5. ASSEMBLY DİLİNDE PROGRAMLAMA

Bu bölümde 8051 Assembly dilinde program yazabilmek için gerekli bilgiler yer almaktadır. Bu dilde yer alan temel komut türleri ayrıntılı olarak anlatılmış, basit programlama örnekleri verilmiştir.

5.1. ASSEMBLER'A GİRİŞ

Assembly "mnemonic" olarak adlandırılan komut yapılarından oluşan makine kodundan sonra en alt seviyelide yer alan bir programlama dilidir.

Programlama Dillerine Giriş

Programlar dört farklı dilde yazılabilir. Bunlar sırasıyla, makine kodu, assembly, orta seviyeli diller ve yüksek seviyeli diller.

Makine kodu sadece hex formatta hazırlanmış sembollerden oluşur. Bu nedenle yazımı ve tekrar kontrol işlemi çok zordur. Ancak direk çalıştırılabilirler.

Assembly "mnemonic" olarak adlandırılan komut yapılarından oluşur. Bu komutlar akılda kalıcı olduğundan makine diline göre yazımı çok daha kolaydır. Ancak yazılan kodların derleme işlemi ardından makine dilleri karşılığı direk makine diliyle yazılmış karşılığına çok yakındır.

Orta seviyeli dillere C dili örnek verilebilir. Burada mnemonicler assembly mnemoniclerine göre daha anlamlıdır. Bu nedenle kod yazımı Assembly e göre daha kolaydır. Algoritmaların uygulanabilirliği daha fazladır ancak bu algoritmaların yapıları önceden belirlenmiş olduğundan uygulamada çok fazla esneklik sağlanamaz.

Yüksek seviyeli dillerde kullanım kolaylığı temel alınmıştır. Bu nedenle arayüz kullanılarak kullanım kolaylığı sağlanmıştır. Ancak bu dillerle oluşturulan programlar genelde boyut olarak daha büyük olurlar.

5.1.1. ASSEMBLY NEDİR, DİĞER PROGRAMLAMA DİLLERİ İLE BENZERLİKLERİ VE FARKLILIKLARI

Assembly makine kodundan sonra en alt seviyeli programlama dilidir. Makine diline seviye olarak çok yakın olmasına karşın yazım kolaylığı makine diline göre daha fazladır. Makine dilinde komutlar sadece sayılardan oluşur ve bu hatırlanması zor, zaman harcayan bir iştir ancak assembly de komutlar akılda kalacak şekilde ve kısadır. Assembly dilinin diğer orta ve yüksek seviyeli diller gibi komut sayısı çok değildir ve bu komut yapıları belirli hazırlanmış formatlarda değildir. Bu nedenle assembly dilinin, diğer programlama dillerine göre yapısı farklıdır.

Assembly dilinin diğer programlama dillerine göre en önemli fark ve avantajlarından birisi kullanılan komutlardaki esnekliktir. Yani kullanıcı yazdığı komut ve algoritmalara daha hakimdir ve bunu kendi isteği doğrultusunda düzenleyebilir. Diğer üst seviyeli programlama dillerinde ise komutların belirli bir yapısı vardır ve bu çoğu zaman yazım kolaylığı sağlasa da yazım esnekliği azalmaktadır. Örneğin orta seviyeli bir dil olan C ile karşılaştırılırsa, assembly daha az komut sayısına ancak daha geniş komut kullanım yapısına sahiptir. Yüksek seviyeli diller ile karşılaştırıldığında ise visual tabanlı bir programın assembly dilinde yazımı daha fazla zaman alır, ancak assembly dilinde yazılan kodun sadece gerekli olduğunda library leri kullandığı ve daha az makine çevrimi sürdüğü düşünülürse assemly dilinin önemi anlaşılır. Yüksek seviyeli dillerin yaygın kullanımına karşın, halen hızın ve verimin önemli olduğu noktalarda yazılan çoğu programda assembly dili kullanılmaktadır.

Assembler; yazılan komutların birleştirilmesi, düzenlenmesi ve makine diline çevrilmesi işlemini yapan çevrim programlarıdır. Assembler sayesinde makine dili bilmeden makine kodları kolaylıkla oluşturulabilmektedir. Assembler program içinde yazım kolaylığı açısından kullanılan tanımlama ve değişkenleri orijinal değerleriyle değiştirir bu sayede kodun asıl yapısı bozulmaz. Assembler lar değişik tiplerde olabilir. Bunlar:

Self Assembler : Bu assemblerda oluşturulan kod aynı sistem üzerinde çalışır. Bu nedenle koddan farklı bir makine koduna dönüşüm işlemi için ek dönüştürme işlemi yapılmaz.

Cross-Assembler : Bu Assembler da, oluşturulan makine kodu farklı bir sistemde çalışacağından program kodu bu çalıştırılacak sistemin makine koduna dönüştürülür.

Makro-Assembler : Makro assembler lar normal Assembler lardan daha gelişmiş bir yapı gösteririler. Bunlar sık kullanılan komutları başka programlardan çağırma ve yüksek seviyeli dillerdeki gibi kontrol işlemleri gibi ek özellikler sunarlar. Normal Assembler a göre daha fazla kolaylık sağlamaktadır.

Assembler lar genel olarak önce kaynak kod (source code) dosyalarından (*.asm *.src vb) hareketle list (*.lst) dosyalarını oluştururlar. List dosyalarında komutların ve açıklamaların yanı sıra opcode'lar, yerleştirildikleri adresler ve eğer varsa hata olan satırlar ile hata türlerine ait açıklamalar bulunur. List dosyalarının oluşturulmasının ardından bu dosyalardan object (*.obj) dosyaları oluşturulur. Object dosyaları bağlanarak (linking) tek bir makine kodu dosyası oluşturulur. Bu bağlama işleminde tanımlanan ve çağrılan alt programlara bakılarak alt programlar uygun olan yerlere yerleştirilirve makine kodu oluşturulur.

Program çalışma aşamasına gelmeden önce birçok değişik aşamadan geçer. Bu aşamalar sırasıyla ; dizayn, kodlama, çevrim, test, debug olarak ifade edilebilir.

Bu aşamaları açıklamak gerekirse dizaynda istenilen amaca uygun algoritma yapısı oluşturulur. Kodlamada ise kullanılan mikroişlemci sisteme uygun komutlarla herhangi bir text editör yada assemblerin kendi text editörü kullanılarak kodlama işlemi yapılır. Çevrim işleminde yazılan kodlar önce list dosyalarına, ardından object dosyalarına dönüştürülür. Debug işleminde ise oluşan hatalar ve hata kaynakları ayıklanmaya çalışılır. Debug işleminin ardından yürütülebilir makine kodu oluşturulur ve istenilen işlem gerçekleştirilir.

5.1.2. ASM51 ASSEMBLER GENEL YAPISI

8051 mikrodenetleyici ailesi için kullanılan ASM51, kullanım kolaylığı ve modüler programlamayı sağlayabilmesi açısından hem amatör hem de profesyonel uygulamalar için yeterlidir. Assembler yada herhangi bir text editörde yazılan kod ASM51 de derlendikten sonra bazı işlemlerin ardından mikrodenetleyici sistemin çalıştırılabileceği hex dosyasına dönüştürülür. Burada dikkat edilmesi gereken nokta text editörde yazıldıktan sonra dosya uzantısı olarak asm (*.asm) şeklinde kaydedilmelidir.

Genelde yazılan kod sadece bir asm dosyasında tutulur. Ancak istenildiğinde daha geniş ve ileriye dönük uygulamalar için alt programlar başka asm dosyalarında olabilir. Bu şekilde yazılan alt programlar daha sonra başka uygulamalarda tekrar kolaylıkla kullanılabilmektedir. Bu işlem için ana programda bazı direktiflerle bu alt programlar kullanılmaktadır. Kod yazma işleminin ardından derleyici ana ve alt programları derleyerek kod yazım hataları olup olmadığını kontrol ettikten sonra list (*.lst) dosyalarını oluşturur. Ardından bu list dosyaları birleştirilerek ana program object (*.obj) dosyası oluşturulur. Bu işlemin ardından assembler bu object dosyasından 8051 komutlarının opcode (operation code) larını oluşturur. Opcode mikrodenetleyici sistemin yazılımla, komut işlemlerinin donanım tarafından yürütülebilir hale getirilmiş halidir. Yani opcode lar yazılım kodunun makine kodu (machine code) karşılığıdır.

5.1.3. MAKİNE KODU, OPCODE, OPERAND, PROGRAM COUNTER

Makine kodu, yazılan kodun mikroişlemci sistemlerde temel işlemleri yapan ALU (Arithmetic Logic Unit), diğer donanım kontrol ve işlem birimleri tarafından verilen komutu aritmetik ve lojik komutlarla fiziksel olarak gerçekleştirilmeye uygun halidir.

Opcode ise yazılımdaki kodun makine kodu karşılığıdır.

```
E582    ; 82H adresindeki değeri A register'ına yazmaya yarayan komut
```

Operand koddan sonra o komutun işleminin gerçekleşmesi için gerekli diğer değişkenlerdir.

```
ADD A,Operand     ;Operand içerisinde yer alan değerile a register'ı
                  ;içindeki değer  toplanıp A'nin içerisine yazılır.
```

Mikroişlemcili sistemlerde kullanılan komutlar uzunluk ve makine çevrimi açısından farklılık gösterirler. Bunun nedeni bazı komutlar sadece bir operandla bir işlemi gerçekleştirirken, bazı komutlar iki oparandla birden çok makine çevrimi kullanarak istenen işlemi gerçekleştirler. Mikroişlemci sistemler hangi komutun ne kadar uzunlukta olduğunu ve buna göre bir sonraki işleme geçmeyi Program Counter yardımıyla yaparlar. Program Counter sayesinde mikroişlemci bir komuttan sonra gelen bilginin operand mı yoksa başka bir komut mu olduğunu anlayıp buna göre işlem yapar. Bu sayede komutlar farklı uzunlukta olsalar bile sorunsuz bir şekilde çalıştırılabilirler.

Instruction Register makine kodunun işlenmeye başlanmadan önce tutulduğu yerdir. Yani ALU ve diğer donanım kontrol birimlerinde yürütülmeden önce opcodeların tutulduğu yerdir. Instruction Registerdeki opcode lar daha sonra çözümlenerek (decode) istenilen işlem gerçekleştirilir.

Mnemonic kodun kullanımına uygun gösteriliş formatıdır.

5.1.4. PROGRAM KODU

Assembly dili ile belirli bir sonuca ulaşmak için belirli bir formatta bir araya gelen komutlar program kodumuzu oluşturur.

Başlık

Bu kısım programın en üst kısmında bulunur. Burada kısaca programın hangi amaç için yazıldığı en son değişim tarihi gibi sonraki inceleme ve kullanımlar için kolaylık sağlayacak bilgiler bulunur. Her satırına noktalı virgülle başlandığı için derleme işleminde makine kodunun oluşturulmasında dikkate alınmaz

Tanımlamalar

Kodun ilk kısmıdır, ancak ilk başta oluşturulmaz. Burada program yazılırken oluşturulan sabit tanımlamaları, bellek adres tanımlamaları ve programın başlangıçı için gerekli tanımlamalar vardır. Program yazılırken zamanla oluşan değişkenler, sabitler burada yapılan tanımlamalarla programın daha kolay ve okunabilir olması sağlanır.

Kesme servis programları

Burası mikrodenetleyici sistemin kullandığı kesme servis programlarının, üretici firma tarafından belirlenmiş adreslere gerekli alt programların yerleştirildiği kısımdır. Bu adresler önceden belirli olduğundan buradaki belirlenen adreslere ORG komutuyla gerekli komutlar yerleştirilir.

Ana Program

Ana program; yazılımın genel kısmıdır. Bu kısım diğer program blokları ve yapıların kullanıldığı, genel kontrol ve diğer işlemlerin yapıldığı kısımdır. Program içerisinde yazım kolaylığı için kullanılan semboller, modüler programlamada kullanılan direktifler ve esas kontrol komutları ana programda bulunur.

Ana program mikrodenetleyici sistemin ilk şartlandırmalarının yapıldığı çalışmaya hazır hale getirildiği konfigürasyon kısmıdır. Burada mikrodenetleyici sistemin donanım ve yazılım olarak ilk şartlandırmaları yapılır. Mikrodenetleyici sistemin giriş çıkış birimleriyle dış sistemlerle etkileşimi arttıkça bu konfigürasyon kısmı da artmaya başlar.

