da geçici olarak bilgi saklayabilir. tuttuğu değeri bir artırıp azaltabilir. En önemli görevi daha sonra bahsedeceğimiz indeksli adreslemede aldığı roldür.
- Y register: X registerine çok benzeyen bir yapısı vardır. hemen hemen aynı amaçlerla kullanılır. En önemli rolü ise yine sonra bahsedeceğimiz dolaylı indexli adreslemede öne çıkar.
- Program Counter: Kısaca PC diye bilinen bu register CPUnun işlediği komutları gösterir. Yani 16 bitlik bu register içinde işlenen komutun adresini tutar. Program içinde başka yerlere atlayan komutlar aslında PC registerinin değeri ile oynayarak bu işi becerirler.
- Status Register: Bu register içindeki bitlere çeşitli özel anlamlar yüklenmiştir. CPU da yapılan çeşitli işlemlerin sonucunda bu bitler sonuçla ilgili bazı bilgileri taşıyacak şekilde değerler alır. Örneğin yapılan işlemin sonucu sıfır çıkarsa Zero Bit denilen bit 1 olur. Eğer sonuç sıfır değilse bu bit 0 olur. Dikkat ederseniz bu bit bir hakem gibi sonucun 0'a eşit olması veya olmaması durumunu bayrak kaldırarak bize bildirir. Bu anlayıştan ötürü bu bitlere flag adı verilir. Status registerdaki flagler sayesinde Koşullu dallanma komutları iki farklı davranış gösterir. Örneğin beq komutu zero flag 1 iken verilen adrese dallanırken 0 iken hiç bir iş yapmadan kendisinden sonraki komutla devam eder.

İlk Komutlar: LDA, LDX, LDY, STA, STX, STY

Öğreneceğiniz ilk komutlar bunlar üç genel amaçlı registerimize bellekten değer yüklemeye ve bu registerlardaki değerleri bellekte bir yere yazmaya yarayan komutlardır. İşte birkaç örnek:

```
lda #10   ; akümülatöre 10 yükle


sta $0400 ; akumulatordeki degeri bellekte
          ; $0400 adresine gönder


ldx #$40  ; x registerine $40 yükle


stx $c000 ; x registerindeki de#eri bellekte
          ; $c000 adresine gönder


ldy #25 ; y registerine 25 yükle


sty $1000 ; y registerindeki de#eri
          ; bellekte $1000 adresine gönder
```

Önemli bir komut: JMP

jmp komutu koşulsuz olarak program akışını bir yerden başka yere atlatır. Bu komutla sonsuz döngüler kurulabilir.Örnegin kendi bulunduğu adrese atlayan bir jmp komutu bilgisayarı sonsuz bir döngüye sokar.

İlk assembler programı:

Artık ilk assembler programınızı yazmaya hazırsınız. Bunun için ya C64ünüzde bir kartuş kullanarak monitöre girmeniz gerekiyor, ya PC'de kullandığınız emülatörün monitörünü kullanacaksınız ya da PC'de herhangi bir text dosyasına program yazıp ACME gibi bir cross-assembler kullanacaksınız. Ben üçüncü yolu kullandığınızı varsayacağım. Bunun için ACME aracını indirip kurun. Widows için hazır executable mevcut. Linux'ta ise çok kısa bir compile işlemi gerekiyor. Bilgisayarınızda executable PATH'de bir yerlere kopyalayın ve adını acme64 olarak değiştirin (bazı linux sistemlerde acme adlı başkabir tool var onunla karışmaması için)

Şimdi yeni bir dosya yaratıp adına prog1.a64 diyelim. dosya eklentisinin ne olduğu aslında önemli değil ama ben kendime genelde böyle bir yol ciziyorum ki zaman içinde dosyalar biriktikçe aralarından c64 programlarını bulmak kolay olsun. size de tavsiye ederim.

Dosyayı text editörü ile açın (notepad, emacs vs.) ve şu satırları yazın:

```
;------------------------------
;; c64 binary dosyay# belirtelim
!to "out.prg"
CERCEVE_RENGI=$d020

*=$c000

start:
    lda# 0
    sta CERCEVE_RENGI
    lda# 1
    sta CERCEVE_RENGI
    jmp start

;------------------------------
```

Dikkat etmeniz gereken nokta şu: Bir satırda noktalı virgülden sonraki bölümler o satırın sonuna kadar ACME'de yorum satırı olarak algılanır. Bu yüzden o bölümlere istediğinizi yazabilirsiniz. Genelde programınızın kritik noktalarına yorum yazmak iyi bir alışkanlıktır.

Bunu kaydettikten sonra kaydettiğiniz klasörde bir konsol (dos) penceresi açın ve şu komutu girin.

```
acme64 prog1.a64
```

Eğer bir hata yapmadıysanız ACME herhangi bir hata vermeden çalışıp çıkar. Artık çalıştığınız klasörde ACMEnin yeni yarattığı out.prg isimli dosya bulunmalı. Bu dosya c64'e yuklendiği zaman çalışacak olan 6510 makine dili programını içeriyor. Şimdi konsolda şunu yazın:

```
x64 out.prg
```

Bu komut sonucu VICE emülatörü çalışıp out.prg'yi yükler ve ardından READY promptunu verir. Çalıştırmak için VICE penceresinde:

```
SYS 49152
```

komutunu vereceksiniz. Ve işte karşınızda ilk çalışan assembler programınız. İstediğiniz kadar izledikten sonra şimdi gelin arka planda olanları inceleyelim.

Neler Oluyor:

Öncelikle prog1.a64'ü inceleyelim. Yorum olmayan ilk satırda şu yazıyor:

```
!to "out.prg"
```

Evet tahmin ettiğiniz üzere bu komut çıkış dosyasını belirtiyor. Yani bu komut aslında bir makine dili komutu değil. Bu programcının ACME aracını kullanırken ACMEye bazı yönlendirmaler yapmak için kullandığı komutlardan biri. ACME komutları kodunuzun içinde çeşitli yerlerde kullanacağınız komutlar. Bunların biri hariç hepsi ! ile başlarlar. Bu komutların detaylı bilgilerini çapraz geliştirme yazı dizisinde bulacaksınız. Şimdilik bilmeniz gereken !to ve * komutları.

Hemen arkasından gelen CERCEVE_RENGI satırını birazdan açıklayacağız. Şimdilik bir sonraki satıra bakalım:

```
*=$c000
```

bu satırda da ACMEye programımızı C64ün belleğinde hangi adrese yerleştireceğini söylemiş oluyoruz. $c000 - $cfff adres bölgesi BASIC belleğinin dışında kalan bir yer olduğu için makine dilinde programlamaya yeni başlayanlar için en ideal çalışma alanıdır. o yüzden biz de programlarımızı o bölgeye yerleştireceğiz.

Sonraki satırda ise etiket kullanımını görüyoruz:

```
start:
```

Burada ACME bellekte makine dili kodunu oluştururken bu satıra geldiği esnada yerleştirme yapacağı adresin ne olduğuna bakar. Biz az önce * komutu ile adresi $c000 yapmıştık. ACME bu adrese etiketteki ismi verir. yani artık ?start? kelimesi $c000'a eşit olmuş olur. Programın diğer satırlarında start geçerse ACME otomatik olarak start yerine $c000 adresini koyacak.

Etiketler kullanarak programcılar bellek adreslerini bilmeye gerek olmadan ve bu esnada sık yapılan dikkatsizlik hatalarının çoğundan kurtularak program yazabilirler. Özellikle dallanma komutlarında adreslere etiket koymak hem programı daha anlaşılır kılar hem de pekçok olası hatayı önler.

Etiketler ayrıca her yerde her değere tanımlanabilir. Az önce atladığımız satıra şimdi bakabiliriz. Bu satırda böyle bir etiket atama işlemini görüyoruz:

```
CERCEVE_RENGI=$d020
```

start nasıl $c000'a eşitlendiyse CERCEVE_RENGI isimli etiket de $d020'ye eşlenmiş oldu. Bu teknik de özellikle bellekteki değişkenlerin veya register adreslerinin unutulmasına karşın kolay hatırlanır isimlerle erişilmesini sağlar.

Etiketlerle ilgili çok önemli bir avantaj da böyle yazılan programların bellekteki yerleşiminin istendiğinde programcı tarafından kolayca değiştirilebilmesidir (buna relokasyon denir). Etiketlerle ilgili daha detaylı bilgiyi bu kursun sonundaki ekler bölümünde bulabilirsiniz.

C64ün grafik işlerinden sorumlu çipi olan VIC-II bir grup registera sahiptir. Bu registerlara zaman içinde detaylı olarak değineceğiz. Şimdilik ilk kullanacağımız register $d020 adresinde bulunuyor. Bu register ekranın çerçeve rengini kontrol eder. C64ün desteklediği 16 renkten birinin numarasını bu registere yazdığınızda çerçeve o renkte çizilir. Renkkodlarına da daha sonra döneceğiz ama şu an için 0 nolu rengin siyah 1 nolu rengin de beyaz olduğunu bilin yeter.

Böylece sonraki satırlarda ne olduğunu artık tahmin edebiliyorsunuzdur:

```
lda# 0
      sta CERCEVE_RENGI
```

aslında $c000 adresine yerleşen ilk komutlar bunlar. Bu ko-

mutlar önce akümülatöre 0 değeri yükleyip ardından cerceve rengi registerine bu değeri yazıyor.

```
lda# 1
      sta CERCEVE_RENG#
```

hemen ardından aynı registere bu sefer de 1 degerini yazıyoruz.

```
jmp start
```

ve arkasından da başa dönüyoruz. Yani programımız hic durmadan çerçeve rengini siyah ile beyaz arasında sürekli değiştiriyor.

Böylece en temel makine dili bilgileri ile tanışmış oldunuz. İlk programınızı da yazdınız. Şimdi çerçeve rengiregisterine yüklenen değerleri 0 ve 1 den farklı rakamlarla deneyip oynayabilir ve renk kodlarını keşfedebilirsiniz. Yazının gelecek bölümünde yeni komutlar öğrenecek ve daha ilginç programlar yazabileceksiniz.

Sayma ve Karşılaştırma

Merhabalar. Geçen sayıda kaldığımız yerden devam edeceğiz. Bu yazıda değineceğimiz konular şunlar olacak:

- video matrix nedir?
- inx, iny, dex,dey komutları
- karşılaştırma komutları: cmp, cpx, cpy
- ilk kontrollü dallanma komutları: bne, beq
- yeni bir adresleme modu: indexli adresleme
- döngülerin tasarımı ve kullanılışları

Video Matrix nedir?

Aslında Video matrix 6510 assembly ile ilgili bir konu değil. Daha çok C-64'ün grafik çipi olan VIC ile ilgili bir konu. Fakat assembly üzerine çalışmalarınızı yaparken, yazdığınız programların yaptığı işlerin ekranda bi takım sonuçlarını görebilmeniz için, yeri geldikçe VIC çipine dair konulara da değiniyorum. Burada da VIC'le ilgili en önemli kavramlardan olan Video Matrixi tanıtacağım.

Video Matrix(kısaca VM) aslında bellekte biryerlerde duran 1000 baytlık bir bölgedir. VIC çipinin içindeki bir register kullanılarak bellekteki hangi 1000 baytlık bölgenin VM olarak kullanılacağı ayarlanabilir. Normal de VICin default olarak kullandığı VM adresi $0400 dan baslayan 1000 baytlık bölgedir.

VM içindeki baytların iki görevi vardır: 1- VIC text modundayken (normalde bilgisayar ilk açıldığında VIC text modundadır) ekranda hangi harfin nerede çıkacağı bilgisini taşır. 2- VIC bitmap grafik modundayken bazı pixellerin renk bilgisini taşır.

Şu an ikinci göreve değinmeyeceğim. Başta VM'i birinci rolde kullanacağız.

C64 ekranı 40 karakter genişliğinde 25 karakter yüksekliğindedir. İşte bu 40 x 25 = 1000 karakterin ne olacağı VM'de yazılıdır. Ekrandaki ilk satırdaki 40 karakter VM'in ilk 40 baytına karşılık gelir. İkinci satır VM'in 41'inciden 80. baytına kadar olan bölgedeki baytlarda saklanır. Böyle böyle satırlar arka arkaya VM içinde 40 bayt 40 bayt dizilir. Her satır kendi içinde soldan sağa sırasını takip eder.

Örnek: VM $0400 adresindeyken - ekranda sol üst karakterin adresi = $0400 - en üst satır ikinci karakterin adresi = $0401 - ikinci satır ilk karakter = $0400 + 40 = $0428

genelde bilgisayarda birşeyleri sayarken hep 0 dan başlamak işlemleri kolaylaştırır. Mesela c64 ekranında bir satırda 40 karakter var demiştik. Bunları numaralandırıken en soldakine 0 en sağdakine 39 demeye alışın. Aynı şekilde en üst satır 0 nolu satır, en alt satır da 24 nolu satır diye isimlendirilir. Böylece ekranda herhangi bir yerdeki karakterin adresini bulmak istersek şöyle formülize edebiliriz.