Alt programlar

Alt program ana program tarafından her zaman kullanılmayan sadece gerektiğinde çağrılan komut bloklarıdır. Okunabilirlik açısından alt programlara, alt programın işleviyle ilgili bir etiket verilmelidir. Ayrıca alt programlar için aynı etiketlerinin kullanılmamasına dikkat edilmelidir, aksi takdirde derleme işlemi sırasında assembler hata mesajı verecektir.
Alt programların sonuna mutlaka RET komutu konulmalıdır, aksi takdirde program yanlış komutları yürütecektir. Alt programlar ana programın sonunda bulunmalıdır. Alt programlardan sonra ana programın bazı kısımları bulunmamalıdır.

Tablolar

Aritmetik ve dönüşüm işlemleri için kullanılan tabloların oluşturduğu kısımdır. Bu tablolar bir etiketle başlar ve DB veya DW komutları yardımıyla oluşturulur.

Haluk Turksoy Sk, No:11 ALTUNIZADE, 34662 ISTANBUL-TURKEY
TEL: +90 216 6518000 FAX: +90 216 6518009
http://www.elektro.com.tr info@elektro.com.tr

5.2. DİREKTİFLER

Direktifler Assembler tarafından doğrudan makine koduna eklenmeyen ancak diğer komut, alt program ve kontrol ifadelerinin işlenmesini sağlayan komutlardır. Bu komutlar sayesinde assembler makine koduna belirli eklemeler ve değişiklikler yaparak programın daha etkili ve kolay bir şekilde oluşturulmasını sağlar.

EQU

EQU komutuyla belirli değişken ismi sabit olarak atanabilir. Derleyici EQU komutuyla tanımlanan bütün değişkenlere derleme işlemi sırasında tanımlanan sabit değerini yerleştirir. Böylece değişkenler makine koduna sabit değerler olarak yerleştirilir. EQU tanımlama komutuyla tanımlama yapılırken aynı değişken ismi yalnızca bir kez kullanılmalıdır. Ayrıca EQU komutuyla komut ve direktifler tanımlama için kullanılmamalıdır.

```
SAYAC  EQU  45H
ARTIM   EQU  32
…
…
MOV A,SAYAC
```

DATA

Data komutuyla belirli bir adres bir değişken olarak atanabilir. Böylece uzun programlarda çok sayıdaki bellek adresleri bu değişken isimler kullanılarak bellek adresleri karışıklığı olmadan kolaylıkla kullanılabilir. Derleyici DATA komutuyla tanımlanan bütün değişkenlere derleme işlemi sırasında tanımlanan bellek adresini yerleştirir. Böylece değişkenler makine koduna bellek adresleriyle işlem yapılmış gibi yansır. Bu komut orta ve yüksek seviyeli dillerdeki değişkenlerin karşılığı olarak düşünülebilir. Dikkat edilmesi gereken nokta DATA tanımlama komutuyla tanımlama yapılırken aynı değişken ismi yalnızca bir kez kullanılmalıdır. Ayrıca DATA komutuyla komut ve direktifler tanımlama için kullanılmamalıdır.

```
SON_DEGER  DATA  45H
```

CODE

Bu komutla belirli bir değişken ismi kod belleğindeki bir adres olarak tanımlanabilir. Böylece kod belleğindeki bir adres belirli bir değişken olarak kullanılabilir. Yine diğer değişken tanımlamalarına olduğu gibi bu değişkenlerde derleme işlemi sırasında adres kod belleği adres değerleri yazılarak derleyici tarafından değiştirilirler.

```
RESET  CODE  00H
TIMERO_KESMESI  CODE  0BH
```

SET

EQU komutu gibi bu direktif de bit tanımlamaları için kullanılır.

```
FLAG  BIT  0
```

Sembol Tanımlamaları

Herhangi bir yerde tanım yada etiket olarak kullanılacak kelimeler en fazla 255 karakter olabilir ve bu tanımlamanın sadece ilk 31 karakteri dikkate alınır.

Örneğin:

```
31_karakterden_uzun_tanımlama_isimleri
```

ile

```
31_karakterden_uzun_tanımlama_i
```

aynıdır.

Sembollerde büyük yada küçük harf duyarlılığı yoktur. Yani Assembler için "TABLO" ile "tablo" aynıdır.

Etiketler

Etiketler çağırma ve sıçrama komutlarında kullanılır. Etiketlerden sonra iki nokta işareti kullanılmalıdır. Etiket isimleri sayesinde çağırma yada sıçrama komutlarında alt programların başlangıç adresleri bilinmek zorunda değildir. Derleme işlemi sırasında derleyici etiketten sonra gelen komutun program belleği adresini etiketin isminin geçtiği bütün komut satırlarına yerleştirir. Böylece kullanıcı komutların uzunluklarını çok gerekmedikçe hesaplamakla uğraşmayacaktır. Etiketler kullanılarak etiketten sonra gelen komutların istenen yerde yürütülmesi sağlanır.

```
ALT_PROGRAM:
      MOV R1,#45H
      MOV A,@R1
      DEC DPTR
      …
      RET
```

Not: Alt programların sonunda RET komutu unutulmamalıdır. Yoksa komut yürütme işlemi aşağı doğru devam eder ve yanlış sonuçlar elde edilebilir.

ORG

Bu direktif sayesinde bu komutun ardından gelen kod belirtilen adresten sonra kod belleğine yazılır. Bu komut sayesinde kesme servis programları donanım üreticileri tarafından belirlenen adreslere kesme servis alt programlarının yerleştirilmesini sağlar.

```
ORG  001BH        ; Timer1 kesme adresi
INC 32H           ; 32H 'ın içeriğini bir arttır
MOV A,32H         ; 32H akümülatöre yaz
RETI              ; kesmeden geri dön
```

Yukarıdaki örnekte Timer1 kesme servis alt programında küçük bir alt program örneği yazılmıştır.

Not : Kesme servis alt programlarının uzunluğu en fazla 8 byte olmak zorundadır. Eğer daha uzun bir servis programı yapılacaksa, bu program bir alt program olarak oluşturulur ve kesme servis alt programında bu alt program çağrılır.

```
ORG  0100H
MOV R0,#90H
MOV A,@R0
```

Bu örnekte ise belirli bir adresten sonra program kodu yazılarak makine kodunun bu adresten sonra program belleğine yazılması sağlanmıştır.

INCLUDE

Bu direktifle önceden yazılan alt program tekrar yazılmadan bu komutla ana program bloğunun içine yerleştirilir. Böylece sürekli kullanılan alt programlar defalarca yazılmak yerine ana programdan INCLUDE komutuyla çağrılarak program belleğine bu komutun ardından yerleştirilir.

```
#INCLUDE  7_SEGMENT.ASM
```

DB

DB komutu kendisinden sonra gelen değerlerin sadece byte olarak sabitler olduğunu belirtir. Böylece bu komuttan sonra gelen karakter belirlenen etiketten sonra program belleğine byte byte yerleştirilir ve DPTR registeri kullanılarak buradan okuma yapılabilir. Bu komutla ayrıca iki tırnak içersine alınan karakter yada karakter dizileri ASCII olarak kaydedilir. ASCII olarak kaydetme işlemi yine derleyici tarafından derleme işlemi sırasında yapılır ve yazılan karakterlerin ASCII karşılıkları kolaylıkla bulunmuş olur. Bu işlem seri porttan veri gönderirken yada LCD gibi gösterge amaçlı uygulamalarda çok kullanılmaktadır.

```
TEK_SAYILAR:  DB 1,2,3,5,7,11,13,17,19

YAZI:  DB 'LCD EKRANINA YAZ'
```

Ayrıca byte ve ASCII veri birlikte de kullanılabilir

```
STOK:  DB 'TRANSISTORLER',90,'OPAMPLAR',20
```

DW

DW komutu da DB komutu gibi kullanılır, tek farkı burada sabitler byte byte değil de WORD olarak yani 16 bit olarak program belleğinde saklanmaktadır.

```
TABLO:  DW  2004,'G', 1900,45,'F'
```

END

Yazılan bütün programlar END direktifi ile bitirilmelidir. Bunun nedeni mikrodenetleyici kod belleğinde komutları sırayla yürütürken kodun nerede bittiğini algılayamaz ve aşağı doğru kod belleğindeki değerlere göre işlem yapmaya devam eder. Böylece program belleğine yazılan son makine komutundan sonra o adreste bulunan rasgele değer makine koduymuş gibi işlem görür ve bu istenmeyen sonuçlar gözlenmesine neden olabilir. Ancak programın sonunda tekrar programın başına sıçrama komutu veriliyorsa bu komut kullanılmayabilir de çünkü mikrodenetleyici hiçbir zaman belirsiz makine kodlarını yürütemez ve böylece istenmeyen sonuçlar gözlenmemiş olur. Ancak programın sadece bir kez çalışması isteniyorsa ve sonra işlemleri durdurması isteniyorsa programın sonuna END komutu eklenmelidir.

```
ANA_PROGRAM:
      …
      …
      END
```

Açıklamalar

Açıklamalar program tekrar incelendiğinde yada başkası tarafından incelenmesi gerektiğinde programda kullanılan algoritmaları anlatan kısa ifadelerdir. Komutlardan sonra noktalı virgül (;) ile başlayan kısımdır. Genellikle programın uzun uzun açıklamsı yapılmaz, sadece o satırdaki işlem bu işlemde ne yapıldığı anlatılmaya çalışılır. Açıklamalar noktalı virgülle başladığından derleyici bu karakterden sonra karakterleri sadece list (*.lst) dosyasına ekler, object (*.obj) ve hexadecimal (*.hex) dosyasında açıklama kısmı görülmez.

```
      MOV  R0,#08H                ;ilk değer
DONGU:
      MOV  A,@R0                  ;R0 ın gösterdiği adresteki sayıyı
                                  ; akümülatöre yükle
      INC  R0                     ;R0 ı arttır
      …
      …
      CJNE  R0,#0E0H,DONGU        ;R0  E0 oluncaya kadar işleme devam et
      …
```

5.3. 8051 KOMUT SETİ KULLANIM ÖRNEKLERİ

Bu bölümde 8051 Assembly dilinde kullanılan komutlar anlatılmış ve kullanımlarına dair basit programlama örnekleri verilerek açıklanmıştır.

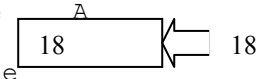
5.3.1. VERİ TRANSFER KOMUTLARI İLE HAFIZA İÇİ VERİ AKTARIMI ÖRNEKLERİ

Hafıza üzerinde her hangi bir adresteki verinin bir yerden başka bir yere aktarılması için veri transfer komutları kullanılır. Veri bu komutlarla farklı adresleme modları ile yer değiştirir.

5.3.1.1. İVEDİ ADRESLEME (IMMEDIATE ADDRESSING MODE)

Hemen adreslemede gönderilecek veri bir sabitten oluşur. Alt programlarda yada genel amaçlı olarak programlarda belirli sabitlere bağlı olarak program akışının devam etmesi için hemen adresleme kullanılır. Genellikle bir byte transfer edilir. Ancak iki byte transferin gerekli olduğu durumlarda DPTR registerıyla iki byte transferi yapmakta mümkündür.

```
MOV A,#18H              ;Akümülatöre 18H değerini yükle
MOV A,#18               ;Akümülatöre 18  değerini  yükle
```



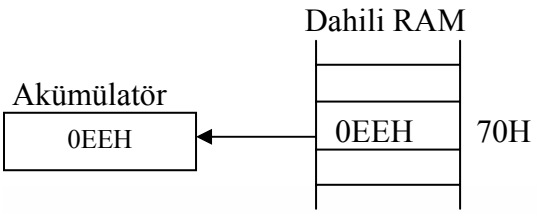

```
MOV R0,#4               ;R0 registerına 4 değerini yükle
MOV PSW,#0              ;PSW registerına 0 değerini yükle
MOV DPTR,#6500H         ;DPTR registerına 6500H değerini yükle
MOV DPTR,#TABLE         ;TABLE etiketindeki bloğun adresini yükle
```

Not : Son örnek alt programlarda tablo oluşturmak, tablodaki bu değerlerin gerektiğinde kullanılması için ve bir veri bloğunun başlangıç adresinin belirlenmesi için kullanılır.