Ekran Adresi = VM başlangıcı + (satır * 40) + sütun,

Mesela: Satır=0, sütun = 2 ---------- > adres = $0400 + (0 * 40) + 2 = $0402 Satır=10, sütun = 12 ------ > adres = $0400 + (10*40) + 12 = $059c

Tabii bu adresler herşeyi açıklamıyor VM ile ilgili anlatılaması gereken bir şey daha var: Karakter kodları....

C64 ün ekranda gösterdiği her harf için bir kod numarası mevcuttur. Mesela A harfinin kod numarası 1 dir. D harfininki 4 tür. Bu kodları c64 kullanma klavuzunda veya Programcının El kitabında bulunabilir.

İşte VM içindeki baytların değerleri VIC tarafından karakter kodları olarak değerlendiril ve VIC ekrandaki görüntüyü oluştururken ekrandaki her adrese VM'deki karakter koduna karşılık gelen harfi çizer.

Mesela $0400 adresinden itibaren 5 adet adrese şöyle baytlar yerleştirsek

```
$0400 < --- 1
$0401 < --- 2
$0402 < --- 1
$0403 < --- 2
$0404 < --- 3
```

ekranda sol üst kosede ABABC yazdıgını goruruz. Bunun kodunu yazalım

```
!to "out.prg"
*=$c000

lda #1
sta $0400
lda #2
sta $0401
lda #1
sta $0402
lda #2
sta $0403
lda #3
sta $0404
end:
jmp end ; sonsuz döngü
```

bu programı acme ile assemble edip çıkan out.prg dosyasını da vice ile açarsanız sys49152 komutuyla programı çalıştırdığınızda ekranın sol üst köşesinde ABABC çıkacak. Alfabenin diğer harflerini ve diğer ekran konumlarını deneyebilirsiniz.

Egzersiz: Ekranın ortasına "MERHABA DUNYA" yazın.

Ekrana böyle tek tek karakterleri lda sta yaparak bişeyler yazdırmanın hayli yorucu olduğunu gördünüz. Bunu daha kolay hallet-menin yolları var. Bu yolları öğrenebilmek için yeni komutlar öğreneceğiz.

INX, INY, DEX, DEY

Bu komutlar X ve Y registerlerindeki değeri bir artırıp bir azaltmaya yararlar. Belki de en sık kullanılan komutlar arasındadırlar. Genelde ya bir döngünün kaç kere döneceğini saymak veya bellekte ardışık baytlara erişmek için kullanılırlar.

- INX X registerindeki değeri bir artırır. Eğer X'te o anda 255 değeri varsa INX'den sonra X'deki değer 0 olur. Bunun sonucu olarak Status Registerde Zero flag 1 olur.

- INY Y registerindeki değeri bir artırır. Eğer Y'de o anda 255 değeri varsa INY'den sonra Y'deki değer 0 olur. Bunun sonucu olarak Status Registerde Zero flag 1 olur.

- DEX X registerindeki değeri bir azaltır. Eğer X'de o anda 0 değeri varsa DEX'den sonra X'deki değer 255 olur.

- DEY Y registerindeki değeri bir azaltır. Eğer Y'de o anda 0 değeri varsa DEY'den sonra Y'deki değer 255 olur.

```
ldx #1
stx $0400
inx
stx $0401
inx
stx $0402
```

Bu kod en üst sol köşede ABC yazmasını saglar.

Karşılaştırma Komutları: CMP, CPX, CPY

Bu komutlar 6510daki üç ana registerdeki değerleri başka değerlerle karşılaştırmaya yarar. Biz öncelikle en basit hallerini göreceğiz

CMP, akümülatördeki değeri başka bi değerle kıyaslamaya yarar. Kıyaslamanın sonucuna göre Status registerdeki bazı bayraklar 1 veya 0 olur. Örneğin

cmp #10

bu komutu görünce CPU şu işlemleri yapar - eğer akümülatör verilen değere eşit ise (bu örnekte 10'a) Status registerde Zero flag 1, Carry flag 1 olur - eğer akümülatör verilen değerden küçükse Status registerde Zero flag 0, Carry flag 0 olur - eğer akumulator verılen değerden buyukse Status registerde Zero flag 0, Carry flag 1 olur.

Bu komutun hemen arkasından kullanılacak bir koşullu dallanma komutu ile (bne, beq, bcs, bcc, ...)bütün koşullarda farklı yerlere dallanılabilir.

Cpx ve cpy komutları aynı işlemleri X ve Y registerleri üzerinde yaparlar.

İlk Koşullu Dallanma Komutları: BEQ VE BNE

BEQ komutu aslında çok basit bir iş yapar. Status registerindeki Zero flag'a bakar. Zero flag 1 ise BEQ komutuyla verilen adrese atlar. Değilse hiçbisey yapmadan program bi sonraki komuttan devam eder. Zero flag hatırlarsanız cmp,cpz,cpy komutlarından sonra eşiylik halinde 1 eşit olmama halinde ise sıfır oluyordu.

BNE ise BEQ'nun tam tersini yapar. Yani zero flag 0 ise verilen adrese atlanmasına sebep olur. 1 ise devam eder.

```
start: ldx #0
loop: stx $d020
      inx
      cpx #4
      bne loop
      jmp start
```

Bu örneği anlamanız çok önemli. Bu programın sonucu olarak yine ilk programımızdaki gibi ekran çerçevesinde renkler goreceksiniz. Fakat bu sefer yalnızca siyah ve beyaz degil kırmızı ve türkuaz da olacak.

Programımız x registerine 0 yazarak başlıyor. Ardından bu değeri VICin çerçeve rengi registerine yazıyoruz.. Sonra X deki değeri bir artırıyoruz. 4 olmuş mu diye kontrol edyoruz. Eğer dört olmamışsa loop adresine atlayıp X'deki yeni değeri çerçeve rengine yazıyoruz. X'i tekrar artırıyoruz. Tekrar kontrol ediyoruz. X registerindeki değer her seferinde bir artırılıp çerçeve rengine yazılmış oluyor. Ta ki X deki değer 4 olana kadar. Bu noktada bne satırı loop adresıne dallanma yapmıyor. Programımız jmp satırına geliyor. Buradan X'in 0 la yüklendiği ilk başa atlayarak herşey başa dönüyor. Program sonsuza kadar X registerinde 0,1,2,3,0,1,2,3,0,1,2,3... değerlerini dödürüp bunları çerçeve rengine yazarak devam ediyor.

Aynı programı bne değilde beq ile de yazabilirdik:

```
start: ldx #0
loop:  stx $d020
       inx
       cpx #4
       beq start
       jmp loop
```

bu sefer X'te 0,1,2,3 değerleri oldukça beq dallanma yapmayacak devam edecekti. Loopa dallanan yer jmp satırı olacaktı.

Yeni Bir Adresleme Modu: İndeksli Adresleme

Aslında adresleme modlarına daha once değınmedik. Farkında olmadan şimdiye kadar iki adresleme modunu kullanageldiniz.

Adresleme modları kod yazarken yazılan komutun aldığı argümanı bellekten nereden alacağını belirler. Bunu biraz daha açalım. Mesela lda komutunu düşünün. Bildiğiniz gibi lda akümülatöre bir değer yüklemeye yarıyordu. İşte lda komutunu kullanacağımız adresleme modu akümülatöre yüklenecek değerin nereden okunacağını belirler.

```
lda #0
```

Yazdığımızda ilk adresleme modu olan "immediate" adreslemeyi kullanıyoruz. Bu modda komut hemen yanına yazılan değeri yüklüyor. Bu modu tanımlayan şey lda'dan sonra gelen # işaretidir.

```
lda $0400
```

ise akümülatöre $0400'daki değeri okur. Bu adresleme moduna da "mutlak" adresleme denir. Bu sefer komuttan hemen sonra gelen baytlar akumulatore dogrudan yüklenmez. Yüklenecek bilginin adresi olarak değerlendirilirler. Bu adresleme modunu kullanmak için # işaretini kullanmaksızın doğrudan adres yazılır.

```
lda 0
```

Bu komut ise yeni başlayanların çok sık başını ağrıtan bi hataya sebep olabilir. Burada "immedıate" adresleme kullandığını düşünen birisi, bu komuttan sonra akümülatöre 0 yüklenmediğini farkettiğinde şaşıracaktır. Halbuki burada # işareti olmadığı için CPU 0 değerini adres olarak değerlendirir ve 0. adresteki baytı okur akümülatöre.

Gelelim üçüncü adresleme moduna. İndexli adresleme.

```
ldx #2
lda $0400,x
```

burada yeni bir adresleme görüyorsunuz. Bu moda x indexli adresleme denir. Xindexli adreslemede adres verilen mutlak adrese x registerindeki değerin eklenmesi ile bulunur. Yani yukarıdaki kod parçası $0402 adresindeki değeri akümülatöre yükler. Aynı şey y registeri ile de yapılabilir.

```
ldy #$80
lda $0400,y
```

akümülatöre $0480 adresindeki değeri yükler. İndexli adresleme pekçok kullanım alanına sahiptir. Ama öncelikle biz bu derste öğrendiklerimizle şöyle önemli birkaç program örneği görecegiz. Fakat bundan da önce son olarak *döngülerden* kısaca bahsedelim

Döngülerin Tasarımı ve Kullanımı

Genelde tekrar eden bitakım işler yaparken döngüler kullanırız. Kimi döngüler sonsuz iken kimileri de sınırlı sayıda çalışacak şekilde tasarlanır. Zaten ilk yazdığımız program da basit bir sonsuz döngüydü.

Sınırlı sayıda çalısacak olan döngülerde ise şu noktalar önemlidir.

- döngude en azından bir değişken (bellekte bir adres ya da bir register) değişiyor olmalı.
- bu değişkenin alabildiği değerlerden biri döngünün sonlanmasına sebep verip döngüden çıkılmasını sağlayabilmeli

en sık kullanılan kontrollü döngülerde, genelde X veya Y registeri 0 değerine getirilerek döngüye girilir. Döngüde yapılması istenen işlem yapıldıktan sonra registerin değeri bir artırılır. Arkasından registerin değeri döngünün dönmesini istediğimiz sayı ile karşılaştırılır. Henüz olmamışsa loop edilir

```
      ldx #0
loop: ...
      ...
      ...
      inx
      cpx #5
      bne loop
      ...
      ...
```

Örnek Programlar

Bu derste aslında bayağı çok şey öğrendik. Çok daha profesyönel görünüşlü programlar yapabiliriz artık.

Ekranı temizleme:

```
        * =$c000
start:  ldx #0
        lda #$20 ; bo#luk karakterinin karakter kodu
loop:   sta $0400,x
        sta $0500,x
        sta $0600,x
        sta $0700,x
        inx
        bne loop
```

Bu örnekte döngünün her dönüşünde VM içindeki 4 byte boşluk karakteri ile dolduruluyor. Son satır biraz faklı gelebilir size. "cpx satırı nerede" diyebilirsiniz. Dikkatli düşünürseniz, bne komutu kendisinden önce ne komut geldiğine bakmadan çalışıyor. Bne komutu zero flag'a bakıyor sadece. Inx komutunu acıklarken 255ten sonra sıfıra döndüğünü ve zero flag'i de bu esnada bir yaptığını söylemiştik. İşte bne komutu bu örnekte 256 kere dallanma yaptıktan sonra 257.seferde x registerindeki değer 0 olduğu anda loopdan çıkılmasını sağlayacak.

Diğer bir nokta x registeri bu 256 gecişte birer arttıgı için sta komutları her geçişte yeni dort hedef adrese erişecek. Yani döngünün ilk geçişinde x 0 iken sta komutu ile $0400, $0500, $0600 ve $0700 adreslerine erişecek. İkinci geçişte x 1 olacağı için $0401, $0501, $0601 ve $0701 adresleri temizlenecek. Son olarak x $ff iken $04ff, $05ff, $06ff ve $07ff adreslerıne $20 yazıldıktan sonra ınx komutu yüzünden x 0 olacak. Zero flag 1 olacak ve bne dallanmayacak.

Text kopyalama:

```
        *=$c000
Start:  ldx #0
loop:   lda text_source,x
        sta $0400,x
        inx
        cpx #40
        bne loop
end:    jmp end

text_source: !scr "merhaba dunya. Ben asm ogreniyorum......"
```

bu program en ust satıra text source daki yazının 40 harfini kopyalayacak. Eger 40 harften az yazarsanız o zaman cpx satırını değiştirebilirsiniz.

Bu programda bir acme komutu daha öğreniyorsunuz. !scr komutu belleğe karakter kodlarıyla yazı yerleştirmeye yarar.

Zaman Geçirme (delay)

```
        ldx #0
delay:  inx
        bne delay
```

Bu kod parçası hiçbirşey yapmadan bi süre bekleme sağlar. Böyle beklemeler demo ve oyunlarda bazen çok hızlı olan şeylerin daha güzel görünüp izlenmesi için kullanılabilir. Yukarıdaki kod yaklaşik 1 mili saniyelik bir beklemeye sebep olur.