5.3.1.2. DOĞRUDAN ADRESLEME (DİRECT ADDRESSING MODE)

Doğrudan adresleme modunda iç veri veya SFR registerından yine iç veri veya SFR registerına veri transferi yapılır. Portlar da SFR registerında adreslendiği için bu adresleme modunda portlardan da veri okunup yazılabilir.

```
MOV A,70H               ;70H adresindeki veriyi akümülatöre gönder.
```



```
MOV 60H,A               ;Akümülatördeki veriyi 60H adresine yaz.
```

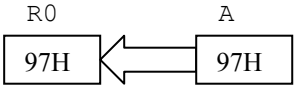
Ayrıca program yazarken kullanım kolaylığı açısından kullanılan Equal tanımı yardımıyla belirli bir adres kolayca adreslenebilir.

```
PORT1 EQU 90H           ;Assembler bu tanımdan sonra PORT1 gördüğü
                        ; yere 90H değerini  yerleştirir.
MOV PORT1,55H           ;55H  adresindeki veri Port1 den gönderilir.
```

5.3.1.3. REGİSTER ADRESLEME MODU (REGISTER ADDRESSING MODE)

Bu adresleme modunda R0,R1,R2,R3,R4,R5,R6,R7 registerları kullanılır. R0-R7 registerlarının adresleri değişkendir. Yani istenirse bu adresler değiştirilerek R0-R7 registerları 4 farklı şekilde konumlandırılabilir. Bunun için öncelikle PSW registerındaki 4. ve 3. bitler ayarlanır. Bu bitlerin varsayılan değeri 00 dır. PSW deki bu iki bit değiştirilerek R registerları 32 register gibide kullanılabilir.

```
MOV R0,A        ;akümülatördeki veriyi R0'a yaz.
MOV PSW,#10H    ;ikinci bank seçilir.
MOV R4,B        ;B akümülatöründeki veriyi R4 registerına yaz.
```



5.3.1.4. ÖZEL REGİSTER ADRESLEME MODU (REGISTER SPECIFIC ADDRESSING MODE)

Bazı komutlar sık kullanıldığı için bu komutlar daha az makine çevrimi veya daha küçük veri boyutu için düzenlenmiştir. Bu şekilde çok fazla kullanılan komutlar daha verimli bir halde gerçekleştirilir ve ana program süresi kısaltılmış olur. Bu adresleme moduna örnek olarak A,R0,R1 registerlarının kullanıldığı bazı komutlar verilebilir.

```
MOV A,#76H              ;A akümülatörüne 76H değerini yaz
                        ;bu işlem için 2 byte
                        ;1 makine çevrimi gerekmektedir.


MOV 0E0H,#76H           ;A akümülatörünün SFR deki adresi 0E0H dir.
                        ;İki komutta aynı işlemi yapmasına karşın
                        ;bu komut 3 byte 2 makine çevrimi sürer.
```

5.3.1.5. REGİSTER DOLAYLI ADRESLEME MODU (REGISTER INDIRECT ADDRESSING MODE)

Önceki bölümlerde anlatıldığı gibi üst veri bölgesi iki parçadan oluşur. Bu iki bölgenin ayrı ayrı kullanılabilmesi bu adresleme moduyla gerçekleştirilir.
Ayrıca bu adresleme moduyla harici veri yada program bloğuna okuma ve yazma işlemleri için erişilebilir.

```
MOV R0,78H              ;R0 a 78H adresindeki değeri yükle

MOV @R0,#4              ;R0'ın gösterdiği yere 4 değerini yaz.

MOV DPTR,#9000H         ;DPTR registerına 9000H değerini yükle

MOVX @DPTR,A            ;akümülatördeki veriyi DPTR registerının
                        ;gösterdiği yere yaz.
```

5.3.1.6. REGISTER İNDEKSLİ ADRESLEME MODU (REGISTER INDEXED ADDRESSING MODE)

Bu adresleme modu en çok tablo kullanımında kullanılır.Bunun için öncelikle tablonun adresi program belleğinde olduğundan 16 bitlik bu tablo adresi DPTR registerına yüklenir. Ardından bu tablo değerlerinden istenilen, akümülatör yardımıyla okunur ve gerekli işlemler yapılır. Aşağıda SAYICI adında 0-9 sayan bir değişkenin içeriğini 7-segment LED display'de görüntüleme alt program örneği verilmiştir. Gerektiğinde bu alt program çağrılarak akümülatördeki sayı 7-segment LED display'e gönderilir.

```
      MOV   A,SAYICI

      MOV   DPTR,# DISPLAY_TABLOSU      ;look up table'ın başlangıç adresi
                                        ;DPTR'ye atanır

      MOVC  A,@A+DPTR                   ;A ya tablonun istenen değeri
                                        ;atanır.

DISPLAY_TABLOSU:                        ;Hanede görünecek olanı seçen
                                        ;tablodur

      DB    00000000B                   ;0 Görünür
      DB    00000001B                   ;1 Görünür
      DB    00000010B                   ;2 Görünür
      DB    00000011B                   ;3 Görünür
      DB    00000100B                   ;4 Görünür
      DB    00000101B                   ;5 Görünür
      DB    00000110B                   ;6 Görünür
      DB    00000111B                   ;7 Görünür
      DB    00001000B                   ;8 Görünür
      DB    00001001B                   ;9 Görünür
```

Bu değerler sırasıyla 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F değerlerinin 7-segment LED display için düzenlenmiş halidir.

Buradaki örnekte tablonun başlangıç adresi

```
MOV   DPTR,# DISPLAY_TABLOSU
```

komutu ile alınmıştır. Bu komut sayesinde tablonun program belleğindeki belirli bir adresten değil de programın ilerleyen kısmında yüklenerek daha verimli kullanılması sağlanmıştır. Bu kullanımın dışında program

```
MOV  DPTR,#8100H
```

komutu ile direk program belleğinden yüklenebilir.

5.3.1.7. YIĞIN UYUMLU ADRESLEME MODU (STACK ORIENTED ADDRESSING MODE)

Bu adresleme modunda temel olarak PUSH ve POP yığın komutları kullanılır. Bu adresleme modunda dikkat edilmesi gereken nokta yığın sisteminin çalışma mantığıdır. 8051 ailesi mikrodenetleyicilerinde first in- last out sistemi kullanılır. Yani yığına ilk gönderilen veri en son çıkartılır. Bu adresleme moduna örnek aşağıdadır.

```
MOV A,12H
PUSH ACC
….
….
POP B
```

Not : Burada akümülatörün ACC şeklinde gösterimine dikkat edilmelidir.
PUSH A komutu derleyici tarafından doğru bir komut olarak algılanmaz çünkü burada özel register adresleme modu kullanılmıştır.

5.3.1.8. YER DEĞİŞTİRME (EXCHANGE) KOMUTLARI

Yer değiştirme komutu herhangi bir adresle akümülatör arasında veri değişimi şeklinde olur. Burada değişim işlemi direkt yapılmaktadır, yani akümülatörün başka bir yere geçici olarak kopyalanmasına gerek yoktur. Burada dikkat edilmesi gereken nokta kaynak doğrudan yada dolaylı adreslenmiş herhangi bir adres olabilir ancak hedef her zaman akümülatördür. Ayrıca bu komut yapısının içersinde nibble taşımaya özel ek bir komut vardır.Bu komut ile akümülatörün düşük anlamlı nibble (dört bit) ‘ı ile diğer operandın düşük anlamlı nibble’ı yer değiştirir. Yüksek anlamlı nibble’lar bu değişimden etkilenmez.Bu ek komut BCD (Binary Coded Decimal) işlemlerinde kolaylık sağlamaktadır. Komutların kullanımına örnekler :

```
MOV A,#023H
MOV B,#045H
XCH A,B          ;Bu işlemin ardından artık A’da 45H,  B’de 23H
                 ;değeri olur.

MOV R0,7FH       ;Bu işlemin ardından 7FH adresindeki düşük anlamlı
XCHD A,@R0       ; nibble ile akümülatördeki düşük anlamlı
                 ;nibble yer değiştirir.
```

5.3.1.9. BİT TRANSFER KOMUTLARI (BIT ORIENTED DATA TRANSFER)

8051 ailesi mikrodenetleyicilerin güçlü özelliklerinden birisi de bit işlemleri yapabilmesidir. Bit işlemleri kontrol amaçlı uygulamalarda sadece belirli bir pinin yada bayrağın kontrolü ve transferi için programlarda büyük bir esneklik ve kolaylık sağlar . Örneğin bir butonun konumu bu komutlarla rahatlıkla kontrol edilebilir. Ayrıca portlardaki herhangi bir bitin kontrolüyle sisteme bağlanan bir birim kontrol edilebilir. Bit işleme komutlarında unutulmaması gereken bir nokta ise SFR registerlarından sonu 0 veya 8 ile biten adreslerin bit bit kontrol edilebileceğidir.Yani bu adreslerden bit okuma ve yazma işlemleri yapılabilir. Ayrıca 20H-2FH arasındaki 16 byte lık bölgede bit olarak adreslenebilmektedir. Bu bit adresleme yöntemi aşağıda açıklanmıştır.

i : 20H ve 2FH arasındaki adres
j : 0. ve 7 . bitler
Bit adres = 8 x (i – 20H) + j
Buradan 16 byte lık bölge 00H-7FH olarak bit adreslenmiştir.

Bu komutun ve bit adreslenebilir bölgenin kullanımına örnek :

```
MOV C,45H              ;45H adresindeki bit elde bayrağına alınmıştır

JC ALTPROG             ;elde set edilmişse ALTPROG 'a git

JNB 16H,ALTPROG2       ;16Hadresindeki bit set edilmemişse ALTPROG2e
                       ;git
```

5.3.2. VERİ İŞLEME KOMUTLARI İLE HAFIZA İÇİ VERİ AKTARIMI ÖRNEKLERİ

Veri işleme komutları aritmetik ve lojik işlemler olarak iki bölüm halinde incelenebilir.

5.3.2.1. ARİTMETİK İŞLEMLER

Bu bölümde toplama-çıkarma, çarpma-bölme gibi aritmetik işlemlere ait komutlar açıklanacaktır.

5.3.2.1.1. TOPLAMA VE ÇIKARMA İŞLEMLERİ

Toplama İşlemi:

8051 komut setinde iki ayrı toplama işlemi komutu vardır, ADD ve ADDC. Bunlar sırasıyla toplama ve elde ile toplama komutlarıdır. Her iki komutta da iki byte toplanarak sonuç akümülatörde saklanır. ADD komutu; akümülatör ve herhangi bir adresteki sayının toplanarak sonucun akümülatöre saklanmasını sağlar. ADDC ise normal toplama komutuna ek olarak elde bayrağındaki bitin de toplamaya katılmasını sağlar. Her iki toplama komutunda da 3 bayrak sonuçtan etkilenir. Bunlar; elde, yarım elde ve taşma bayraklarıdır. Elde bayrağı çıkan sonuçtan elde oluşuyorsa set edilir. Yarım elde bayrağı ise toplanan sayılardaki düşük nibble lardan taşma oluşuyorsa set edilir. Yarım elde BCD sayılarla yapılan işlemlerde kolaylık sağlamaktadır. Taşma bayrağı ise 6.bitlerin değil de sadece 7. bitlerin toplamından elde oluşuyorsa set edilir, veya 7. bitlerin değil de sadece 6. bitlerin toplamından taşma oluşuyorsa set edilir. Bu taşma bayrağı işaretli sayılarla yapılan işlemlerde toplanan sayıların işareti dikkate alınarak toplama işleminin yapılmasını sağlar.

Bu komutların kullanımına örnekler :

```
MOV A,#03
ADD A,#04     ;bu komutun ardından artık
              ;akümülatörde 07 değeri olur.
MOV 78H,#02
ADD A,78H     ;akümülatörde 09 değeri olur.
```

78H -79H ve 7AH-7BH adreslerindeki ikişer byte lık sayıların toplamı

```
....
MOV 78H,#34H
MOV 79H,#12H
MOV 7AH,#0EFH
MOV 7BH,#12H

MOV A,78H
ADD A,7AH          ;bu işlemin ardından 78H de 23H değeri oluşur
                   ;ve elde
MOV 78H,A          ;bayrağı set edilmiş olur
MOV A,79H
ADDC A,7BH
MOV 79H,A          ;bu işlemin ardından 79H de 25H değeri oluşur
....
```

Çıkarma İşlemi:

Toplama işleminin aksine çıkarma için sadece SUBB komutu mevcuttur. “Subtract with Borrow” kelimelerinin kısa ifadesi olan bu komutta “Carry – Elde” biti, “Borrow - Borç” biti olarak değerlendirilir.

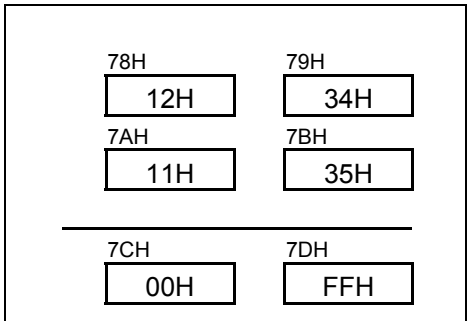
Çıkarma işleminde de toplama işleminde olduğu gibi sonuç yine akümülatörde saklanır. Akümülatördeki sayıdan herhangi bir adresteki sayı çıkarılarak sonuç akümülatöre yazılır. Çıkarma işleminde akümülatördeki değer diğer sayıdan küçükse elde biti set95

edilerek işlem sırasında eldenin ödünç alındığı belirtilir. Bu özellik çoklu çıkarma işlemlerinde kolaylık sağlamaktadır. Eğer eldeli çıkarma yapılması istenmiyorsa, yani birden fazla byte'dan oluşan iki değişken üzerinde çıkarma yapılmıyorsa, elde bitinin önceki değerinin çıkarma sonucunu etkilememesi için elde biti temizlenir (CLR C).

Yarım elde bayrağı da toplamada olduğu gibi BCD sayılarla yapılan işlemlerde kullanılır. Akümülatördeki sayının düşük nibble 'ı çıkan sayıdan küçükse yarım elde bayrağı set edilir. Yine taşma bayrağı da toplamada olduğu gibi işaretli sayılarla yapılan işlemlerde sayıların işaretlerini koruması için kullanılır. Yani taşma bayrağı 6. bitlerin değil de 7. bitlerin yada 7. bitlerin değil de 6.bitlerin çıkarma işleminden dolayı elde ihtiyacı oluşuyorsa set edilir. Çıkarma işlemine örnek aşağıdadır:

Hatırlatma: Çıkarma işleminden önce gerektiğinde elde bayrağı temizlenmelidir.

78H ve 79H adreslerindeki iki byte uzunluğundaki sayıdan 7AH ve 7BH adreslerindeki sayının çıkartılması




```
.....
MOV 78H,#12H
MOV 79H,#34H
MOV 7AH,#11H
MOV 7BH,#35H
CLR C

MOV A,79H          ;Bu işlemin ardından
SUBB A,7BH         ;Akümlatörde #0FFH değeri oluşur
MOV 7DH,A          ;elde bayrağı (Carry flag) set değerini alır.

MOV A,78H
SUBB A,7AH         ;Çıkarma esnasında Carry biti de dikkate alındığın-
MOV 7CH,A          ;dan sonuç 00H olarak değeri olarak elde edilir.
.....
```

5.3.2.1.2. ÇARPMA BÖLME İŞLEMLERİ

Çarpma ve bölme işlemleri mikrodenetleyiciler için fazla makine çevrimi isteyen komutlardır. Bu nedenle çarpma ve bölme işlemleri için ekstra donamım yapıları kullanılarak bu işlemlerin süresi kısaltılmaya çalışılır. 8051 ailesi mikrodenetleyicilerinde bu donanım vardır ve çarpma ve bölme işlemleri her biri 4 makine çevrimi sürer. Bu sürenin azaltılması aritmetik işlemlerin çok kullanıldığı uygulamalarda genel program süresini oldukça kısaltmaktadır. Çarpma ve bölme işlemlerinde adresleme modu özel register adresleme modunda olduğundan sayılar A ve B akümülatörlerine yüklenmelidir.

Çarpma işleminde çarpılacak 8 bitlik iki sayı önce A ve B akümülatörlerine yüklenir. Çarpma komutunu ardından yüksek ağırlıklı byte B akümülatöründe düşük ağırlıklı byte ise A akümülatöründe saklanır. Ancak burada çarpım sonucunun FFFFH değerini geçmemesine

dikkat edilmelidir. Bu nedenle bu işlemde elde bayrağı hiçbir zaman set edilmez. Sonuç eğer FFH değerinden büyükse taşma bayrağı set edilir.

Bölme işleminde ise bölünecek 8 bitlik sayı önce A akümülatörüne yüklenir ardından B akümülatörüne bölen sayı yüklenir. Bölme işleminin ardından tam kısım A akümülatöründe kalan ise B akümülatöründe saklanır. Bölme işleminden sonra elde bayrağı her zaman temizlenir sadece taşma bayrağı 0 ile bölme işlemi oluşursa set edilir. B akümülatörüne eğer 0 ile bölme için 0 yüklenirse akümülatörlerdeki sonuç rasgele değerlerdir. Bu nedenle yazılan programlarda B akümülatöründe bölme işleminden önce 0 değeri olmaması sağlanmalıdır.

Çarpma ve bölme işlemlerine örnek aşağıdadır:

```
MOV A,#99          ;99 x 5 = 495 = 1EFH
MOV B,#5
MUL AB             ;bu işlemin ardından A akümülatöründe #0EFH
                   ;B register'inde ise #01 değeri oluşur.

MOV A,#99          ;99 / 5 = 19 x 5  + 4
MOV B,#5
DIV AB             ;Bu işlemin ardından A akümülatöründe 13H
                   ;ve B akümülatöründe 4 değeri oluşur
```

5.3.2.1.3. ARTTIRMA-EKSİLTME İŞLEMLERİ

Programlarda genel olarak herhangi bir adresteki verinin arttırılması ve azaltılması INC ve DEC komutlarıyla gerçekleştirilir. Bu komutlar sayesinde herhangi bir adrese yada veri bloğuna arttırma ve azaltma komutlarıyla erişilebilir. Bu erişim için bir alt program yazılabilir ve gerektiğinde bu alt programla istenilen adrese erişilebilir. Bunun için R0-R7 registerlarına öncelikle bloğun adresi yazılır ardından dolaylı adreslemeyle bu adresin ilk bloğuna ulaşılır ve arttırma azaltma komutuyla R0-R7 registerları azaltılarak işleme devam edilir. Alt programın sonuna bir kontrol komutu konularak belirli bir değere kadar bu işlemin tekrarlanması sağlanabilir. Aşağıda 70H-77H adreslerindeki sayıların birer arttırılarak 80H-87H adreslerine yerleştiren bir alt program örneği vardır:

```
ALT1:
      MOV R0,70H          ;kaynak bölgesinin ilk adresi yüklenir sonucun
      MOV R1,80H          ;yazılacağı bölgenin ilk adresi yüklenir 77H
DONGU:                    ;değerine kadar işleme devam edilmesi için
      INC @R0             ;R0 ' ın gösterdiği yeri bir arttır
      MOV A,@R0           ;bu değeri akümülatöre yükle
      MOV @R1,A           ;Akümülatördeki değeri R1 'in gösterdiği yere yaz
      INC R0              ;bir sonraki adrese git
      INC R1              ;bir sonraki değere git
      CJNE R0,78H,DONGU ;77H değerine kadar işleme devam et
      RET                 ;alt programdan çık
```

Not: Azaltma komutunda dikkat edilmesi gereken nokta azaltılacak adresteki sayının

00H olmamasıdır. İçeriği 00H olan adresteki sayı azaltma komutuyla adreste FFH değeri oluşur.

5.3.2.1.4. ON TABANINA GÖRE AYARLAMA (DECİMAL AYARLAMA)

On tabanına göre ayarlama işlemi BCD sayılarla toplama işlemi yapıldığında oluşan sonucunda BCD olması için tek komutluk bir ayarlama işlemi yapar. Burada toplama işleminde düşük ağırlıklı nibble lar toplanırken sonuç 9 değerinden taşıyorsa bir üst nibble toplamına sonuç eklenir. Eğer bu üst nibble ların toplamından da taşma oluşuyorsa elde bayrağı set edilir. Bu şekilde BCD iki sayı toplandıktan sonra on tabanına çevirmek için daha fazla işlem yapılmaması sağlanmış olur.

Not1: Onluk ayarlama işleminden önce elde bayrağı set edilmişse bu elde düşük ağırlıklı nibble ların toplamında toplamaya katılır.

Not2: BCD değerler arttırma yada azaltma komutlarıyla işlem yapıldığında 9 değerinden A,B,C,D,E,F ye taşma veya 0 dan azaltma işlemiyle A,B,C,D,E,F ‘e taşma hataları oluşur. Bunun yerine arttırmak için 01, azalmak için 99H ile toplanıp ardından DA A komutuyla tekrar BCD şekline getirilirler.

Akümülatörde ve R0 da BCD olarak 48 ve 63 değerleri olsun. Bu sayılar 58H ve 63H olarak registerlara alındıktan sonra toplama işlemi yapılır. Akümülatördeki değer CBH dir. DA A komutuyla onluk tabana ayarlandıktan sonra akümülatörde 21 değeri oluşur ve elde bayrağı set edilir yani sonuç 121 dir.

5.3.2.2. LOJİK İŞLEMLER

Mikrodenetleyici sistemlerde sıklıkla kullanılan lojik işlemler burada açıklanmaktadır.

5.3.2.2.1. AND, OR, XOR İŞLEMLERİ

Bu üç komutta doğrudan, dolaylı ve diğer register adresleme modlarını kullanır. Ancak bu komutlarda bir işlem elemanı ve akümülatör kullanılır. VE ve VEYA genelde bazı bitleri okumak veya değiştirmek işlemleri için maskeleme işleminde kullanılır. Bu yöntemle bit olarak adreslenemeyen adreslerdeki verilerde bit olarak istenilen değişiklikler yapılabilir. Aşağıda bu işlemlere ait örnekler verilmiştir.

```
MOV A,#18H
ORL PSW,A   ;Bu işlemin ardından PSW registerındaki 3. ve 4. bitler
            ;ilk durumlarına bakılmaksızın set edilmişlerdir.
```

Port1 ‘e bağlanmış 8 led lojik işlemlerle yakılmaktadır.

```
MOV P1,#10101010b ;sırasıyla 2,4,6,8  numaralı ledler yakılmaktadır.
MOV A,#0FH
ANL P1,A          ;1, 2, 3, 4 numaralı ledler yakılmaktadır.
```

5.3.2.2.2. TÜMLEME-TEMİZLEME İŞLEMLERİ (COMPLEMENT,CLEAR)

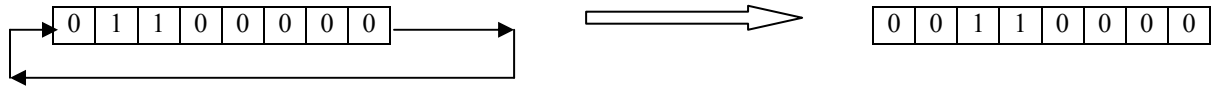
Bu komutlarda özel register adresleme kullanılır. Eşleniği alma seçilen adresteki bütün bitlerin eşleniğini alır, temizleme işlemi ise seçilen adresteki bitler temizlenir yani o adrese 00H değeri yazılır.

```
MOV A,#0AAH
CPL A            ; bütün bitlerin eşleniği alınır.
CLR A            ; bütün bitler temizlenir.
```

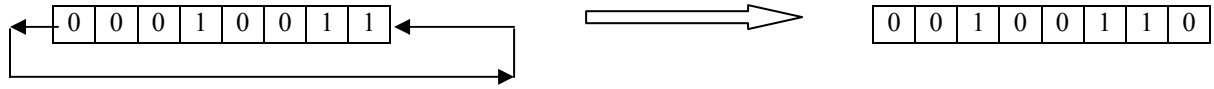
5.3.2.2.3. DÖNDÜRME-ÖTELEME İŞLEMLERİ (Rotate-Shift)

Döndürme işlemi özel register adresleme modunu ve sadece akümülatör kullanılır. Döndürme işlemleri için dört komut vardır: Sola ve sağa öteleme, sola ve sağa döndürme.Sırasıyla bir sola döndürme işlemiyle sayı 2 ile çarpılmış olur.Bu işlem için en düşük ağırlıklı bite 0 yazılır. Sağa döndürme işleminde ise kalan dikkate alınmaksızın sayı 2 ile bölünmüş olur. Elde ile sola döndürme işleminde ise elde bayrağındaki bit en düşük ağırlıklı bit olacak şekilde sola kaydırma işlemi oluşur. En ağırlıklı bit ise elde bayrağına geçer. Bu komuttan önce elde bayrağı kontrol edilmelidir. Elde ile sağa kaydırma işleminde de önce eldedeki bit en ağırlıklı bit olacak şekilde sağa kaydırma işlemi olur. En düşük ağırlıklı bit ise elde bayrağına geçer. Bu işlemden öncede elde bayrağı kontrol edilmelidir.