Daha çok delay:

```
        ldy #$30
        ldx #0
delay:  dex
        bne delay
        dey
        bne delay
```

Bu kod parçası ise yaklaşık yarım saniyelik bir beklemeye sebep olur. Döngü toplam 256 * 48 kere döner.

Ödev: Yazacagınız program ekranı bosaltsın. Ardından ekranın ortasında 12. satıra "merhaba dünya" yazsın. Bir süre bekledikten sonra merhaba dunyayı silip "iste gelıyorum" yazsın.

Bıraz daha bekleyıp "bangır bangır " yazsın ve oylece kalsın.

Bu odevi yaptıgınız zaman ilk ciddi programınız hazır olacak. Gelecek bölümde aritmetik ve mantık işlemleri ile altprogramlara (subroutine) değineceğim.

Her zaman sorularınızı,yorumlarınızı ve düzeltmelerinizi night-lord at nightlord nokta dr2 nokta net adresine mail atabilirsiniz. Forumlarda da beni bulma şansınız çok yüksek.

Registerlar Arası Transfer

Daha ileri konulara geçmeden önce basit bir konuyu daha aradan çıkaralım. 6510 içinde sürekli kullandığımız sevgili 3 registerimiz olan A, X ve Y arasında dataları transfer etmek mümkün. Bunun için TXA, TYA, TAX ve TAY komutlarını kullanıyoruz. TXA X'deki değeri A ya yüklüyor. TAX tam tersine A'daki değeri X'e yüklüyor. TYA ve TAY'nin ne yaptığını da tahmin ettiğinizi sanıyorum.

Alt Programlara Atlayıp Geri Dönmek

Bu konu bütün programlama dillerinde onyıllar boyunca en çok konuşulan, uygulanan ve işe yarayan kavramlardan biri olmuştur. Buradaki mantığı iyi anlamanız gerekiyor.

Alt programlar kavramına ilk olarak programcıların bazı kodları tekrar tekrar yazmak istememesi ile doğuyor. Örneğin bi önceki bölümün sonunda verilen ödev programı yaptıysanız, yazıların ekrana çıkmaları arasında iki yerde aynı delay kodunu yazmanız gerekli idi. İşte programcılar bu gibi durumlarda programın çeşitli yerlerinde aynı işi yapan farklı bölümler yazmaktan kacınırlar. Buna yazılım mühendisliğinde "redundancy" (boş yere tekrar) denir. Sistem güvenliği için aynı şeyden bilerek birden fazla kopya bulunduralan durumlar haricinde Redundancy yazılımda istenmeyen birşeydir.

Bu yüzden programlama dillerine alt bir program parçasına atlayıp atlanılan yere geri dönebilme kavramı ekleniyor. Dikkat ederseniz bu işlem sıradan bir JMP işleminden farklı. Burada programınızın herhangi bir yerindeyken alt programa dallanıyorsunuz. Fakat bilgisayar dallanmadan önce son kaldığınız yeri aklında tutuyor. Alt program çalışmasını bitirdikten sonra programınız son kaldığı yerden devam ediyor. JMP komutunda ise bilgisayar nereden atlandığını hatırlamıyor.

Bu olay 6510 Assembly'de iki komut ile uygulanabiliyor. JSR ve RTS komutları. JSR komutu verilen adresteki alt programa atlıyor. Bunu yaparken CPU JSR komutundan bir sonraki komutun adresini kaydediyor. Alt program yapacağı işleri yaptıktan sonra RTS komutu ile bitiyor. CPU rts komutunu gördüğü anda kaydettiği adresi tekrar Program Counter registerine yüklüyor. Böylece programınız JSR komutundan hemen sonraki komutla çalışmasına devam ediyor.

Alt programlar sayesinde programınızı daha küçük parçalara organize bir şekilde bölerek, daha verimli programlar yazmanız mümkündür. Programınızda redundancy olmamış olur. Fakat böyle alt programlarla tasarım yapmanın bazı başka çok önemli faydaları da vardır. Modülerlik gibi.

Kursun ilk bölümünde modüler tasarımın ne olduğundan ve problem çözmeyi sistematik olarak nasıl kolaylaştırdığından bahsetmiştim. Bır diğer avantajı da yazdığınız bazı kodları tekrar tekrar kullanabilmenizi sağlamasıdır. Buna da yazılım mühendisliğinde "code reuse" denir. Yapacagınız programlarda hep tekrar eden problemler olacak. Bellekte bir bölgeyi bir değerle doldurma, bir bölgeyi başka bir bölgeye kopyalama, hatta ekrana pixel basma, ekrana poligon çizme vs... Bunlar için alt programlar yazıp biriktirdikçe kendinize bir takım kod kütüphaneleri oluşturabilirsiniz. Bu sayede yeni programlar yazarken daha önceden çözdüğünüz problemlerle vakit kaybetmeyip sadece yeni problemlere odaklanabilirsiniz.

Fakat şunu da belirtmekte fayda var. Demoscene programcılığı yer yer yazılım mühendisliğinden farklılık gösterir. Demolarda aynı efektlerin tekrar tekrar kullanılması çok hoş karşılanan bir durum değildir. Buna "recycling effects" denir ve bunu yapan coderlara çok iyi gözle bakılmaz. Bu yüzden kütüphanelerinizi oluştururken çok spesifik bazı efektleri kütüphane haline getirmenin faydasını göremezsiniz. Örneğin "bir plasma alt programı yazayım... demolarda tekrar tekrar jsr yapar kullanırım" diye düşünmeyin. Ama efektten bağımsız olarak çok kullanılan pixel basma, poligon çizme, ekran temizletme, delay, bellek kopyalama gibi işlemleri yapan alt programları bir kere yazıp tekrar tekrar kullanmanızda hiçbir sakınca yoktur. Çok da fayda vardır.

Son olarak hakında çok fazla olumlu şey söylediğmiz alt programların bize bir belei olduğunu da söylemeliyiz. Bu bedel sizi daha bayağı ilerleyene kadar endişelendirmesin. Daha çok, ileri seviye demo efektleri kodalarken karşınıza çıkabilecek bir problem bu. Hız... Jsr ve rts komutları beraber 12 cycle zaman tüketirler. Henüz cycle kavramından bahsetmedik ama kısaca bu iki komutun genelde 3 - 4 assembly komutu kadar zaman harcadığını söyleyebiliriz. Bunun ne önemi var?

Diyelimki yazdığınız alt program çok çok basit ve küçük üstelik de programın geri kalanında çok az yerden çağırılıyor. Bu durumda programınızda alt programolarak yapılan bölümdeki komutların aldığı süre ile jsr ve rts komutlarının harcadığı süre bazen alt program komutlarının tek başına harcadığı süreyi iki katına çıkarabilir. Bu çoğu zaman önemli değildir. Fakat eğer jsr komutu çok tekrar eden bir döngünün içindeyse (mesela 10000 kere donen bir döngüde) o zaman altprogramdakı komutları aynen dongunun içine koyup jsr rts olayından kutularak programı iki kat hızlandırabilirsiniz. Fakat dediğim gibi böyle bir durumla karşılaşmak için biraz erken.

Şimdi birkaç örneğe bakalım. İlk olarak bir önceki bölümdeki ödevi yeni öğrendiğimiz komutlarla çözelim:

```
* = $c000

jsr clear_screen
jsr delay
jsr copy_text1
jsr delay
jsr copy_text2
jsr delay
jsr copy_text3
end:
jmp end


clear_screen:
ldx #0
lda #20
cs_loop:
sta $0400,x
sta $0500,x
sta $0600,x
sta $0700,x
inx
bne cs_loop
rts

screen = $0400 + (12*40)

copy_text1:
ldx #0
ct_loop1:
lda text1,x
```

```
sta screen,x
inx
cpx #40
bne ct_loop1
rts

copy_text2:
ldx #0
ct_loop2:
lda text2,x
sta screen,x
inx
cpx #40
bne ct_loop2
rts

copy_text3:
ldx #0
ct_loop3:
lda text3,x
sta screen,x
inx
cpx #40
bne ct_loop3
rts

delay:
ldy #0
ldx #0
d_loop:
dex
bne d_loop
dey
bne d_loop
rts
text1:
!scr "merhaba dunya                              "
text2:
!scr "iste geliyorum                                "
text3:
!scr "bangir bangir                                 "
```

Bu kod bir onceki ödevde verilen problemi çözüyor. Fakat hala bu koda bakınca sizi rahatsız eden birşeyler olmalı. Copy_text1, copy_text2 ve copy_text alt programları sanki çok fazla birbirine benziyor.

Zaten bir programı yazarken tecrübe kazandıkça böyle bi takım kod parçalarında problemin kokusunu almaya başlayacaksınız. Bu tam olarak öğretilebilen birşey değil. Biraz tasarımcı içgüdüsü biraz da tecrübe ile alakalı. Fakat buradaki problemi giderip programı daha da basitleştirebilmek için birşeyden daha bahsetmemiz gerekiyor. Sonraki bölüme geçip alt programları esnekleştiren argümanları öğrenebilirsiniz.

Alt Programlara Argüman Geçirme

Argüman kelimesini matematik derslerinden tanıyor olduğunuzu varsayıyorum. Burada bir altprogramı biraz daha esnek hale nasıl getirebileceğimizi tartışacağız. Şimdiye kadar alt programları sadece programın içinde birbirinin tıpatıp aynısı olan kod parçalarını tek bir alt programa indirgerken kullandık.

Delay alt programını ele alalım.

```
delay:
ldy #0
ldx #0
d_loop:
dex
bne d_loop
dey
bne d_loop
rts
```

Bu alt program sabit bir süre bekleme yapıyor. Peki ya programımızda değişik yerlerde değişik uzunluklarda bekleme yapmak isteseydik. Bu alt program işimizi görmeyecekti peki programı şöyle değiştirsek:

```
delay:
ldx #0
d_loop:
dex
bne d_loop
dey
bne d_loop
rts
```

şu anda sadece Y'yı sıfırlayan satırı attık. Buna ilave olarak programın değişik yerlerinden bu alt programı çağırıken jsr komutundan hemen önce Y'ye istediğimiz değeri yüklesek. Mesela

```
ldy#$30
    jsr delay
```

bu çağırışta yaklaşık yarım saniyelik bir delay olacak. Daha uzun bekleme gereken başka bir yerde

```
ldy#$c0
    jsr delay
```

olarak çağırırsak yaklaşık iki saniyelik bi bekleme olacak. Dikkat ederseniz çağırdığımız yerde delayin uzunluğunu ayarlayabiliyoruz. Böylece çok daha esnek bir alt program elde etmiş olduk. Bunu da alt programa delay süresini değiştiren bir datayı register üzerinde geçirerek sağladık. Y registeri programın geri kalanı ile alt program arasında argüman taşıyan bir köprü görevi gördü.

İşte böyle bir alt programa argüman geçirdiğiniz zaman bu alt program kütüphanenizde daha çok reuse edilebilen daha esnek bir alt program olmuş olur. Buradaki mantığı iyi kavramanız önemli. Bir alt programı çağıran diğer kod parçalarına çoğu zaman "client" (müşteri) denir. Müşteri tarahı sizin alt programınızdan alacağı hizmeti kendi ihtiyacına göre değiştirebilmiş olur.

Bu argüman geçirme konusunu kullanarak alt programlarınızı tasarlarken ne secenekleriniz olduğunu inceleyelim.

- delay örneğinde olduğu gibi bir registeri kullanarak argüman geçirebilirsiniz.
- Bellekteki herhangi bir adresi kullanabilirsiniz. Müşteri o adrese argümanı yükler. Alt program o adresten argümanı okuyup işlemini yapar.

Bu seceneklerden hangisini kullanacağınız birkaç faktöre bağlıdır.

- alt programa kaç tane argüman geçiriliyor. Örnegin delay'de bir argüman geçti. Mesela ekrana pixel koyan bir alt programa ise çoğu zaman üç argüman geçirilir (x,y, renk)ç Eğer üçten az argüman geçirecekseniz direk olarak 3 registerden istediklerinizi kullanabilirsiniz. Eğer üçten fazla argüman geçecekse (örneğin bir çizgi rutinine en az 4 değer geçer: x1,y1,x2,y2) bellekteki adresleri kullanmak gerekecektir.
- Bazen üçten az argüman geçirseniz bile bellek kullanmak isteyebilirsiniz. Ama burada ne demek istediğimi şimdi açıklamayacağım. İleride değineceğim.

Bu esneklikten faydalanarak deminki ödev programında copy_text alt programlarını teke indirgeyeceğiz fakat bunu yaparken kullanacağımız bir trik daha var.