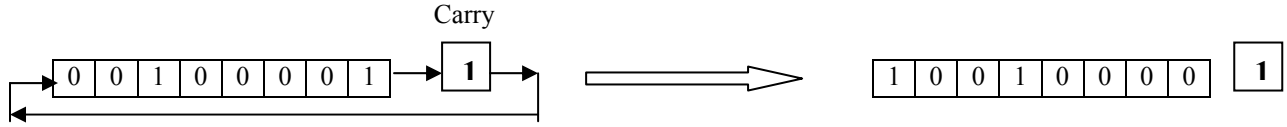
```
MOV A,#60H       ;bu işlemin ardından akümülatörde 30H sayısı oluşur
RR A             ;sayı ikiye bölünmüştür.
```




```
MOV A,#13H
RL A             ;işlemin ardından akümülatörde 26H sayısı oluşur.
                 ;sayı ikiyle çarpılmıştır.
```




```
MOV A,#21H
SETB C           ;eldeyi set et .elde en ağırlıklı bit olur sağa
RRC A            ;kaydırma işleminin ardından  en düşük ağırlıklı
                 ;bit eldeye geçer,akümülatörde 90H  değeri oluşur
                 ;ve elde set edilir.
```



5.3.2.2.4. SWAP (Yüksek ve düşük 4 bit' in (nibble) yerdeğiştirmesi)

Bu işlemde sadece özel register adresleme modunu kullanır. Bu komutun ardından akümülatördeki nibble lar başka lojik hiçbir işlem yapmadan değiştirilmektedir. Bu komut ASCII karekterlerine dönüştürme işlemi için kullanılan bir komuttur.

```
MOV A,#18H

SWAP A      ;bu komutun ardından akümülatörde 81H sayısı oluşur.
```

5.3.2.2.5. BİT-ORIENTED LOJİK İŞLEMLER

Lojik bit işlemleri bit olarak adreslenebilen herhangi bir adresteki bir bitle işlem yapmak için kullanılabilir. Bu komutlarla eşleniğini alma, temizleme, set etme, VE ve VEYA işlemleri uygulanabilir. VE ve VEYA işlemlerinde ilk işlem elemanı ve hedef elde bitidir. Ancak bu işlemin ardından elde biti istenilen yere yazılabilir. Bu komutlarda dikkat edilmesi gereken özel register adresleme modu kullanıldığıdır. Burada akümülatörü A ile değil de ACC şeklinde göstermek gereklidir. Bu komutların kullanıldığı örnekler aşağıdadır:

```
CLR A            ; akümülatörü temizle
SETB ACC.0       ; akümülatörün en düşük bitini temizle

SETB C           ; eldeyi set et
ORL C,/ACC.0     ; elde ile akümülatörün en düşük bitinin eşleniğine
                 ; veya işlemi uygula sonucu eldeye yaz
```

5.3.3. PROGRAM AKIŞI KONTROL KOMUTLARI KULLANIMI

Buraya kadar adresleme modları, aritmetik ve lojik işlemler anlatılmaktadır. Ancak bu komutlarla programlarda olması gereken kontrol ve bu kontrol işlemlerine göre diğer işlemler yapılamaz. Bu nedenle programlarda belirli yerlerde kontroller, bu kontrollerle alt program çağırma ve sıçrama komutları gereklidir. Bu komutlar sayesinde belirli koşullara göre sıçramalar ve döngü işlemleri yapılabilir. Ayrıca bu komutlar sayesinde mikrodenetleyici dış sistemlerle etkileşime geçebilmektedir. Örnek olarak seri olarak veri göndermek gerektiğinde veri öncelikle seri veri tamponuna alınır ve bit olarak gönderilir. Burada veri gönderimi bittiği zaman SFR deki seri veri iletişim arabirimindeki bir bit set edilir. Küçük bir alt program ve bu alt programın içindeki kontrollü bir döngü yardımıyla seri veri tamponundaki veri gönderilene kadar işlem beklenir.

Sıçrama ve dallanma komutlarında şimdiye kadar anlatılan komutlarda kullanılmayan bir yapı kullanılır. Bu komutlarla PC (Progaram Counter) registerına doğrudan veri yazılır. Şimdiye kadar anlatılanlarda ise PC mikrodenetleyici iç kontrolörüyle komutlara göre arttırılır, sıçrama ve dallanma komutlarında ise gidilecek program adresi yada etiketindeki adres yazılır.

Programlarda sıklıkla kullanılan ORG (ORIGIN) komutuyla ORG komutundan sonraki komutların bu adresten sonra program belleğine yerleştirilmesi sağlanır. Ayrıca

ORG komutuyla Timer ve diğer kesme alt programlarının reset kesme adresleri belirlenmiş olur. ORG komutunun bunların dışında program içerisinde kullanılması durumunda program hafızasında boş bölgeler oluşur ve boyut olarak büyük programlarda yer sorununa neden olur. Bu nedenle programlarda ve alt programlarda etiket kullanmak program hafızasının verimli kullanımı için önemlidir.

5.3.3.1. KOŞULSUZ DALLANMALAR

Bu komutlarda PC ' a gidilecek olan adres yazılır ve mikrodenetleyici bu adresteki komutu ve sonraki komutu yürütmeye başlar. Bu komutlarda herhangi bir koşul olmaksızın sıçrama yapılır. Dört farklı komut vardır: Kısa, mutlak, uzun ve indeksli sıçrama komutu. Kısa sıçrama komutunda 127 byte ileri veya 128 byte geri sıçrama yapılabilir. Mutlak sıçramada ise gidilecek yerin adresi 11 bittten oluşur, yani 2 kbyte lık blokluk program bölgesinde herhangi bir yere sıçrama yapılabilir. Uzun atlama komutuyla ise adres bilgisi 16 bittir ve bu 16 bitle 64 kbyte program hafızasında istenilen yere sıçrama yapılabilir. İndeksli atlama ise önemli bir atlama komutudur. Bu komutla öncelikle DPTR registerına atlanacak tablo yada bölgedeki ilk adres yazılır ve ardından akümülatörle indeksleme yapılarak A+DPTR adresindeki komuta sıçrama yapılır. İndeksli sıçrama ve diğer koşulsuz sıçrama komutlarına örnekler aşağıdadır:

```
      MOV A,40H              ; Not: burada 40H önceden başka bir program
      MOV DPTR,#TABLE        ; tarafından belirli bir sayı içermektedir
      JMP @a+DPTR

TABLO:
      LJMP ALT_1             ; ilk alt programa sıçra
      LJMP ALT_2             ; ikinci alt programa sıçra
      LJMP ALT_3             ;    .........
      LJMP ALT_4
      LJMP ALT_5
      .......
ALT_1:
      MOV P1,#0AAH           ; P1 'e  AAH değerini yolla ilk komutun
      AJMP DEVAM             ; ardından alt programın sonuna git


ALT_2:
      MOV P1,#02H
      AJMP DEVAM

ALT_3:
      MOV P1,#4BH
      AJMP DEVAM

ALT_4:
      MOV P1,#12H
      AJMP DEVAM

ALT_5:
      MOV P1,#56H
      AJMP DEVAM
```

```
      . . . . . .
DEVAM:
      . . . . . .
```

5.3.3.1. KOŞULLU DALLANMALAR

Bu komutlarla belirli koşullara göre belirli işlemlerin yapılması sağlanır. Elde, akümülatör yada herhangi bir adresteki bit veya byte koşullarına uygun olarak sıçrama yapılır. Bu komutlarla programlarda istenilen kontroller sağlanarak programın daha etkin olması sağlanır.

Koşullu sıçrama komutlarında aranan koşul gerçekleşiyorsa program belirlenen adrese sıçrama yapar, eğer koşul gerçekleşmiyorsa program bir sonraki komuttan devam ettirilir. Koşul sağlanırsa komutla birlikte verilen etiket yada program adresi PC ye aktarılır ve program bu adresteki komutları yürütmeye başlar.
Koşullu sıçrama komutları yapı olarak ikiye ayrılabilir;

1 - Eldenin, akümülatörün yada belirli bir bitin durumuna göre sıçrama ve belirli bir adresteki sayıyı girilen değere göre karşılaştırma.

2 - Azaltarak karşılaştırma işlemine göre sıçrama olarak ayrılabilir.

İlk komutlar bitlerin set yada temizlenmesine göre sıçrama yaparken ikinci komut yapıları byte ların adreslerdeki verilerle karşılaştırılıp sıçrama yapılmasını sağlar. Bu ikinci komut yapıları yüksek seviyeli dillerdeki for ve while döngülerindeki gibi belirli kontrollere göre belirli sayıda çevrim alt programları çalıştırabilirler. Aşağıda P1.0 'a bağlı olan bir led 12 Mhz kristal bağlı bir sistemde yaklaşık bir saniye aralıklarla yanıp sönmektedir.

```
      ORG 1000H
      CLR P1.0
BASLA:
      MOV R0,#0FFH      ;254 x 254 x 15 = 967740 mikro saniye bekleme
      MOV R1,#0FFH      ;alt  programı
      MOV R2,#10H

BEKLE_1:
      DJNZ R0,BEKLE_1
      DJNZ R1,BEKLE_1
      DJNZ R2,BEKLE_1
      CPL P1.0
      SJMP BASLA
```

5.3.3.2. ALT PROGRAM ÇAĞIRMALAR

Çağırma komutları alt programlarda çok önemli kullanım alanına sahiptir. Bu çağırma komutları sayesinde gerekli olduğu yerde alt programlar çağrılarak bu alt programlar çalıştırılıp alt program bittikten sonra ise ana programa tekrar dönülür. Çağırma komutları iki tanedir : Mutlak ve uzun çağırma komutları

Mutlak çağırma komutunda çağrılan alt programın adresi 11 bittir, yani 2 kbyte program bloğu içerisindeki herhangi bir yerden alt program çağrılabilir. Uzun çağırma komutta ise çağrılan alt programın adresi 16 bittir ve 64 kbyte program hafızası içerisinde herhangi bir yerden alt program çağrılabilir.

Alt programların çağrılması işleminde alt program adresi PC 'ye yüklenmeden önce 16 bitlik PC registerının önce yüksek sonra da düşük byte 'ı yığına itilir. PC yığına itildikten sonra gidilecek yerin adresi PC registerına yüklenir. Burada unutulmaması gereken bir nokta etiket kullanarak komut yürüten diğer komutlarda olduğu gibi burada da etiketin olduğu adres derleme işlemi sırasında belirlenir ve o etiketin kullanıldığı bütün yerlere etiketten sonraki ilk komutun program hafızasındaki adresi yazılır.

Alt programlardan çıkılıp tekrar ana programa dönülmesi için alt programların sonuna RET komutu yazılmalıdır. Eğer RET komutu alt programların sonuna yazılmazsa program aşağı doğru devam eder ve hatalar oluşur. Ayrıca kesme alt programlarında da benzer olarak RETI komutu kullanılır. Bu komutun RET komutuna göre farkı bu komutla mikrodenetleyici bir kesme servis alt programı işlemi yürüttüğünü ve bu işleme ait alt programın gerçekleştirildiğini belirten bayrakların set yada temizlenmesidir. RET yada RETI komutuyla yığından PC 'ın önce düşük ağırlıklı byte sonrada yüksek ağırlıklı byte 'ı alınır ve PC bu korunmuş değerlere göre işlemleri yapmaya devam eder. Bir önceki örnek başına bir etiket konup sonuna da RET konursa ve kısa sıçrama komutu kaldırılırsa led'i yaklaşık bir saniye aralıklarla yanıp söndüren bir alt program yazılmış olur ve istenildiği zaman çağrılabilir.

```
      ORG 1000H
BASLA:
      CLR P1.0
      MOV R3,#08              ;8 defa işlemi tekrarla
LED:
      ACALL LED_1             ;led yakıp söndürme alt programını çağır
      DJNZ R3,LED             ;led etiketine giderek 8 defa işlemi tekrarla
      SJMP BASLA              ;işlem bittiğinde yanıp sönme işlemi
                              ;tekrarı için işleme
                              ;devam et
LED_1:
      MOV R0,#0FFH            ;254 x 254 x 15 = 967740 mikro saniye
      MOV R1,#0FFH            ;bekleme alt programı
      MOV R2,#16H
```

```
BEKLE_1:
      DJNZ R0, BEKLE_1
      DJNZ R1, BEKLE_1
      DJNZ R2, BEKLE_1
      CPL P1.0
      RET
```

5.4. 8051 KOMUT SETİ KULLANIMINA DAİR TEMEL PROGRAMLAMA ÖRNEKLERİ

Assembly Dili Program örnekleri:

Buraya kadar anlatılan komut yapıları kullanılarak farklı birçok program örneği verilecektir. Bu kodlar ve yapılacak gerekli değişikliklerle anlatılacak örnekler 8051 mikrodenetleyicisi ile yapılabilecek birçok uygulama için yeterli ve yol gösterici olacaktır. Program örneklerinin birer alt program olabilmesi için başına bir etiket konmalıdır ve kodun sonuna ret komutu eklenmelidir.

Yazılan programlarda kod hatası olmamasına rağmen bazen program istenildiği gibi yada tam olarak çalışmaz. Bunun sebebi genellikle komut yapılarının yanlış kullanımı ve alt programlarda register değerlerinin değişmesidir. Bu hataları azaltmak için öncelikle komut yapısı iyice anlaşıldıktan sonra program yazım aşamasına geçilmelidir. Program içerisinde register değerlerinin değişimini önlemek içinse iki yol uygulanabilir. Birincisi alt programa girmeden önce alt programda kullanılacak registerların yığına itilmesi ikincisi ise alt programın içinde değeri değişen registerların yığına itilmesidir. Birinci yöntem alt program aynı akış içerisinde birçok defa çağrılıyorsa verimli olabilir. Burada registerlar ana programda bir kez yığına itilmekte ve alt programlar bittikten sonra tekrar yığından geri çekilmektedir. Bu yöntem sayesinde tekrarlanan alt programlarda her defasında registerların yığına itilip sonra alt program sonunda tekrar yığına çekilmesi ve mikrodenetleyicinin yığın işlemleri ile zaman kaybetmemesi sağlanmaktadır.

İkinci yöntem ise program akışı içerisinde değeri değişen registerların alt programın başında yığına itilmesi ve alt program sonunda yığından çekilmesidir. Bu yöntemin avantajı sadece o alt programda değişen registerlar yığına itilmektedir. Ancak bu yöntemin verimli olabilmesi için o alt programın çok fazla çağrılan bir alt program olmaması gerekir. Bu iki yöntem programın yapısına göre uygun bir şekilde kullanılarak register değerlerinin korunması sağlanabilir.

Burada anlatılan örneklerde registerların yığına itilmesi önemli olmayan durumlar dışında yapılmayacaktır. Bu nedenle buradaki alt programların kullanımından önce registerlar yığına itilmeli ve alt program sonunda tekrar yığından çekilmelidir.

5.4.1. MATEMATİKSEL İŞLEMLER

8051 komut setinde aritmetik işlemleri gerçekleştiren komutlar kullanılrak yapılmıştır.

5.4.1.1. ÜÇ BYTE' LIK DEĞİŞKENİN TOPLAMINI HESAPLAYAN PROGRAM

```
;SORU: 3'er byte lık iki değişkenin toplanarak
;sonucun 4 bytelık bir değişken içine atandığı bir program yazınız

;İlk sayı(ILK_SAYI_0/1/2) 06DH-06FH adres aralığına,
;ikinci sayı(IKINCI_SAYI_0/1/2) 06AH-06CH adres aralığına
;sonuc (SONUC_0/1/2/3)  066H-069H  adres aralığına yerleştirilmiştir.

      ILK_SAYI_0        DATA  6FH         ;ilk sayının değer
      ILK_SAYI_1        DATA  6EH         ;değişkenleri
      ILK_SAYI_2        DATA  6DH

      IKINCI_SAYI_0     DATA  6CH         ;ikinci sayının değer
      IKINCI_SAYI_1     DATA  6BH         ;değişkenleri
      IKINCI_SAYI_2     DATA  6AH

      SONUC_0           DATA  69H         ;sonucun değer
      SONUC_1           DATA  68H         ;değişkenleri
      SONUC_2           DATA  67H
      SONUC_3           DATA  66H
;------------------------------
;basit toplamada olduğu gibi iki sayının en küçük
;kısımları toplanır

      MOV   A,ILK_SAYI_0
      ADD   A,IKINCI_SAYI_0
      JNC   TPL_DVM0
      INC   SONUC_1
TPL_DVM0:

      MOV   SONUC_0,A
;------------------------------
;en kucuk ikinci kısımlar toplanıyor

      MOV   A,ILK_SAYI_1
      ADD   A,IKINCI_SAYI_1
      JNC   TPL_DVM1
      INC   SONUC_2

TPL_DVM1:

      ADD   A,SONUC_1
      JNC   TPL_DVM2
      INC   SONUC_2

TPL_DVM2:
```

```
      MOV   SONUC_1,A
;---------------------------
;büyük kısımlar toplanıyor
      MOV   A,ILK_SAYI_2
      ADD   A,IKINCI_sAYI_2
      JNC   TPL_DVM3
      INC   SONUC_3

TPL_DVM3:

      ADD   A,SONUC_2
      JNC   TPL_DVM4
      INC   SONUC_3

      MOV   SONUC_2,A

      RET
;************************
```

5.4.1.2. N ADET BİR BYTE' LIK DEĞİŞKENİN TOPLAMINI HESAPLAYAN PROGRAM

```
;SORU:N adet bir byte lık değişkenin toplamını iki bytelık
;bir değişken içine atayan program

;NOT:BU PROGRAM 101 ADET BİR BYTELIK SAYIYI
;TOPLAYABİLİR

N_TOPLA:

;Sayılar 40h adresinden başlasın
SAYAC DATA  6FH

      SONUC_L     DATA  6EH
      SONUC_H     DATA  6DH

      MOV   R0,#40H
      MOV   SAYAC,"n"
;---------------
N_TPL_DONGUSU:

      ADD   A,@R0
      JNC   N_TPL_DVM
      INC   SONUC_H

N_TPL_DVM:

      INC   R0
      DJNZ  SAYAC,N_TPL_DONGUSU
;----------------
      MOV   SONUC_L,A

      RET
```

5.4.1.3. İKİ BYTE' LIK DEĞİŞKENLER ÜZERİNDE ÇIKARTMA İŞLEMİ PROGRAMI

```
      ;31H ve 30H adreslerindeki 2 byte'lık  sayıdan
      ;41H ve 40H adreslerindeki sayı çıkarılıp sonuç
      ;51H ve 50H adreslerinde saklanmaktadır.

16BIT_CIKARMA:
      MOV A,30H
      SUBB A,40H
      MOV 50H,A
      JNC DEVAM         ; çıkarma işleminde elde ödünç
      DEC 31H           ; alınıyorsa 31H i azalt
DEVAM:
      MOV A,31H
      SUBB A,41H
      MOV 51H,A
      RET
```

5.4.1.4. İKİ BYTE' LIK İKİ DEĞİŞKENİN ÇARPIMINI HESAPLAYAN PROGRAM

```
      D1_H  DATA  078H              ;Sayıların yüksek ve düşük
                                    ;bytelarının atanması
      D1_L    DATA      079H
      D2_H    DATA      07AH
      D2_L    DATA      07BH

      S1      DATA      07CH        ;sonuc bytelarının yazılacağı
                                    ;adreslarin atanması
      S2      DATA      07DH
      S3      DATA      07EH
      S4      DATA      07FH        ;not:lütfen işlemden önce
                                    ;içeriklerini sıfırlayınız.
      ELDE1 DATA  077H              ;yığından eldelerin çekilebilmesi
                                    ;için kullanılacak olan ;değişkenler
      ELDE2 DATA  076H

;Iki_16bitlik_Sayiyinin_Carpimi:

;BIRINCI BASAMAKLARIN ÇARPIMI

      MOV   A,D1_L
      MOV   B,D2_L
      MUL   AB

      MOV   S1,A                    ;Sonucun birinci byteını ata
      MOV   R1,B                    ;birinci çarpımın yüksek byteını
                                    ;elde için R1' de sakla
;IKINCI SAYININ BIR,BIRINCI SAYININ IKINCI BASAMAK CARPIMI

      MOV   A,D2_L
      MOV   B,D1_H
      MUL   AB
```

```
      MOV   R2,A                    ;ikinci carpımın düşük byteını
                                    ;elde için R2' ye ata

      PUSH  B                       ;ikinci çarpımın yüksek byteını
                                    ;elde için yığına gönder
;IKINCI SAYININ IKI BIRINCI SAYININ BIRINCI BASAMAK CARPIMI

      MOV   A,D1_L
      MOV   B,D2_H
      MUL   AB

      PUSH  B                       ;Carpımın yüksek byteını elde
                                    ;için yığına gönder
;IKINCI BYTE ın ELDELERININ TOPLANMASI

      ADD   A,R1
      JNC   DVM                     ;yeni elde kontrolü
      INC   S3                      ;elde durumundaki arttırma

DVM:
      ADD   A,R2
      JNC   DVM2                    ;yeni elde kontrolü
      INC   S3                      ;elde durumundaki arttırma
DVM2:
      MOV   S2,A                    ;toplamda gelen veriyi  sonucun
                                    ;ikinci byteına ata
;IKINCI BASAMAKLARIN CARPIMI

      MOV   A,D2_H
      MOV   B,D1_H
      MUL   AB

;UCUNCU BYTE ın ELDELERİNİN TOPLANMASI

      POP   ELDE1             ;ilk eldeyi yığından çek
      MOV   R1,ELDE1          ;ve R1 e ata
      POP   ELDE2             ;ikinci eldeyi yığından çek
      MOV   R2,ELDE2          ;ve R2 ye ata
      ADD   A,R1            ;Birinci eldeyi ekle
      JNC   DVM3            ;yeni elde kontrolü
      INC   S4              ;elde durumundaki arttırma
DVM3:
      ADD   A,R2
      JNC   DVM4             ;yeni elde kontrolü
      INC   S4               ;elde durumundaki arttırma

DVM4:
      ADD   A,S3
      JNC   DVM5             ;yeni elde kontrolü
      INC   S4               ;elde durumundaki arttırma

DVM5:
      MOV   S3,A             ;sonucun ücüncü byteını ata
      MOV   A,B
```

```
;DORDUNCU BYTEın ELDELERİNİN TOPLANMASI

      ADD   A,S4
      MOV   S4,A            ;Sonucun dorduncu byteını ata

      RET
```

5.4.1.5. BİR BYTE' LIK BİR DEĞİŞKENİN İÇERİĞİNİ DİGİTLERE (HANE DEĞERLERİNE) AYIRAN PROGRAM

```
;SORU:Bir baytlık bir sayı değişkenin yüzler onlar ve birler basamakları
;olarak ayır

BASAMAKLARA_AYIRMA:

      BIRLER      DATA  30h   ;BIRLER basamağının tutulacağı yer
      ONLAR       DATA  31h   ;ONLAR basamağının tutulacağı yer
      YUZLER      DATA  32h   ;YUZLER basamağının tutulacağı yer


      MOV   A,#k              ;Ayıklanıacak sayıyı A ya ata
      MOV   B,#100            ;B ye 100 ata
      DIV   AB                ;A yı B ye böl
                              ;Not:Bu işlemin özelliğinde sonuc
                              ;A'ya kalan Byte yazılır


      MOV   YUZLER,A          ;Sonuc olarak gelen yüzler
                              ;basamağını yaz
      MOV   A,B               ;Geri kalan onlar ve birler
                              ;basamaklarını A ya yaz
      MOV   B,#10             ;B ye 10 ata
      DIV   AB                ;A yı B ye böl
      MOV   ONLAR,A           ;Onlar basamağını yaz
      MOV   BIRLER,B          ;Birler basamağını yaz
      RET
```