Kendi Kendini Değiştiren Programlar

Bir programın kendisini değiştirmei ne demek diye şaşırabilirsiniz. Dikkat ederseniz programlar bellekteki ard arda dizili baytlardan ibaret CPU içindeki Program Counter bu baytların üzerinde dolaşırken CPU, okuduğu değerlerin bazılarını komut bazılarını da komutlara verilen argümanlar olarak okuyup programı çalıştırıyor. Komutlar CPUnun bellekte her yere okuyup yazmasını sağlayabiliyor. Dolayısıyla progrmımızdaki konutlar programın başka bir bölgeindeki baytlara yazım yapılmasını ve oralardaki "programın" değişmesini sağlayabilir.

Bir programın böyle kendi kendini değiştirmesi aslında bazı tehlikeleri de barındırır. Kendini değiştiren programlar yazan birisi bazı hatalar yaptığı zaman bulup düzeltmesi daha zor olabilir. Hatta bu yüzden modern sistemlerde tercih edilmez ve hatta yasaklanır. Örenğin PC'lerdeki işletim sistemleri bir programı çalıştırırken kod bölgesini bellekte read-only olarak bulundururç birileri o bölgeye yazmaya kalkarsa izin vermez.

Ama C64 dünyasında bazı zamanlarda kendini değiştiren kod yazmak en etkin ve kolay çözümdür. Şimdi bu tekniği copy_text rutinimize uygulayacağız. Faka t birkaç şeyi birden öğreneceksiniz. O yüzden şimdi dikkatinizi çok iyi toplayın ve bu bölümü sonuna kadar sabırla okuyun.

```
copy_text:
stx ct_loop + 1
sty ct_loop + 2
ldx #0
ct_loop:
lda $0000,x
sta screen,x
inx
cpx #40
bne ct_loop
rts
```

Şu anda copy_text alt programının tasarımını deiştirdik artık kopyalanmasını istediğimiz texti müşteri tarafında belirtmemiz ve alt rutine geçirmemiz gerekiyor yani müşteri alt programı şöyle çağıracak

```
ldx #<text1
ldy #>text1
jsr copy_text
```

Şimdi yine ilk defa gördüğünüz işaretler var. Bu işaretlerin ne olduğunu anlatabilmek için "endianness" adı verilen kavramdan kısaca bahsetmem lazım.

Bilgisayarda 0 - 255 arası sayıların bir bayta sığdığını biliyorsunuz. Peki ya bundan büyük sayılar. İşte 256 ile 65531 arasındaki sayılar iki bayta sığarlar yani hex sayı sisteminde yazarsak, $0100 ile $ffff arasındaki sayılardan sözediyoruz.

Bu iki baytlık (veya 16 bitlik) sayılarla uğraşırken bunların bellekte nasıl tutulabıleceğini düşünürseniz, iki secenek gelir aklınıza. Örneğin $c000 sayısı ardışık iki bayta $c0 $00 olarak kaydedilebilir (büyük bayt önce) ya da $00 $c0 diye kaydedilebilir (küçük bayt önce). Her işlemci mimarisi bu sıralama yollarından birini seçmiştir. 65xx serisi işlemcilerde küçük bayt önce saklanır. Bu seçime o çipin "endianness" ı denir. 65xx çipleri "little endian" işlemcilerdir.

Mesela lda $c000 komutu bellekte $ad $00 $c0 diye üç bayt kaplar. İlk bayt lda'nın mutlak adresleme modundaki halinin kodudur. Ardından gelen iki argüman baytı önce küçük bayt ardından büyük bayt olarak okunur.

İşte ACME cross-assemblerinde 16 bitlik bir değerin küçük baytını veya büyük baytını tek bir bayt olarak alıp okumak istediğimizde kullanabileceğimiz iki sembol vardır. Bunlar büyük ve küçük işaretleridir (< >) küçük işareti küçük baytı bize verirken büyük işareti büyük baytı verir. Bu işaretler ACME için yazdığınız bütün etiketlerle kullanılabilir.

```
SCREEN = $0400
lda #<SCREEN
ldx #>SCREEN
```

kodu aslında ACME tarafından şu koda çevrilir

```
lda #$00
ldx #$04
```

Evet tekrar dönelim copy_text alt programına müşterinin bu alt proramı çağırırken

```
ldx #<text1
ldy #>text1
jsr copy_text
```

diye çağıracağını söylemiştik. Şimdi bu satırları anlıyor olmalısınız. Text kod içinde herhangi bir adres olduğuna göre 16 bitlik bir sayı. Burada yaptığımız şey, x registerine text1 adresinin küçük baytını y registerine ise büyük baytını yüklemekten ibaret. Yani 16 baytlık bir argümanı iki 8 bitlik registera bölüp alt programımıza geçirmiş oluyoruz.

Copy_text alt programının hemen başında ise bu geçirilen değeri kullanarak alt programın kendi kendisini değiştirmesini sağlıyoruz. Lda $0000,x komutu bellekte ct_loop adresinden başlayan üç baytı kaplıyor. Bu üç baytın ilki lda komutunun kodu (komut kodlarına kısaca opcode denir) ardından da indexli adreslemenin başlayacağı mutlak adresi ifade eden iki bayt var. Bu baytlar ilk başta 0 olacak. Ama daha CPU o satıra varamadan önce biz stx ve sty yaparak o iki baytı müşterimizin bize geçirdiği text1 adresinin küçük ve büyük baytı ile yazıyoruz. Bunu yaparken yine little endian yani küçük baytı öne gelecek şekilde yazıyoruz.

Yanı bu alt program her çağrılışında kendini değiştiripsonra döngüye giriyor yani her çağrılışında kopyalayacağı texti müşteri değiştirebiliyor. Müşterinin göderdiği iki registera bölünmüş argüman alt rutinin başında texti okuyan satırı değiştirip programın istenen texti kopyalamasını sağlıyor.

Şimdi bu tekniklerle beraber ödev programını daha da kısaltalım.

```
* = $c000

jsr clear_screen
ldy #$40
jsr delay
ldx <text1
ldy >text1
jsr copy_text
ldy #$80
jsr delay
ldx <text2
ldy >text2
jsr copy_text
ldy #$c0
jsr delay
ldx <text3
ldy >text3
jsr copy_text
end:
jmp end


clear_screen:
ldx #0
lda #20
cs_loop:
sta $0400,x
sta $0500,x
sta $0600,x
sta $0700,x
inx
bne cs_loop
rts

screen = $0400 + (12*40)

copy_text:
stx ct_loop + 1
sty ct_loop + 2
ldx #0
ct_loop:
lda $0000,x
sta screen,x
inx
cpx #40
bne ct_loop
rts

delay:
ldx #0
d_loop:
dex
bne d_loop
dey
bne d_loop
rts
text1:
!scr "merhaba dunya                              "
text2:
!scr "iste geliyorum                                "
text3:
!scr "bangir bangir                                 "
```

Programcılığın en önemli kavramlarından olan alt program kullanımı ile tanışmış oldunuz. Bir sonraki bölümde aritmetik ve mantık işlemlere değinecek, ve alt programların bir diğer kullanımını yani fonksiyonları tanıtacağım.

Aritmetik İşlemler

Toplama

Önce en temel işlem olan toplama ile başlayalım. C64'te toplama ile ilgili iki komut vardır. CLC ve ADC. Once bunların kodda nasıl göründüğüne bakalım:

```
lda say#1
clc
adc say#2
sta sonuc
```

Bu kod parçası, bellekte sayı1 ve sayı2 adreslerınde bulunan değerşeri toplayıp sonucu "sonuc" adresine yazar. Asıl toplama işlemini gerçekleştiren komut ADCdir. Bu komut akümülatörde olan değer ile argümanında belirtilen değeri (adresleme moduna göre) toplayıp sonucu akümülatöre geri yerleştirir. Fakat bu toplama işlemine aslında bir sayı daha girer: Status registerindeki Carry flagi. Yani adc aslında,

```
A <- A + adc'nin argüman# + Carry
```

İşlemini yapar. Carry bildiğiniz gibi bir veya sıfır olabilir. Bu yüzden eğer iki adet bir baytlık değeri toplayacaksak yuarıdaki kodda olduğu gibi adcden once clc komutunu kullanırız. CLC komutu Carry bitini 0 yapar.

Toplama yapısının niye böyle tasrlandığını ilk bakışta anlayamayabilirsiniz. Bunu anlayabilmek için 16 bitlik sayıları nasıl toplayacağınızı düşünmelisiniz.

```
lda say#1_lo
clc
adc say#2_lo
sta sonuc_lo
lda say#1_hi
adc say#2_hi
sta sonuc_hi
```

Bu kod parcası sayı1_lo ve sayı1_hi adreslerinde sırasıyla küçük ve büyük baytları bulunan 16 bitlik bir sayıyı yine 16 bitlik sayı2 ile toplayıp sonucu "sonuc_lo" ve "sonuc_hi" olarak isimlendirilmiş iki adrese yazıyor. Burada carry flaginin oynadığı rolü görüyoruz.

Önce küçük baytlar toplanıyor. Bu toplamın sonucu 255'i gecerse sonuc 9 bitlik bir sayı oluyor. Küçük baytların toplamından elde edilen bu 9 bitlik sayının küçük sekiz biti akümülatöre konuyor. 9. bit ise carry flag'e kopyalanıyor. Yani toplamadaki "elde var bir" durumunu bilgisayar böyle takip etmiş oluyor. Ardından Büyük baytlar toplanırken clc yapmadan direk adc komutunu kullanıyoruz. Bu komut carry flagi de toplama kattığı için elde bir varsa büyük baytların toplamına bir elde eklenmiş oluyor.

Bunu matematiksel düşünmeyi seven arkadaşlar şöyle de ifade edebilirler. 16 bitlik sayıları 256'lık sayı düzeninde iki basamaklı sayılar olarak düşünebiliriz. Öyle ki, birler basamağındaki rakam (0-255 arası bir rakam) küçük bayt, 256'lar basamağındaki rakam ise büyük bayt olur. Biz toplama işleminde birler basamağını toplayıp elde yi daha sonra 256lar basamağını toplarken hesaba katmış oluyoruz.

Ödev: İki tane 24 bitlik sayıyı toplayan kod parcasını yazın birinci sayı sayı1_lo, sayı1_mid ve sayı1_hi adreslerinde olsun. İkinci sayı sayı2_lo, sayı2_mid ve sayı2_hi adreslerinde olsun. Sonucu sonuc_lo, sonuc_mid ve sonuc_hi adreslerine yazın.

Çıkarma

Tamamen toplama ile paralel bir yapıdadır. CLC ye karşılık SEC(carry biti 1 yapar) ve ADCye karşılık da SBC (argümanı ve carry'nin tersini akumulatorden çıkar ve sonucu akumulatore ve carry'ye yaz)

Örnekle görmek yine sanırım daha kolay. 8 bitlik çıkarma işlemi:

```
lda say#1
sec
sbc say#2
sta sonuc
```

16 bitlik çıkarma işlemi:

```
lda say#1_lo
sec
sbc say#2_lo
sta sonuc_lo
lda say#1_hi
sbc say#2_hi
sta sonuc_hi
```

İki İle Çarpma

6510 komut setinde maalesef tek bir komutla genel bir çarpma imkanı yoktur. Zaten çarpma ve bölme işlemleri, CPUlarda tasarım bakımından hep toplama ve çıkarmadan daha zor işlemler olmuşlar ve CPUların fiyatlarında artmaya sebep olmuşlardır.

Bu yüzden çarpma işlemi için başka teknikler kullanılır. Bu teknikler çoğu zaman yavaş, bazen isabetsiz(küçük hatalarla sonuc veren) veya anlaşılması şu an için zor olabilecek ileri seviye konulardır.

Fakat biz genel çarpmadan şu an bahsetmesek de 2 ve 2'nin kuvvetleri ile çarpmadan bahsedeceğiz. Çünkü bu işlemleri çok hızlı yapmanın kolay bir yolu vardır: bit kaydırma (shift)

İkilik düzendeki herhangi bir sayının bütün bitlerini sola doğru bir kere kaydırırsanız sayı iki ile çarpılmış olur. Mesela

```
%01101010 x 2 = %11010100
%00101111 x 2 = %01011110
```

Dolayısıyla bitleri n kere sola kaydırırsanız, sayıyı 2 üzeri n ile çarpmış olursunuz. Aynı şekilde bitleri n kere sağa kaydırırsanız sayıyı 2 üzeri n'e bölmüş olursunuz.

Bu shift işlemlerini yapmaya yarayan 4 komut vardır. ASL ve ROL bitleri sola kaydırır. ASL kucuk 7 biti sola kaydırıp en kucuk basamaga 0 koyar. En buyuk basamaktan dısarı cıkan biti de carryiye kopyalar. ROL ise kucuk 7 bıtı bır bıt sola kaydırır. En kucuk basamaga carry'yı koyar en buyuk basamaktan cıkan biti ise carry'ye koyar.