5.4.1.6. 16 ADET BİR BYTE' LIK DEĞİŞKENİN ARİTMETİK ORTALAMASINI HESAPLAYAN PROGRAM

Bu alt programda 16 sayının ortalamasını alan bir alt program örneği verilmiştir. 30H den 3FH e kadar olan 16 sayının ortalaması alınarak 61H adresinde saklanmaktadır. Ayrıca kalan 60H adresine yazılmaktadır. Öncelikle 16 sayı toplanmakta ve ardından bu toplam 16 ya bölünmektedir. Bilindiği herhangi bir tabandaki sayıyı o tabana bölmek o sayıyı sağa bir basamak kaydırmak demektir. Örneğin onluk tabanda bir sayıyı 10 a bölmek için sayıyı bir sağa kaydırmak ve dolayısıyla varsa virgülü sola kaydırmak demektir. Burada da toplanan sayıyı 16 ya bölme işlemi için toplam sayısı bir nibble sağa kaydırılmaktadır. Oluşan toplam en fazla üç nibble dan oluşabileceğinden üç defa kaydırma işlemi yapılmaktadır.

```
ORTALAMA:
      MOV R0,#30H             ; ilk adres
      MOV A,30H               ;ilk değeri yükle
      CLR C                   ; sonraki işlemler için eldeyi temizle
DONGU:
      INC R0                  ; bir sonraki adres
      ADD A, @R0              ; değerleri topla
      JNC  DEVAM              ; elde oluşuyorsa
      INC 41H                 ; bir üst dijiti bir arttır
      CLR C                   ; sonraki işlem için eldeyi temizle
DEVAM:
      CJNE R0,#3FH,DONGU      ; 15 kez toplama işlemi yap
      MOV 40H,A
BOLME:
      MOV A,40H               ; 40H adresindeki düşük nibble ı kalan olarak
      ANL A,#0FH              ; 60H  adresine yaz
      MOV 60H,A
      MOV A,40H               ; 40H adresindeki yüksek nibble ı 61 adresinin
      ANL A,#0F0H             ; düşük nibble ı olarak yaz
      SWAP A                  ; nibble ları yer değiştir
      MOV 61H,A
      MOV A,41H               ; 41H adresindeki düşük nibble ı yükle
      SWAP A                  ; düşük nibble ı yüksek nibble yap
      ADD A,61H               ; 61H deki düşük nibble ı yükle
      MOV 61H,A               ; akümülatördeki yüksek nibble ile
                              ;topla ve kaydet
      RET
```

5.4.1.7. N BYTELIK SAYININ İŞARET DEĞİŞİMİ

İşaretli sayılarla direk işlem yapılamadığı zaman yada işaret değişimi yapmak gerektiğinde bu alt programla kolayca bu işlem gerçekleştirilebilmektedir. Bilindiği gibi işaretli sayıların en yüksek ağırlıklı byte ındaki en yüksek ağırlıklı biti 0 ise sayı pozitif, 1 ise sayı negatiftir. Pozitif bir sayının negatif karşılığı bulunmak için öncelikle sayının bütün bitlerinin eşleniği (complement) alınır. Bu işlemin ardından sayıya 1 eklenerek istenen negatif sayı bulunur. Negatif bir sayının pozitif karşılığı da aynı şekilde bulunur. Bu yöntem sayesinde mikroişlemci sistemlerde kolaylıkla işaret dönüşümü yapılabilmektedir.

Aşağıda verilen örnekte 20H den 2FH e kadar olan adreslerde tutulan maximum 16 byte uzunluğundaki sayının işaret değişimi yapılmaktadır. Ancak istenirse bu uzunluk daha da arttırılabilir. Bu alt programda önce sayının uzunluğu bulunmakta sonra bu sayının işaret değişimi yapılmaktadır.

```
      SAYAC       DATA  070H        ; sayının byte sayısı (N değeri).
      ELDE        DATA  071H        ;işlemde oluşan eldenin tutulacağı alan

ISARET_DEGISIMI:

      MOV   R0,#020H                ;sayının LSB kısmının basladıgı adres
      MOV   SAYAC,#10H              ;sayının kac byte oldugu bilgisi

      MOV   A,@R0                   ;R0'ın gösterdiği adresteki ilk byte
                                    ;(LSB) A'ya yazılır.
      CPL   A                       ;Complementi alınır.
      ADD   A,#01                   ;Sayıya 1 eklenir.
      JNC   ELDE_YOK_0              ;Carry yoksa diğer byte gec.
      MOV   ELDE,#01                ;Carry varsa elde buffer 'ına yazılır.
ELDE_YOK_0:
      MOV   ELDE,#0                 ;Carry yoksa elde bufferı temizlenir.
      MOV   @R0,A                   ;Diğer byte A'ya alınır
      INC   R0                      ;Adres değeri 1 arttırılır.
      DEC   SAYAC                   ;Byte sayısı 1 azaltılır.
      MOV   A,SAYAC                 ;byte sayısı kontrol için A'ya alınır.
      JZ    SON                     ;sayı bitmişse(tek byte ise)SON'a git
TOP:                                ;ilk byte'tan sonraki byte'ler dongu
                                    ;içinde tümleyenleri alınır.
      MOV   A,@R0                   ;Diğer byte A'ya alınır
      CPL   A                       ;Tümleyeni alınır.
      ADD   A,ELDE                  ;ELDE buffer'ındaki değer eklenir.
      JNC   ELDE_YOK                ;Carry yoksa ELDE_YOK Label 'ına git.
      MOV   ELDE,#01                ;Carry varsa ELDE buffer'ına 1 yaz.
ELDE_YOK:                           ;Carry olmaması durumunda
      MOV   ELDE,#0                 ;ELDE buffer'ı temizlenir.
      MOV   @R0,A                   ;Byte yeni haliyle tekrar yazılır.
      INC   R0                      ;adres değeri arttırılır.
      DJNZ  SAYAC,TOP               ;SAYAC 0 değilse TOP'a git.
SON:
RET
```

5.4.1.8. DPTR REGİSTERİNİN AZALTILMASI

DPTR registerini azaltmak

8051 komut sisteminde çok kullanılmadığı için DPTR registerini azaltmak için bir komut yoktur. DPTR registeri herhangi bir 16 bit adrese ulaşma imkanı sağlar. 16 bit adrese yada herhangi bir sırçama tablosuna ulaşımda bir sonraki değere geçme işlemi için DPTR yazmacı arttırılır. Küçük bir alt programla bu azaltma işlemi kolaylıkla yapılabilir. DPTR registeri iki tane 8 bitlik DPH ve DPL registerlerinden oluşur. Burada azaltma işlemi için DPL registeri azaltılmaktadır. Ancak dikkat edilmesi gereken DPL registerinde 00H değeri varsa azaltma işleminin ardından DPL registeri FFH değerine taşar. Burada önce DPL registerinden 1 çıkarılmakta, eğer 00H değerinden çıkarma işlemi oluşuyorsa elde set edileceğinden DPH registerini 1 azaltılmakta elde set edilmiyorsa işlem bitmektedir.

```
DEC_DPTR:
      CLR C                  ;sonraki işlemler için elde bayrağını  temizle
      MOV A,DPL              ;akümülatöre DPL yi yükle
      SUBB A,#01             ;akümülatörden 1 çıkar
      JNC SON3               ;elde set edilmemişse son3 e git
      DEC DPH                ;elde set edilmişse DPH i azalt
SON3:                        ;bitir
      RET                    ;geri dön
```

5.4.2. BLOK AKTARMA PROGRAMLARI

Veri bloğu transferi

Programlarda bazen harici belleğin bir kısmının başka bir yere transferi gerekmektedir. Bunun için R0 ve R1 registerları kullanılarak MOVX komutuyla veri transferi alt programı hazırlanmıştır. Kaynak ve varış harici hafıza olduğundan 16 bitlik registerlarla harici veri blokları gösterilmelidir ancak 8051 de sadece bir tane 16 bitlik register vardır. Bu nedenle P2 üzerinden harici belleğin yüksek ağırlıklı byte programın içerisinde byte olarak girilmektedir. Burada öncelikle P2 kullanılarak erişim yapılmalı ardından DPTR ile adresleme yapılmalıdır. Bu nedenle P2 üzerinden yapılan adreslemede kaynak olan harici veri belleği adreslenmeli sonra ise hedef olan yere DPTR registerı ile veri yazılmalıdır. Bu örnekte 0450H adresinden başlayarak 4FH kadar veri AA00H adresine transfer edilmektedir.

```
TRANSFER:

      MOV DPTR,#0AA00H       ;hedef adresi
      MOV P2,#4              ;kaynak adresin yüksek ağırlıklı byteı

      MOV R0,#50H            ;kaynak adresin düşük ağırlıklı byte'ı
      MOV R1,#4FH            ;4FH kadar transfer yap
```

```
GONDER:
      MOVX A,@R0              ;kaynaktan oku
      MOVX @DPTR,A            ;hedefe yaz
      INC R0                  ;kaynak işaretçisini arttır
      INC DPTR                ;hedef işaretçisini arttır
      DJNZ R1,GONDER          ;4FH defa işlemi yap
      RET
```

5.4.2.1. INTERNAL RAM' DE YER ALAN N BYTE' LIK VERİ BLOĞUNU INTERNAL RAM' DE BAŞKA BİR ADRESE AKTARAN PROGRAM

```
;Dahili hafıza üzerindeki 40h ve üzerindeki adreslemede yer
;alan "k" bytelık veri bloğunu hedef adresine aktaran bir ;program
;     a)Hedef adres>Başlama Adres + k
;     b)Hedef adres<Başlama Adres
;     c)Hedef adres>Başlama Adres

;A)
YETERLI_ARALIKTA_TABLO_AKTARMA:

      SAYAC DATA  30h         ;Sayacının değerinin tutulacağı  adres.
      MOV   R0,Bas_Adrr       ;Başlangıç adresini R0 a ata
      MOV   R1,Hdf_Adrr       ;Hedef Başlangıç adresini R1'e ata
      MOV   SAYAC,#k          ;Sayac değerini ata

DONGU:
      MOV   A,@RO             ;R0 ın gösterdiği bilgiyi A'ya ata
      MOV   @R1,A             ;A nın içeriğini R1 in gösterdiği adrese ata
      INC   R0                ;R0 ı arttır
      INC   R1                ;R1 i arttır
      DJNZ  SAYAC,DONGU       ;Sayac sıfırlanana kadar döngüyü tekrarla
      RET                     ;Alt işlemden çık

;NOT:BU PROGRAMDA VERİLERİN ÜST ÜSTE BİNMESİ GİBİ BİR SORUN OLMADIĞINDAN
BAŞTAN BAŞLANMIŞTIR
;B)
BELIRLI_ARALIKTA_TABLO_AKTARMA:
      SAYAC DATA  30h         ;Sayacın değerinin tutulacağı adresine ata
      MOV   R0,Bas_Adrr       ;Başlangıç adresini R0 a ata
      MOV   A,R0
      ADD   A,#kh             ;Aktarıma sondan başlayabilmek için
      MOV   R0,A              ;R0 ı k kadar ötele
      MOV   R1,Hdf_Adrr       ;Hedef Başlangıç adresini R1 e ata
      MOV   A,R1
      ADD   A,#kh             ;Aktarıma sondan başlayabilmek için
      MOV   R1,A              ;R1 i k kadar ötele

      MOV   SAYAC,#kh         ;Sayac değerini ata

DONGU:
      MOV   A,@RO           ;R0 ın gösterdiği bilgiyi A ya ata
      MOV   @R1,A           ;A nın içeriğini R1 in gösterdiği adrese ata
      DEC   R0              ;R0 ı azalt
      DEC   R1              ;R1 i azalt
```

```
      DJNZ  SAYAC,DONGU     ;Sayac sıfırlanana kadar döngüyü tekrarla
      RET                   ;Alt işlemden çık

;NOTLAR:BU PROGRAMDA İKİ ADRES ARASI BOŞLUK BİLİNMEDİĞİNDEN
;AKTARIMA SONDAN BAŞLANMIŞTIR BU SAYEDE AKTARILMIŞ VERİLERİN
;DAHA AKTARILMAMIŞ OLAN VERİLERİN ÜZERİNE YAZLIARAK VERİ KAYBI
;OLMASI ENGELLENMİŞTİR.
;C)

YETERLI_ARALIKTA_TABLO_AKTARMA:
      SAYAC DATA  30h         ;Sayacın değerinin tutulacağı  adres
      MOV   R0,Bas_Adrr       ;Başlangıç adresini R0 a ata
      MOV   R1,Hdf_Adrr       ;Hedef Başlangıç adresini R1'e  ata
      MOV   SAYAC,#k          ;Sayac değerini ata
DONGU:
      MOV   A,@RO             ;R0 ın gösterdiği bilgiyi A ya ata
      MOV   @R1,A             ;A nın içeriğini R1 in gösterdiği adrese ata
      INC   R0                ;R0 ı arttır
      INC   R1                ;R1 i arttır
      DJNZ  SAYAC,DONGU       ;Sayac sıfırlanana kadar döngüyü tekrarla
      RET                     ;Alt işlemden çık


;NOTLAR:KÜÇÜK OLAN HEDEF ADRESLER KULLANILDIĞINDA AKTARIMA İLK ;ADRESTEN
BAŞLANMASI VERİ KAYBINI ENGELLER
```