Dikkatli okuyucular buradaki carry kullanımının yine 8 bitten büyük sayılarla alakalı olduğunu anlayacaklar. Hemen örneklere bakalım. İşte 8 bitlik sayının 2 ile çarpımı:

```
lda say#1
asl
sta sonuc
```

16 bitlik bir sayı için ise:

```
lda say#_lo
asl
sta sonuc_lo
lda say#_hi
rol
sta sonuc_hi
```

İki İle Bölme

Aynen çarpma komutarında olduğu gibi kullanılan LSR ve ROR komutları mevcuttur. LSR baytın büyük 7 bitini sağa kaydırır. 8.basamağa 0 yazar. 1. basamaktan çıkan biti de carry'ye kopyalar. ROR ise LSR den farklı olarak 8. basamağa carrydeki değeri yazar. 8 bitlik sayının bölünmesi:

```
lda say#
lsr
sta sonuc
```

16 bitlik sayının bölünmesine dikkat edin. Bu sefer önce büyük baytı sağa kaydırıyoruz. Çünkü onun birinci bitinin küçük baytın 8. bitine kaymasını istiyoruz:

```
lda say#_hi
lsr
sta sonuc_lo
lda say#_lo
ror
sta sonuc_lo
```

Ödev

sayı_lo ve sayı_hi adresinde duran 16 bitlik bir sayıyı 8 ile çarpıp sonucu sonuc_lo ve sonuc_hi adresine yazan bir kod yazın.

Mantıksal İşlemler

Kursun en başında belirttiğim gibi AND OR ve Exclusive OR işlemlerinin ne olduğunu bildiğinizi varsayıyorum. Fakat burada bu işlemleri 6510 assembly de gerçekleştiren komutları tanıttıktan sonra kısaca bu komutların en sık kullanıldığı birkaç yerden bahsedeceğim.

AND komutu akümülatördeki değer ile argümanda belirtilen değerleri "AND"(VE) işlemine sokar ve sonucu akümülatöre geri yazar:

```
lda say#1
and say#2
sta sonuc
```

ORA komutu akümülatördeki değer ile argümanda belirtilen değerleri "OR"(VEYA) işlemine sokar ve sonucu akümülatöre geri yazar:

```
lda say#1
ora say#2
sta sonuc
```

EOR komutu akümülatördeki değer ile argümanda belirtilen değerleri "Exclusive OR" işlemine sokar ve sonucu akümülatöre geri yazar:

```
lda say#1
eor say#2
sta sonuc
```

AND ve ORA nın çok sık kullanıldığı durumlar vardır. Şimdi biraz bunlardan bahsedelim. Çoğu zaman bir baytın içindeki bir veya birkaç bite erişip sadece onlar üzerinde değişiklik yapmak isteriz. Bu VIC gibi çeşitli giriş çıkış çiplerinin registerleri ile uğraşırken çok olur. Aynı registerin bir biti ekranı kapatıp açarken başka 3 biti y yönünde kayma, başka bir bit bitmap moduna geçme gibi görevlere sahip olabilir.

Bu durumlarda Biz sadece istediğimiz bitleri sıfırlamak için and işlemini kullanabiliriz. Örneğin

```
lda register
and #%11101111
sta register
```

bu kod registerin diğer bitlerine dokunmadan yalnızca 5. biti 0 yapar. AND komutunda bu sefer immediate adresleme kullandığımıza dikkat edin.

Tersine bir biti bir yapmak istersek o zamaz da ora kullanırız.

```
lda register
ora #%00010000
sta reg#ster
```

bu kod ise diğer bitlerde değişiklik yapmadan yalnızca 5.biti 1 yapar. Bazen de registerdaki bir grup biti kontrol etmek isteyebiliriz. Mesela registerin üst 4 bitine değişiklik yapmadan alt 4 bitine 6 değerini yazmak istiyoruz.

```
lda register
and #%11110000
ora #6
sta register
```

bu kod once regısterde okuduğumuz değerin alt 4 bitini 0 yapıyor sonra da istediğimiz değeri ora ile o bitlere yazıyoruz.

Son olarak EOR komutu ise genelde bir takım bitleri tersine çevirmek için kuulanılır. Örneğin

```
lda register
eor #%01000001
sta register
```

kodu regısterın 1. ve 7. bitlerini tersine çevirir. Böylece registerin aynı iki değer arasında sürekli gidip gelmesi sağlanılabilir (bu da bazen interlace resim gösterirken kullanılır)

EOR komutunun çok önemli bir kullanım alanı da filled polygon çizerken kullanılan EOR-filling adı verilen methodda oynadığı roldür.Buna daha ileride değineceğiz.

Ödev

"register" adresindeki bir registerın 2.3.4. bitlerine 5 değerini yazıp 8. bitini tersine çeviren bir kod yazın.

İki Kullanışlı Komut: INC VE DEC

Bu iki komutu da aradan çıkarmak için uygun bir zaman. Bu komutlar argümanlarında belirtilen bellek adresindeki değeri bir artırıp bir azaltmaya yararlar. Güzel olan özellikleri bunu yaparken bir registerı kullanmanıza gerek bırakmamalarıdır.

Değer Döndüren Alt Programlar: Fonksiyonlar.

Daha önce bir alt programa nasıl argüman geçireceğimizi tartışmıştık. Çoğu zaman programdan alt programa bilgi geçirme ihtiyacımız olduğu kadar alt programdan geriye de değer döndürme ihıtyacımız olabilir. Yine aynı şekilde registerları veya bellekteki bazı adresleri bu iş için kullanabiliriz. İşte bir alt program eğer çağrıldığı yere değer döndürürse ona fonksiyon da denir.Hemen bir örneğe bakalım.

Örnek: x ve y regısterlerında bulunan bir adresteki ardışık dizilmiş 4 tane 8 bitlik sayının ortalamasını alıp döndüren bir fonksiyon yazın.

```
get_average:
    stx adder + 1
    sty adder + 2
    ldx #0
    lda #0
loop:
    clc
adder:
    adc $0000,x
    inx
    cpx #4
    bne loop
    lsr
    lsr
    rts
```

Bu kod önce x ve y registerlerindeki adrese göre kendini değiştiriyor. Ardından 4 baytı okuyup topluyor. En son da sonucu 4'e bölüp akümülatörde bırakarak geri dönüyor. Müşteri bu rutini çağırdığı zaman geri dödüğünde akümülatördeki değeri kullanarak başka işlemler yapabilir. Özellikle matematiksel işlem yapan ve bir sayı döndüren fonsiyonlarda o sayıyı akümülatörde döndürmek iyi bir seçimdir. Çünkü müşteri akümülatördeki sayıyı doğrudan başka aritmetik veya mantık işlemlerine sokabilir.

(Not: yukarıdaki program 4 sayının toplamı 256yı gecerse çalışmaz. Ama basitliğini bozmamak için şu an böyle bıraktım.)

Text Scroll: Bir Scene Klasiği

Scene denince kuşkusuz akla ilk gelen programcılık konusu kayan yazılardır. Bunlar aslında ilk introları tanımlayan temel parçalar olarak teknik yönden de önemli olmakla beraber asıl önemleri belki de sosyolojiktir. Bugün bile iyi yazılmış scroll textler çoğu old skool scener'ı heyecanlandırır. Fakat geçmişte scroll testler, scene içindeki başlıca haberleşme kanalıydı. İntrolarda gruplar birbirlerine mesajlar yazar, yaptıklarıyla övünür, düşmanlarına nefret, dostlarına selam gönderir, rakiplerine meydan okurdu.

Şimdi ilk scroller'ınızı yazdıktan sonra içine yazacağınız mesajı düşünürken, biraz biraz o scroll mesajlarının büyüsünü hissedeceğinizi tahmin ediyorum. Bir programı bitirmiş, ekranda sizin yaptığınız çalışan bir parça kod olacak ve içinde siz ne isterseniz o mesaj geçecek.

Yumuşak Kayma (Smooth Scroll)

Aslında gerçek anlamda scroller rutinleri yazıları pixel pixel kaydırırlar. Buna smooth scroll (yumuşak kaydırma) denir. Ancak biz ilk etapta karakterlerı VM içinde kaydıracağız. Yani harfler 8 pixellik atlamalarla kayacak.

Renk Belleği

Bu esnada VIC ile ilgili basit iki yeni şeyi daha öğreneceksiniz. Bu öğrendiklerinizi de kullanarak, şimdi yapacağınız program bayağı introya benzeyecek.

Bunlardan birincisi ekran rengini değiştirmek olacak. Daha önceden $d020 adresindeki VIC registerini kullanarak çerçeve rengini değiştirebilmiştik. Aynı şekilde VVIC çipinin $d021 adresli registeri de ekranın rengini değiştirir. Hemen şu programı yazarak sonucu görebilirsiniz.

```
*=$c000
lda #0
sta $d020
lda #2
sta $d021
jmp *
```

bu program ekran rengini kırmızı çerçeve rengini ise siyah yapar.

Hazır renklerden bahsederken, sizi renk balleği yani Color RAM (kısaca CR) ıle tanıştırmanın zamanı da geldi. CR $d800 adresinden başlayan 1000 baytlık bölgedir. Bu bölge biraz VM'ye benzer. C64 text modundayken, VM nasıl ekranda hangi harfin çıkacağı bilgisini tutuyorsa, CR de ekrandaki her pozisyona basılan harfin hangi renk olacağı bilgisini tutar. VM nin içinde karakter kodları tutuluyordu. CR'nin içinde ise renk kodları tutulur. Örneğin,

```
lda #1
sta $0400
lda #0
sta $d800
```

kodu, ekranın sol ust koşesine siyah bir A harfi basar. Gelin hemen renkli bir program yazalım.

```
!to "out.prg"
* = $c000

VM_SATIR_0 = $0400 + (0 * 40)
VM_SATIR_1 = $0400 + (1 * 40)
VM_SATIR_2 = $0400 + (2 * 40)
VM_SATIR_3 = $0400 + (3 * 40)
VM_SATIR_4 = $0400 + (4 * 40)
VM_SATIR_5 = $0400 + (5 * 40)
VM_SATIR_6 = $0400 + (6 * 40)
VM_SATIR_7 = $0400 + (7 * 40)

CR_SATIR_0 = $d800 + (0 * 40)
CR_SATIR_1 = $d800 + (1 * 40)
CR_SATIR_2 = $d800 + (2 * 40)
CR_SATIR_3 = $d800 + (3 * 40)
CR_SATIR_4 = $d800 + (4 * 40)
CR_SATIR_5 = $d800 + (5 * 40)
CR_SATIR_6 = $d800 + (6 * 40)
CR_SATIR_7 = $d800 + (7 * 40)



lda #0
sta $d020
sta $d021

ldx #0
loop: lda text_source,x
sta VM_SATIR_0,x
sta VM_SATIR_1,x
sta VM_SATIR_2,x
sta VM_SATIR_3,x
sta VM_SATIR_4,x
sta VM_SATIR_5,x
sta VM_SATIR_6,x
sta VM_SATIR_7,x
lda color_source,x
sta CR_SATIR_0,x
sta CR_SATIR_1,x
sta CR_SATIR_2,x
sta CR_SATIR_3,x
sta CR_SATIR_4,x
sta CR_SATIR_5,x
sta CR_SATIR_6,x
sta CR_SATIR_7,x
inx
cpx #40
bne loop

end: jmp end

text_source:
!by $20,$20,$20,$20,$20,$20,$20,$20
!by $20,$20,$20,$20,$20,$20
!scr "merhaba dunya"
!by $20,$20,$20,$20,$20,$20,$20,$20
!by $20,$20,$20,$20,$20

color_source:
!by $00,$00,$00,$00,$00,$00,$00,$00
!by $00,$00,$00,$00,$00,$00

!by $09,$02,$08,$0a,$0f,$07,$01,$07
```

```
!by $0f,$0a,$08,$02,$09

!by $00,$00,$00,$00,$00,$00,$00,$00
!by $00,$00,$00,$00,$00
```

Bu programı çalıştırdığınızda ekranda ilk 8 satırda renkli harflerle MERHABA DUNYA mesajını göreceksiniz. Programın başında gerek VM gerekse CR için satır başlarını acmeye hesaplattırıp etiketlere atıyoruz. Programımız ekran ve çerçeve renklerini siyah yaparak başlıyor. Ardından bır döngü içinde yazıyı okuyup VM'deki 8 satıra kopyalarken aynı zamanda renkleri okuyup CR'deki 8 satıra kopyalıyoruz.

Kodun arkasından gelen text source ve color source baytlarında yeni bir acme komutunu daha öğreniyorsunuz. !by veya !byte komutu tam olarak o adrese istediğiniz baytları yerleştirmenizi sağlıyor. Bu örnekte text_source etiketi $c048 adresine denk geliyor. !by komutu ve arkasından gelen değerleri kullanarak $c048 adresinden itibaren ilk 14 bayta boşluk karakterini yazıyoruz. Ardından gelen 13 bayta !scr komutunu kullanarak 13 karakterlik "merhaba dünya" mesajını oluşturan karakterlerin kodlarını yerleştiriyoruz. Ardından da satırın sonuna kadar kalan 13 karakteri de boşluk karakteri ile doldurmak için 13 adet daha boşluk kodunu !by komutu ile dızıyoruz. Yani text_source (= $c048) etiketinden itibaren 40 bayt yerleştirmiş oluyoruz.

Ardından color source etiketi $c048den 40 bayt sonra geliyor ve dolayısıyla $c070'e eşit oluyor. Burada da önce 14 bayt 0 (siyah) dizdikten sonra kahverengiden beyaza kadar bir renk geçişi elde edip sonra tekrar kahverengiye dönecek şekilde 13 tane renk kodu diziyoruz. Bu kodlar sırasıyla: Kahverengi, kırmızı, turuncu, açık kırmızı, açık gri, sarı, beyaz, sarı, açık gri, açık kırmızı, turuncu, kırmızı, kahverengi

Bu onüç renk baytından sonra tekrar 13 baytlık siyah yerleştiriyoruz.

Karakterleri Kaydırmak

Bellekteki bir texti kayarak VM içinden geçirmek şimdiye kadar uğraştığımız problemlerden oldukça farklı. Çözümün çok zor olmadığını göreceksiniz. Ve bu çözümü gördükten sonra, mevcut bilgilerinizle pekçok şey yapabileceğinizi göreceksiniz. Zaten bu bölümden sonra oldukça güzel bir intro yapacağız. Ama önce scroll konusunu masaya yatırıyoruz.

```
!to "out.prg"
* = $c000

VM_SATIR_0 = $0400 + (0 * 40)

lda #0
sta $d020
sta $d021

loop1:

ldx #0
loop2:
lda VM_SATIR_0+1,x
sta VM_SATIR_0,x
inx
cpx #39
bne loop2

read:
lda text_source
sta VM_SATIR_0+39
inc read+1

ldy #$40
ldx #0
delay:
dex
bne delay
dey
bne delay

end: jmp loop1

!align 255,0
text_source:
!scr "merhaba dunya... iste karsinizda yazdigim "
!scr "ilk scroll rutini... henuz smooth degil ama "
!scr "onemli degil. onu da bi kac gune kadar yapmis "
!scr "olacagim zaten. greetings to ali, veli, 49, 50..."
!scr "the new codemaster signs off..."
!fill $2c,$20
```

Programımız yine ilk önce ekranı ve çerçeveyi siyah yapıyor. Daha sonra programın asıl döngüsüne giriyoruz. Döngü içinde programın yaptığı üç temel iş var:

- Yazının kayacağı satırdaki adresleri 0'dan 39'a numaralandırdığımızı düşünelim. Programın yaptığı ilk iş, 1 ila 39 nolu adreslerdeki toplam 39 karakteri, 0 ila 38 nolu adreslere kopyalamak. Yani satırdaki bütün harfleri bir pozisyon sola kaydırmak. Bunun sonucu olarak en soldaki 0 nolu adresteki karakter siliniyor. Çünkü 1 nolu adresteki karakter onun üzerine yazılmış oluyor. Aynı zamanda 38 ve 39 nolu adreslerdeki karakterler birbirinin aynı oluyor. Çünkü 39 nolu adresteki karakter 38 adresine kopyalandı. Şimdi 39 nolu adrese kayan yazının yeni harfinin girmesi gerekli
- Bu 39 karakterlik kaydırmadan sonra read etiketli kodla kayacak olan mesajın yenı harfi okunuyor ve ekranda 39 nolu adrese basılıyor. Kod donup tekrar buraya geldiğinde lda ile yazının bir sonraki harfinin okunması gerekli. Bu yüzden inc read+1 ile lda komutunun argumanı olan mutlak adresin küçük baytı bir artırılıyor. Yani program kendini değiştiriyor.
- Bütün bunlar çok hızlı olduğu için yazının kaymasının okunabilecek bir hıza indirilmesi gerekiyor bunu da döngünün sonunda basit bir delay ile hallediyoruz. Delayin ardından jump komutu ile sonsuz döngünün başına atlyoruz.

Kayacak yazıyı belleğe text_source etiketiyle yerleştirmeden önce bir yeni acme komutu ile daha tanışıyosunuz. !align komutu kendisinden sonra gelen baytların bellekte "istediğimiz gibi" bir adresten başlamasını sağlıyor. Bunun anlamı şu: Bu programda biz, text_source'un tam bir "page" başından başlamasını istiyoruz. Peki page nedir?

C64'ün 64K'lık belleğini 256 baytlık bloklara bölerseniz 256 tane blok elde edersiniz. Bu blokların herbirinin adresi 256'nın katları olur. Bu blokların her birine page (sayfa) adı verilir. Bu pagelerin ilki $0000 - $00ff arasındaki baytları kapsar. Bi sonraki page $0100 - $01ff arasındaki baytları kapsar.

Biz kaymasını istediğimiz yazıların page başından başlamasını istiyoruz. Bunu sağlamak için !align 255,0 komutu kullanılıyor. 255 ve 0'ın ne demek olduğuna şu an takılmayın. Bilmeniz gereken tek şey kodun içinde !align gördüğünüz anda bir sonraki parcanın gerekirse arada boşluk bırakılıp en yakın page başından başlayacağı. Bu programda align satırı olmasa text_source etiketi $c02b'ye eşitlenecek ve text mesajını oluşturan karakter kodları o adresten itibaren yer alacaktı. Ama o satır sayesinde bir sonraki en yakın page başı olan $c100 adresine yerleştirildiler.

Peki biz text_source'u page başına koymak için niye uğraşıyoruz? Bunun cevabı read etiketindeki lda satırında. Bu satırdaki adres değerinin küçük baytını sürekli artırdığımızı hatırlıyorsunuz. Bu demektir ki bu bayt 255'e kadar artacak ve ardından 0'a dönüp yeniden artmaya devam edecek. Biz eğer text_source'un $c02b olmasına izin verseydik, o zaman en son c0ff deki harfi okuduktan sonra adresin küçük baytı artıtıldığında 0 olacaktı. Dolayısyla bir sonraki read işlemi lda $c000 şeklinde gerçekleşecekti. Orada ise code var mesaj yok. Biz c100 a yerleştirmekle yazının sağlıklı bir biçimde başa dönmesini sağlamış olduk. Programın şu halinde lda satırı c100 ile c1ff arasını okuyup tekrar c100'a donerek çalışıyor.

Son olarak bir acme komutu ile daha tanışıyorsunuz: !fill. Bu komut adından da tahmin edebileceğiniz gibi programınızın kapladığı bellekteki bir bölgeyi bir değerle doldurmaya yarıyor. Burada c100'dan başlayan mesaj textinin sonundan page sonuna (c1ff) kadar boşluk karakteri ile doldurmak için kullanıyoruz. !fill iki argüman alıyor. İlk argüman kaç baytlık bölgeyi dolduracağını ikinci argüman da ne ile dolduracağını belirtiyor.

İlk Pre-İntro

Yaptığınız birşeye gerçekten intro diyebilmeniz için içinde müzik logo, raster barlar ve smooth scroll olmalı. Ama bunlar olmada yapacağınız ve introya benzeyen şeyler olacak bunlara o yüzden pre-intro diyeceğim. Artık ilk pre-intronuzu yapmaya hazırsınız. Birazdan vereceğim programda yeni hiç birşey yok. Sadece bugün bahsettiğimiz CR ve scroll kodları ile önceki derslerden tanıyacağınız ekran silme rutinini kullanarak ilk pre-intronuzu yapabilecek durumdasınız. İşte kod:

```
!to "out.prg"
* = $c000

VM_SATIR_L = $0400 + (4 * 40) + 9
VM_SATIR_S = $0400 + (12 * 40)
CR_SATIR_L = $d800 + (4 * 40) + 9
CR_SATIR_S = $d800 + (12 * 40)

lda #0
sta $d020
sta $d021

jsr clear_screen
jsr color_setup
jsr logo_setup

loop1:

ldx #0
loop2:
lda VM_SATIR_S+1,x
sta VM_SATIR_S,x
inx
cpx #39
bne loop2


read: lda text_source
sta VM_SATIR_S+39
inc read+1

ldy #$40
ldx #0
delay: dex
bne delay
dey
bne delay

end: jmp loop1

clear_screen:
ldx #0
lda #$20
cs_loop:
sta $0400,x
sta $0500,x
sta $0600,x
sta $0700,x
inx
bne cs_loop
rts


color_setup:
ldx #0
stp_loop:
lda scroll_colors,x
sta CR_SATIR_S,x
inx
cpx #40
bne stp_loop

logo_setup:
ldx #0
lloop:
lda logo_text,x
sta VM_SATIR_L,x
lda logo_colors,x
sta CR_SATIR_L,x
inx
cpx #21
bne lloop
rts

!align 255,0
text_source:
!scr "merhaba dunya... iste karsinizda yazdigim "
!scr "ilk scroll rutini... henuz smooth degil ama "
!scr "onemli degil. onu da bi kac gune kadar yapmis "
!scr "olacagim zaten. greetings to ali, veli, 49, 50..."
!scr "the new codemaster signs off..."
!fill $2c,$20

scroll_colors:
!by $06,$0b,$04,$0e,$0f,$01,$01,$01
!fill 24,1
!by $01,$01,$01,$0f,$0e,$04,$0b,$06

logo_colors:
!by $09,$09,$02,$08,$0a,$0f,$07,$01
!by 1,1,1,1,1
!by $01,$07,$0f,$0a,$08,$02,$09,$09
logo_text:
!scr "6510 assembly has you"
```

Bu kodla oynayın. Renkleri mesajları değiştirin. Logonun ekranda durduğu yerle oynayın. Bu programda ne olduğunu tam olarak anladiğinizdan emin olun. Sorularınız olduğunda mutlaka sorun.

Ödevler

Gerçekten kafasını yormak isteyen okuyucular için beyin cimnastiği yapabilecekleri iki soru ile bu bölümü tamamlıyorum.

Ödev: Scroll rutininde kayan mesaj 256 bayttan uzun olursa ne olur? Programın çalışması için ne gibi değişiklikler düşünürdünüz?

Ödev: Pre-intro'da logonun renklerini kaydırarak çok çarpıcı bir görüntü elde edebilirsiniz. Bunu scrolle aynı anda nasıl yapabilirsiniz?

Bu iki sorunun da hayli zor olduğunu belirtmek isterim. Bunları cevaplamanızdan ziyade bunların üstüne kafa yormanız önemli. Böylece bunlara cevap olabilecek şeyler öğrendiğiniz zaman kafanızda hemen gereken ilişkilendirmeyi yapabilirsiniz.

Yığıt (Stack)

Stack (Yığıt) kavramı Bilgisayar bilimleri alanından gelen soyut bir veri yapısıdır. Öyle ki, temel amacı içine bi takım objeler saklamak olan bu yapının temel özelliği içine son koyduğunuz objeyi ilk geri almanızdır. Yani bir stack'iniz varsa iki işlem yapabilirsiniz: PUSH yani stack'e obje saklamak, ve POP yani stack'ten obje almak. Başka türlü bir erişim yoktur. Dolayısıyla demin soylediğimiz gibi stack'e arka arkaya a, b ve c objelerini push ederseniz, ardından pop ettiğinizde elinize c objesi gelir. Bir daha pop ederseniz b objesini alırsınız. Yani son giren ilk çıkar (last in first out:LIFO)

Stack yapısı daha sonraları CPU mimarilerinde kullanılmaya başlamıştır. Pek çok CPU bellekte bazı bölgeleri stack olarak kullanır. Bu amacla genelde CPUlarda stack pointer adı verilen registerler olur. Bu registerlerin neye yaradığını anlatacağım.

6510'da da $0100 - $01ff arası bölge stack olarak kullanılır. 6510'un içinde Stack Pointer (kısaca SP) adı verilen 8 bitlik bir register bulunur. Bu register ilk başta $ff değerindedir. yani stackin son adresini gösterir. Bu esnada stack boştur yani içinde saklanan hiç bir bayt yoktur. Eğer bir bayt push edilecek olursa stack bölgesinde (başka bir deyişle stack page'de) SP'nin gösterdiği adrese ($01ff) kopyalanır. Hemen ardından SP'nin değeri otomatik olarak bir azalır ve $fe olur. Böylece bir sonraki push komutunda push edilen bayt $01fe adresine kopyalanır.

Pop komutu geldiğinde ise SP bir artırılır ve böylece son push edilen baytı geri döndürür. Dikkat ederseniz, stack içine obje push edildikçe objeler page'in sonundan geriye doğru stack page'e kopyalanır ve bu esnada SP hep geriye doğru ilerler. Pop edildikçe de SP page sonuna dogru ilerler ve boylece stack boşalmış olur.

Programcı ısterse stack bölgesini geçici olarak veri saklamak için kullanabilir. Bunun için iki komut sağlanmıştır: PHA ve PLA

PHA akümülatördeki değeri stacke push etmeye yarar. Komuttan sonra akümülatördeki değer hala akümülatörde kalmaya da devam eder. PLA ise stack'ten son push edilen değeri alıp akümülatöre yükler.

JSR ve RTS Komutlarının Stack Kullanımı

Stack kullanmaya niyetlendiğinizde bilmeniz gereken çok önemli bir konu var. Hatırlarsanız, JSR komutundan bahsederken, CPU bu komutu gördüğü zaman o anda bulunduğu yeri kaydeder ve alt programa atlar demiştik. İşte bu kaydetme işi stack'te olur.

CPU JSR komutunu gördüğü zaman jsrden bir sonraki komutun adresini oluşturan iki baytlık değeri (küçük ve büyük baytlar) stack'e push eder. Daha sonra alt program çalışıp bittiğinde, RTS komutuna gelindiği zaman, CPU stack'ten iki değeri pop edip bu iki değeri Program Counter'a yazar. Böylece program kaldığı yerden devam eder.

Dikkat ederseniz bu yapı alt programlardan, başka altprogramları çağırabilmenizi de sağlar. Mesela bir JSR ile bir altprograma atladığınızda dönüş adresi stack'e iki bayt olarak push edilir. Ardından rts demeden once bir jsr komutu daha gecerseyenı bı alt programa atlanır ve ıkı yenı bayt daha stacke push edilir. Daha sonra rts komutu gorulurse, yapılan son jsr yüzünden stack'e push edilmiş olan adres baytları ilk pop edilecek (son giren ilk çıkar). böylece içiçe pekçok alt program çağrılabilir. Bunun sayesinde programlarınız bir ana programdan çağrılan alt programlar, ve onların çaırdığı daha alt programlar şeklinde hiyerarşık bir yapıda tasarlanabilir.

Burada iki tehlike söz konusudur:

- stack overflow: yani stack taşması. Bu durum çok fazla push işlemi sonucu SPnin 0'a kadar azalıp daha sonra $ff'e dönerek stackteki önceden push edilmiş dataların üzerine yazmaya başlaması. Bunun sonucu çok büyük olasılıkla programın kilitlenmesi olacaktır.
- stack'te dengelenmemiş push ve poplar. Her alt programda kac tane bayt push edilmişse o kadar beytın pop edilmesi gerekir. Mesela bir alt programda bir bayt (mesela 0 olsun) push eder ve rts'den önce pop etmezseniz şu durum oluşur. Stackteki son üç bayt sırasıyla 0, donüş_adresi_küçük_bayt, ve donüş_adresi_büyük_bayt olur. rts komutu gelsiği anda stackten dönüş adresi olarak son iki baytı yani yanlış iki baytı alır ve yanlış adrese döner. Bunu da sonucu çoğu zaman programınızın kilitlenmesi olacaktır.

Bu yüzden programınızı yazarken, bir alt programdan dönmeden önce mutlaka push ettiğiniz her baytın pop edildiğini ve fazladan bir pop işlemi olmadığını kontrol etmelisiniz.

Kesintiler (Interrupt)

İşte geldik Bilgisayar bilimlerinin en meşhur konularından birine. Genelde bu kouları öğrenen insanların hep biraz zorlandığı bir konudur bu. Bu yüzden çok dikkatli okuyun. Anlaşılmayan yerleri mutlaka bilen birilerine sorun. Daha sonra VIC Programlama kursunda öğreneceğimiz efektlerin neredeyse tamamı interruptlar kullanılarak yapılacak.

Interrupt Nedir:

Bir CPU'nun bir programı nasıl çalıştırdığını artık biliyorsunuz. Bellekte bir bölgede duran bir grup bayt CPU'nun PC registerinin adım adım onalrın üzerinde ilerlemesi ile komutlar ve bu komutlara giren argümanlar olarak okunuyor.yanı CPU her defasında bellekten PCnin gösterdiği adresteki komutu alıyor, o komutta belirtilen işleri yapıyor. Bunun sonucunda PC ya bir sonraki komutu gösterecek şekilde artıyor ya da bir dallanma olacaksa dallanılan adres PC'ye yükleniyor ve CPU tekrar PCnin gösterdiği adresteki komutu alıp devam ediyor. Bu şekilde, yazılan programlar tasarımlarından gelen akış planına göre CPU terefından çalıştırılmış oluyor.

İlk CPU'larda insanlar sadece bunu yeterli görüyor. Fakat kısa süre içinde bu CPU'larla gerçek dünyada çalışan bazı cihazlar yapılmaya çalıştığında bir problem ortaya çıkıyor. Cihaz ister bir bilgisayar olsun, ister bir fabrika robotu ister bir ölçü cihazı aynı problem tekrar ediyor. Cihazın kendine verilen programın izin verdiği akış dışında bazen dış dünyada olan çeşitli olaylara tepki vermesi gerekiyor. Mesela fabrika robotunda CPUnun dışındaki harici bir zamanlama devresinin her 5 saniyede bir robota bir sinyal göndermesi ve robtun CPUsunun bu sinyali alınca o anda yaptığı işi bırakıp başka birşey yapması istenebiliyor. Veya bir bilgisayarda kullanıcı bir tuşa bastığında CPU o anda çalıştırdığı programda her ne yapıyor ise durup çıkması gerekebiliyor.

İlk başlarda bunu programın tasarımını değiştirerek yapabilmeyi deniyor insanlar. Mesela fabrika robotunun programında periyodik aralıklarla zamanlayıcıdan gelen sinyali kontrol etmek veya bilgisayarda belirli aralıklarla çıkış tuşunu kontrol etmek gibi.

Fakat bu sefer bir de bakıyorlar ki, bu çözüm çok verimsiz. CPU vaktinin çoğunu bir sinyali ve ya başka bir olayı kontrol ederek boşa harcıyor. CPUların performanzı pahallı ve oönemli bir kaynak olduğu için bişeyi sürekli kontrol edip olmasını bekleme metodundan (ki buna "polling" denir) farklı bir çözüm düşünülüyor.

CPU lara yeni bir özellik ekleniyor. CPUnun pinlerinden biri bu işe atanıyor. CPU normal şekilde bir programı çalıştırırken, dış dünyadan bu pinden bir sinyal gönderilirse, o zaman CPU donanımı özel bir tepki veriyor. Bu tepki tam olarak şu oluyor.

- CPU o an bir komutu uygulamanın ortasında ise o komutu bitirene kadar bi değişiklik olmuyor.
- O komut bittiği anda CPU o anki PC değerini stack'e saklıyor. Ardından o anki Status Registerinin değerini de aynen stack'e push ediyor.
- Daha sonra o CPU için tanımlanmış bellekte bilinen bir adrese bakıyor. O adresten iki baytı alıyor ve PC'ye yazıyor.
- PCnin gösterdiği adresteki kodu çalıştırmaya başlıyor

Böylece dış dünyadan gelen sinyale karşı CPU o an yaptığı işi ilk fırsatta bırakıp başka bir program parçasını çalıştırmaya başlıyor. İşte bu olaya interrupt deniyor. CPunun bir interrupt gerçekleştiği zaman çalıştırdığı kod parçasına interrupt rutini diyoruz. Bir süre interrupt rutini çalıştıktan sonra genelde o programı bıtıren "interrupttan geri dön" anlamına gelen bir komut oluyor her CPUda. CPU bu dönüş komutunu gördüğü zaman daha önce stack'e push ettiği status registeri ve PC adresini pop edip CPUnun interrupttan önce son kaldığı yere dönmesini sağlıyor.

Yukarıda üçüncü maddede belirtilen sabit adrese genelde interrupt vektörü deniyor. Vektör başka bir yere bakan (başka bir adresi içinde taşıyan baytların adresi) adres anlamına geliyor. Programcı programını yazıarken ınterrupt gelmesı halınde olmasını istediği şeyleri halletmek üzere bir interrupt rutini yazıyor. Bu interrupt rutininin adresini de interrupt vektörü olan adresteki iki bayta yazıyor.

Daha sonraları CPUlar geliştikçe daha çok interrupt vektörü ve daha çok interrupt pinine de izin veren tasarımlar ortaya çıkıyor. Böylece CPUlar birden fazla değişik olaya farklı acil tepkiler gösterebilir hale geliyor.

Fakat interruptların yine de ana program tarafından kontrol altında tutulması özelliği de CPUlarda sağlanıyor. Yani CPUlar bazı interrupt kaynaklarını geçici süreliğine dinlemeyi durdurup, o kayanktan interrupt sinyali gelse bile interrupt rutinine atlamadan devam edebiliyor. Bunun niçin gerekli olduğunu daha sonra örnekler üzerinde görebileceksiniz. Böyle bir interrupt kaynağını geçici süre dinlememeye o interruptı "maskelemek" deniyor.

6510'da Interruptlar

6510 CPUsunda iki tane interrupt kaynağı var. Başka bir deyişle dış dünyadan 6510a interrupt oluşturmak amaçlı sinyal göndermek için kullanabileceğiniz iki tane pin var. Bunlara IRQ ve NMI deniyor.

IRQ (Interrupt request) ıstendiğinde maskelenebilen ve programlanabilen, yani bizim işimize yarayıp en çok kullanacağımız interrupt çeşidi olacak.

NMI (Non maskable ıinterrupt) ise oluşumuna engel olamadığımız dolayısıyla bizi çok ilgilendirmeyen bir interrupt. Dolayısıyla şu an NMI'lara kafa yormanıza gerek yok.

6510'da IRQ'ları maskelemek ve maskelemeyi kaldırmak için iki komut mevcuttur: SEI ve CLI. Bu komutlar argüman almazlar. SEI komutunu vermenizden itibaren artık hiçbir IRQ sinyali prog-

ramınızı bölemez. Ta ki CLI komutuna kadar

6510 Assembly'de ınterruptlarla ilgili öğreneceğiniz so komut da RTI'dır. Bu komut interrupt rutininin sonuna konulr ve CPU bu komutu görünce stackten PC ve status registerini pop ederek iCPUnun interrupttan önceki duruma dönmesini sağlar.

C64'ünüzde 6510 çipinin IRQ bacağına sinyal gönderebilen üç tane çevre çipi vardır. Bunlar VIC, CIA1 ve CIA2 çıpleridir. CIA çipleri klavye okuma, disket sürücü ve joystickleri okuma gibi işler için kullanılır. Hayatınızın büyük bir bölümünde CIA interruptlarını kullanmayacaksınız. O yüzden ilk etapta onları da unutabilirsiniz.

Bir demoscene programcısı için hayattaki en önemli şeylerden biri VIC ile 6510 arasındaki etkileşimdir. Scene programlarınızın çoğunda VIC'in bazı registerlerini, çeşitli koşullarda 6510'a interrupt sinyali gönderecek şekilde programlayacaksınız. Ardından da bu sinyallerin sonucu, VIC'in bazı registerlerine, bazı yeni değerler yazan interrupt rutinleri yazacaksınız. Bu rutinler VIC'de öyle kritik değişiklikler yaratacak ki, muhteşem efektler çıkacak ortaya.

Bunu başarabilmek için VIC ve onun belleği kullanımı ile ilgili daha çok şey öğrenmeniz gerekiyor. Bu kurs size, bellek ve registerlera istediğiniz şekilde yazıp çizmenizi sağlayacak, istediğiniz akış ve mantık sırasıyla programlarınızı tasarlamanıza izin verecek olan 6510 programcılığını öğretti. Bundan sonraki basamak olan "VIC Programlama" kursunda, VIC'i sizlere daha iyi tanıtıp bu bölümde öğrendiğiniz interruptlarla ilgili teorik bilgilerin pratikte nasıl kullanıldığını anlatacağım.

Ek A. ACME ve Etiketler

6510 Kursunun okuyuculardan aldığı tepkiler doğrultusunda assembler aracının çalışma detayları ve etiketler (label) konusunun daha detaylı anlatılması gerektiğine karar verdim. Aslında gelişmiş assembler araçlarının kullanılması, zaman zaman programcının bellek diziliminden kopmasına sebep olabiliyor. Eski kartuş monitörlerinde program yazan insanlar kadar bellekteki baytların dizilimi konusuna hakim olmak zorlaşıyor.

Öncelikle yazdığınız programların bellekte ard arda dizilmiş baytlardan ibaret olduğunu hatırlayalım.

```
!to "out.prg"

* = $c000
lda #$00
sta $d020
jmp $c005
```

programını ele alalım. Bu program assembler aracılığı ile bir dizi bayta çevrilir. Bu baytlar çıkış dosyasına yazılır. Yani out.prg dosyasını bir hex editör ile açarsanız içinde şu baytları göreceksiniz.

```
00 c0 a9 00 8d 20 d0 4c 05 c0
```

Şimdi yavaş yavaş bu baytları yorumlayacağız.

C64 dosya formatı

C64'te dosyalar çok basit bir formatta saklanırlar. Her dosya c64 tarafından bellekte yalnızca ardışık baytlardan oluşan bir bölgeye yüklenir. Bu yüzden daha modern sistemlerde rastladığımız çok parçalı (segment'li) dosyalarda olduğu gibi kompleks header bölgeleri kullanılmaz. Bir C64 dosyasında sadece ilk iki bayt dosyanın bellekte yüklenmeye başlanacağı adresi gösterir. Ardından da yüklenecek baytlar sıralanır.

Dolayısıyla out.prg dosyasını bir diskete yazıp c64'e yükletirsek şu olur. C64 önce ilk iki baytı okur ve dosyada geri kalan baytları nereden itibaren yükleyeceğini anlar. Ardından 3. baytı alıp ilk adrese koyar. 4. baytı alıp bir sonraki adrese koyar vs.

Yani c64 out.prg'nin ilk iki baytından dosyanın c000'a yükleneceğini anlar ardından c000 ile c007 adresleri arasındaki 8 bellek adresine yukarıdaki a9 dan başlayan 8 baytı yerleştirir.

Vice emülatörüne

```
x64 out.prg
```

komutunu verdiğinizde de aynen bu olay olur.

İşte ACME assembler aracı da verdiğiniz kod dosyalarını işleyip en son olarak c64 dosya formatında bir dosya oluşturur. Bu dosyayı oluşturabilmek için dosyanın yükleme adresini (yani dosyanın ilk iki baytını) bilmesi gerekir. İşte bu yüzden * komutu kaynak kodlarınızda geçip, bir adrese eşlenmek zorundadır. Eğer kodunuzun içinde hiçbir yerde * = $c000 gibi bir eşitleme olmazsa o zaman ACME oluşturacağı dosyanın bellekte nereye yüklenmesi gerektiğini bilemeyeceği için o dosyadaki ilk iki baytı doğru koyamaz ve hata verir.

* komutu dosyada birkaç yerde geçebilir. Bu durumda olan biteni anlayabilmeniz için ACME'nin nasıl çalıştığını daha detaylı bilmeniz gerekiyor.

Assembly işleminin detayları

Aslında ACME programını çalıştırdığınızda ACME 64 KB'lık yani 6510'un tüm belleği kadar bir alanı PC'nizin belleğinde ayırır. Ardından bu 64KB yani 65536 baytın tamamını 0 ile doldurur. Bu baytlar grubuna şimdiden sonra Tampon adını vereceğim.

ACME programı, kaynak kodları okudukça, karşısına çıkan komut ve data satırlarını baytlara çevirip bunları tampona yerleştirmeye başlar.

Bu işlemi yaparken elde ettiği baytları tamponun hangi adresine yazacağını takip etmek kullanılan 16-bitlik bir işaretçi vardır. Buna çıkış işaretçisi (output pointer) denir.

ACME, işlediği kaynak kodları içinde diyelim ki lda #$00 satırını gördü. Bu satırı a9 00 baytlarına çevirir. Şimdi bu iki baytı tamponda nereye yazacağına bakmak için çıkış işaretçisine bakar. Diyelim ki şu an çıkış işaretçisinde $c010 yazıyor. Bunun üzerine ACME gidip ilk bayt olan a9'u tampondaki c010'uncu bayta yazar. Ardından 00 değerini de tamponda c011'inci bayta yazar. Bu iki baytı yazdıktan sonra da çıkış işaretçisini iki artırır. Böylece bir sonraki komutu işlediği zaman elde ettiği baytları da tamponda c012'nci adresten itibaren yazabilir.

Bu işlem yalnızca kod satırlarında değil, !by ve !scr komutlarının geçtiği satırlarda da aynen tekrar edilir. Örneğin çıkış işaretçisi $2400 değerini taşıyorken ACME şu satırlarla karşılaşırsa

```
!by 0,0,0,0
!scr "abc"
```

önce !by komutunu işleyip tampondaki $2400'üncü adresten itibaren 4 adrese 0 değerini yazar. Ardından çıkış işaretçisini 4 artırır (by ile 4 bayt yerleştirdiği için). Sonra !scr komutunda gecen harfleri karakter kodlarına çevirir ve elde ettiği 3 baytı tamponda çıkış işaretçisinin gösterdiği adres olan $2404 adresinden itibaren 3 adrese yerleştirir.

Bu örneklerde, çıkış işaretçisinde bir değer varken, ACME'nin, kaynak kodlarınızdaki satırları teker teker nasıl baytlara çevirip, tampon içinde çıkış işaretçisinin gösterdiği adrese yazdığını gördünüz. Fakat çıkış işaretçisinin içine ne yazıldığı konusunu anlatmadık. ACME ilk çalışmaya başladığında çıkış işaretçisinin değeri tanımsızdır. İşte * komutu çıkış işaretçisinin içine değer yazılmasını sağlar. Yani ACME kaynak kod satırlarını teker teker tararken, karşısına * = $xxxx komutu geldiği zaman çıkış işaretçisini $xxxx değeri ile yükler. Böylece o satırın ardından gelen komutlar, baytlara çevrildikçe tampon içinde $xxxx'ten başlayan adreslere yerleşmeye başlarlar. Eğer yeni bir * komut satırı gelirse ACME çıkış işaretçisinin değerini yeni verilen komuta göre değiştirir ve tampona baytlar yeni bir adresten itibaren yerleşmeye başlar.

Bütün kaynak kod satırları ACME tarafından tarandıktan sonra çıkış dosyasının yaratılma zamanı gelmiştir. ACME bu noktada tampon içine yaptığı yazma işlemleri esnasında eriştiği en küçük adres (MIN_ADRES diyelim) ile en büyük adres (MAX_ADRES diyelim) arasında kalan bütün baytları kapsayan bir dosya yaratır. Yükleme adresi olarak MIN_ADRES'i oluşturan iki baytı dosyanın başına yerleştirir. Ardından MAX_ADRES - MİN_ADRES tane baytı tampondan alıp dosyanın içine sıralar. Böylelikle MAX_ADRES - MIN_ADRES + 2 boyutunda bir dosya oluşur.

Mesela şu koda bakalım

```
!to "out.prg"
*=$1000
!by $10,$20,$30,$40

*=$2000
lda #$00
sta $d020
jmp $2005
```

out.prg dosyasının ilk iki baytı 00 10 olur (dosyanın yükleme adresi $1000). Ardından 10 20 30 40 baytları gelir. Onu takiben tam 4092 tane 0 baytı yer alır. Bunlar $1004 ile $2000 arasında tamponda başta 0'lanmış ve daha sonra da üzerine bir şey yazılmamış baytlardır. Ardından kodu oluşturan 8 bayt yer alır. Toplam 5006 baytlık bir dosya oluşacaktır.

Etiketler

Bir programcı isterse hiç etiket kullanmadan program yazabilir. Zaten eskiden gelişmiş assembler araçları yokken böyle kod yazılırdı. Kartuşlar kullanılarak kod yazarken bütün komutların argümanı olan adres ve bayt değerleri rakamsal ifadelerle yazılırdı.

```
lda $c0f8,x
clc
adc $c1fa,y
sta $d018
inx
cpx #$20
bne $c020
```

Böyle kod yazdığınız zaman hangi adreste hangi bilgiyi sakladığınızı hangi değerin hangi anlama geldiğini dikkatlice bir deftere not alırdınız. Böylece aradan birkaç hafta geçtikten sonra tekrar koda bakınca "yahu ben c0f8'de neyi tutuyordum? Niye c1fa'daki değerlerle topluyorum d018 neydi?" gibi soruları cevaplayabilirsiniz. Bu hayli zor bir olaydır.

Bu problemleri biraz olsun aşmak için assembler araçları geliştikçe etiket kavramı kullanılmaya başlandı.

Etiket herhangi bir 8 veya 16 bitlik SABİT sayıyı tanımlamak için kullandığınız bir isimdir. Aslında 6510 sizin programınızı çalıştırırken etiketlerden habersizdir. Etiketler yalnızca ACME sizin kaynak kodlarınızı işlerken, ACME tarafından kullanılırlar. ACME tampona baytları dizerken etiketleri rakamsal değerlerine çözümler ve etiketlerin görevi orada biter. Out.prg dosyasına etiketlerle ilgili bir bilgi taşınmaz.

ACME çalışması esnasında bir etiket listesi tutar. Kodu tararken bu listeye yeni etiketler ekleyebilir. Bir komutta argüman olarak bir etiket kullanıldığını görürse bu etiketin sayısal değerini çözümlemek için etiket listesine bakar. Bu listeden okuduğu sayısal değeri kullanarak komutun baytlarını tampona yazar.

Etiketlerden bahsederken iki kavramdan bahsetmeliyiz:

- etiketlerin tanımlanması
- etiketlerin kullanılması

Etiketlerin tanımlanması Etiket ile ifade ettiği Sayısal sabitin eşlenmesi anıdır.

```
SCREENCOLOR = $d021
```

Satırı bir etiket tanımlamasıdır. ACME bu satırı gördüğünde tampona bayt yerleştirme işlemi yapmaz. Onun yerine daha önce belirttiğimiz gibi tuttuğu etiket listesine yeni bir etiketi ve onun eşlendiği sayısal değeri ekler.

Aynı şekilde

```
Printline:
Ldx #00
```

Şeklinde de etiket tanımlanabilir. Böyle kod içindeki bir komutu işaretleyen etiketler kullanıldığında ACME şöyle davranır. Etiketi gördüğü anda çıkış işaretçisinin değeri etiketin ifade ettiği sayısal değer olarak ACME tarafından etiket listesine eklenir.

Bütün etiketler ister birinci şekilde ister ikinci şekilde tanımlansın aynı etiket listesine eklenirler. Etiket tanımlamak ACME'nin tampona yazdığı baytları etkilemez.

Etiketler ancak bir komut satırında argüman olarak geçtiklerinde tampona yazılan baytları etkilerler. Bu etki daha önce belirttiğimiz gibi ACME o etiketin değerini etiket listesinden okuduğu için olmaktadır.

Etiketlerle ilgili kafa karıştıran bir nokta etiketlerin 8 bit veya 16 bitlik büyüklüğe sahip olabilmesidir.

Etiket sonuçta bir sayıdır. Kimi sayılar 256'dan büyüktür. Kimi sayılar 256dan küçüktür. 256dan küçük olan etiketler tek baytlık bir argüman kabul eden bir komutta direk olarak kullanılabilir.

256dan büyük etiketler ise 16 bit oldukları için bazı 6510 assembly komutlarında direk kullanılamayabilir. Burada 6510 komutunun aldığı argümanın 8 bitlik veya 16 bitlik olması belirleyicidir.

Kimi assembly komutları 16 bitlik argümanlar alabilir. Örneğin

```
lda $1000
sta $d020
jsr $3400
jmp $c000
```

gibi argüman olarak 16 bitlik adresler alan komutlar 16 bitlik etiketleri doğrudan kullanabileceğiniz yerlerdir.

Immediate adresleme modu kullanan komutlarda ise 16 bitlik etiketleri doğrudan kullanamazsınız.

```
ETIKET1 =$c000

lda #ETIKET1
adc #ETIKET1
```

gibi komutlar ACME'nin hata vermesine sebep olur. Çünkü bu komutlar 8 bitlik argüman kabul eden komutlardır. ACME ETIKET1 değerini listeden okuduğu zaman karşısında 16 bitlik $c000 değerini görünce lda # komutunun argümanı olarak hangi baytı yazacağını bilemez ve hata verir.

Bu yüzden 16 bitlik etiketlerle çalışırken bu etiketlerin küçük ve büyük baytlarına ulaşmak istersek < > işaretlerini kullanabiliriz. ACME

```
ETIKET1=$c000

ldx #<ETIKET1
ldy #>ETIKET1
```

komutları bu yüzden başarı ile çalışırlar. Ldx# komutunun kabul ettiği 8 bitlik değer ACME tarafından şöyle bulunur. Etiket listesinden ETIKET1 bulunur. Bunun değerinin $c000 olduğu okunur. Sonra < işaretinden dolayı bunun küçük baytı olan 00 alınır satır ldx #$00 olarak yorumlanır ve tampona baytlar yazılır. Ardından gelen satır da aynı şekilde ldy #$c0 olarak yorumlanır.

Önemli Hatırlatma

Etiketlerin, hatırlaması zor olabilen adres ve dataların yerini tutsun diye kullanılan basit isimlendirmeler olduğunu unutmayın. Onlar komutlar gibi özel anlamları olan kelimeler değildir. Onlar sadece $3f8a gibi bir değeri 3 gün sonra unutmayın diye EKRAN_TEMiZLE gibi bir isimle yazdığınız altprogramının adresi ile eşleştirmenizi sağlayan bir kolaylıktan ibarettir.

Bir diğer not, etiketlerin sabit sayıları tanımlamak için kullanıldığını söyledik. Kimi assembler araçları etiketlerin birkaç yerde farklı değerlerle yeniden tanımlanmasına izin verir. Fakat bu şekilde etiket kullanmanızı asla tavsiye etmem. Her etiketi yalnızca bir sayısal değer ile kullanın yoksa etiketin kullanıldığı komut satırlarında hangi değeri alacağını tahmin etmeniz zorlaşabilir. Kodu bir gün başka bir assemblerda assemble etmeye çalışırsanız problem yasayabilirsiniz. Bu sebeplerden ötürü bir etiketi yalnızca bir değere tanımlayarak kullanın.