5.4.2.2. INT_TO_XRAM; XRAM_TO_INT PROGRAMLARI

İç veri belleğinden dış veri belleğine veri aktarımı

İç veri belleğinin yeterli olmadığı durumlarda dış veri belleği kullanmak gerekebilir. Ancak dış veri belleğine erişim iç veri belleğine erişimden daha fazla zaman alır. Bu nedenle iç veri belleği kullanılmalıdır, iç veri belleğinin yetmediği durumlarda dış veri belleği kullanılmalıdır. İç veri belleğinin 256 byte olması nedeniyle çok fazla veri belleği kullanan programlarda dış veri belleği de kullanmak gerekebilir. Bu alt programda iç veri belleğinden dış veri belleğine belirli bir sayıda byte transferi yapılmaktadır. Bunun için dış veri belleğini adreslemede DPTR registerı kullanılmaktadır. 30H adresinden 7FH adresine kadar olan veri belleği 1000H adresinden 104FH adresine kadar kopyalanmaktadır.

```
INTRAM2XRAM:
      MOV R0,#30H             ; kaynak ilk adresi yükle
      MOV DPTR,#1000H         ; hedef adresini DPTR e yükle
KOPYALA:
      MOV A,@R0               ; R0 ın gösterdiği adresteki değeri
                              ; akümülatöre al
      MOVX @DPTR,A            ; akümülatördeki değeri DPTR nin gösterdiği
                              ; yere yaz
      INC R0                  ; kaynak adresini arttır
      INC DPTR                ; hedef adresini arttır
      CJNE R0,#80H,KOPYALA    ; 30H den 7FH e kadar devam et
      RET
```

Dış veri belleğinden iç veri belleğine veri transferi

Dış veri belleğinde bulunan verilerin iç veri belleğine alınıp burada işlem yapılması gerekebilir. Bu durumda yukarıda anlatılan işlem benzer bir şekilde veri transferi yapılır. Ancak burada kaynak dış veri belleği hedef iç veri belleğidir. Alt programda 1000H adresinden 104FH adresine kadar olan veriler, iç veri belleğinde 30H den 7FH e kadar kopyalanmıştır.

```
XRAM2INTRAM:
      MOV R0,#30H             ; hedef ilk adresi yükle
      MOV DPTR,#1000H         ; kaynak adresini DPTR e yükle

KOPYALA2:
      MOV A,@DPTR             ;DPTR  nin gösterdiği adresteki  değeri
                              ;akümülatöre al
      MOVX @R0,A              ;akümülatördeki değeri R0 ın gösterdiği yere
                              ;yaz
      INC R0                  ; hedef adresini arttır
      INC DPTR                ; kaynak adresini arttır
      CJNE R0,#80H,KOPYALA2   ; 30H den 7FH e kadar devam
      RET
```

5.4.2.3. EXTERNAL RAM' DE YER ALAN N BYTE' LIK VERİ BLOĞUNU EXTERNAL RAM' DE BAŞKA BİR ADRESE AKTARAN PROGRAM

Dış veri belleğinden dış veri belleğine veri transferi

Burada 1000H adresinden başlayarak N adet veri 2000H adresinden itibaren kopyalanmaktadır.

```
XRAM2XRAM:
      MOV DPTR,#1000H         ;kaynak adresini yükle
      MOV R0,#00H             ;R0 ı sıfırla

KOPYALA3:
      MOV DPH,#10H            ;kaynak adresi için DPH 10H,yani DPTR ye 1000H
      MOVX A, @DPTR           ;yükle ve bu adresten akümülatöre kopyala
      MOV DPH,#20              ;DPH a 20H, yani DPTR ye 2000H değerin yükle
      MOVX @DPTR,A            ;akümülatördeki sayıyı DPTR nin gösterdiği
                              ;yere yaz
      INC DPTR                ;DPTR yi arttır
      INC R0                  ;kopyalama işlemi sayısı için R0ı arttır
      CJNE R0,#30H,KOPYALA3   ;47 defa kopyalama işlemi yap
      RET
```

5.4.2.4. İKİ ADET DPTR KULLANARAK XRAM DE BLOK AKTARMA PROGRAMI

İki DPTR kullanarak XRAM den XRAM e veri transferi

XRAM kullanarak veri transferi yapılan işlemlerde adresi göstermek için DPTR registeri kullanlır, ancak XRAM den XRAM e veri tarnsferi yapabilmek için iki DPTR registeri kullanmak gereklidir. İki DPTR registeri olmadığı durumlarda ikinci bir DPTR registeri yazılımla oluşturulabilir. Burada DPTR registerinin DPH ve DPL registerlerini kullanarak ikinci bir DPTR registeri varmış gibi işlem yapılabilir. Bunun için iki adres DPH ve DPL registerlerine kopyalanarak DPTR registeri iki işlem içinde kullanılabilir. Böylece XRAM den XRAM e veri tarnsferi yapmak mümkün olabilmektedir.
Bu alt programda 1000H adresinden başlayarak N adet byte 2000H adresinden başlayarak kopyalanmaktadır. 30H adresinde N adet byte bilgisi tutulmaktadır.

```
XRAM2XRAM:
      MOV DPTR,#2000H
      MOV 40H,DPL
      MOV 41H,DPH
      MOV DPTR,#1000H
GONDER:
      MOVX A, @DPTR
      INC DPTR
      PUSH DPH
      PUSH DPL
      INC 40H
      CJNE 40H,#00,DEVAM          ; 40H adresinde FFH den 00H e
      INC 41H                     ; taşma varsa 41H arttır.
DEVAM:
      MOV DPH,41H
      MOV DPL,40H
      MOVX @DPTR,A
      POP DPL
      POP DPH
      DJNZ 30H,GONDER
      RET
```

5.4.3. ARAMA PROGRAMLARI

5.4.3.1. KARŞILAŞTIRMA PROGRAMLARI

```
;Belli bir k boyutlu adres aralığından oluşan bir tablodaki bölümlerde
bulunan değerler arasında

;     a)En küçük sayıyı bul
;     b)Tek sayıların adedini bul
;     c)Çift sayıların adedini bul
;     d)En büyük sayıyı bul

;A)

EN_KUCUK_DEGER:

      EN_KUCUK DATA     31h         ;Sonucun saklanacağı yeri belirle
      SAYAC DATA  30h               ;Sayac değerinin saklanacağı yeri
                                    ;belirle
      MOV   R0,#TUT                 ;İlk tutulacak sayının adresine
                                    ;yaz
      MOV   R1,#TUT+1               ;İlk karşılaştırılacak sayının
                                    ;adresi
      MOV   SAYAC,#k                ;Sayacın değerini ata

CIKAR:
      MOV   A,@R0                   ;Bu bölümde R0 da gösterdiğimiz
                                    ;sayıdan  R1  de  gösterdiğimiz  sayıyı
                                    ;çıkararak
      SUBB  A,@R1                   ;elde oluşup oluşmadığına göre
                                    ;daha küçük olan  sayı belirlenir
      JNC     DEGIS                 ;Elde durumuna göre atlama veya devam
                                    ;etme(elde yoksa atla)

DEGISME:
      INC   R1                      ;Bir dahaki sayı için R1 i arttır
      MOV   EN_KUCUK,R0             ;En küçük sayı adresine R0 ı ata
      DJNZ  SAYAC,CIKAR             ;Bir dahaki sayıyı deneme işlemine;
                                    ;geç

      RET                           ;Alt işlemi bitir

DEGIS:
      MOV   A,R1                    ;A yardımı ile R1 içeriğini al
      MOV   R0,A                    ;A yı R0 içine yaz
      INC   R1                      ;Bir dahaki sayı için R1 i arttır
      MOV   EN_KUCUK,R0             ;En küçük sayı adresine R0 ı ata
      DJNZ  SAYAC,CIKAR             ;Bir dahaki sayıyı deneme işlemine
                                    ;geç
      RET                           ;Alt işlemi bitir
```

```
;B)
TEK_SAYI_ADEDI:

      ADET  DATA  30h              ;Tek sayı adedinin tutulacağı bölüm
      SAYAC DATA  31h              ;Sayacın tutulacağı bölüm

      MOV   ADET,00h               ;Adet içeriğini sıfırla
      MOV   R0,#BASAdrr            ;R0 a ilk adresi ata

SORGU:
      MOV   A,#00000001b           ;Sorgu işlemi Sayının 000000001b
                                   ;sayısıyla AND
      ANL   A,@R0                  ;Kapısı kullanılarak yapılır ve ANL
                                   ;komutunun özelliğinden
                                   ;sonuc 1 veya 0 olarak A içeriğine
                                   ;yazılır
      JNZ   TEK                    ;A nın içeriği 0 değil ise TEK
                                   ;etiketine atla
      INC   R0                     ;A nın içeriği 0 ise R0 ı arttır
      DJNZ SAYAC,SORGU             ;Yeni sayıyı sorgulamak için başa dön
      RET

TEK:                               ;Tek sayı ise
      INC   ADET                   ;ADET içeriğini arttır
      INC   R0                     ;R0 ı arttır
      DJNZ SAYAC,SORGU             ;Yeni sayıyı sorgulamak için başa dön
      RET

;C) Çift sayı adedini bulma
;Bu alt programda 30H den 40H kadar olan adresteki sayılar ;içinden çift
sayı adedi bulunmakta ve 50H adresinde kaç tane ;olduğu gösterilmektedir.
Elde ile birlikte döndürme komutu ;kullanılarak son bitin 0 veya 1 olmasına
göre çift yada tek ;olmasına bakılır.

      MOV R0,#30H
CIFT:
      MOV A, @R0
      INC R0
      RRC A
      JC DEVAM
      INC 50H
DEVAM:
      CJNE R0,40H,CIFT
      RET
;D)
EN_BUYUK_DEGER:

      EN_BUYUK DATA     31h        ;Sonucun saklanacağı yeri belirle
      SAYAC    DATA     30h        ;Sayac değerinin saklanacağı yeri
                                   ;belirle
      MOV   R0,#TUT                ;İlk tutulacak sayının adresini yaz
      MOV   R1,#TUT+1              ;İlk karşılaştırılacak sayının adresi
      MOV   SAYAC,#k               ;Sayacın değerini ata
```

```
CIKAR:
      MOV   A,@R0                ;Bu bölümde R0 da gösterdiğimiz
                                 ;sayıdan R1 de gösterdiğimiz sayıyı
                                 ;çıkararak
      SUBB  A,@R1                ;elde oluşup oluşmadığına göre daha
                                 ;büyük olan  sayı belirlenir
      JC    DEGIS                ;Elde durumuna göre atlama veya devam
                                 ;etme(elde varsa atla)

DEGISME:
      INC   R1                   ;Bir dahaki sayı için R1 i arttır
      MOV   EN_BUYUK,R0          ;En büyük sayı adresine R0 ı ata
      DJNZ  SAYAC,CIKAR          ;Bir dahaki sayıyı deneme işlemine geç

      RET                        ;Alt işlemi bitir

DEGIS:
      MOV   A,R1                 ;A yardımı ile R1 içeriğini al
      MOV   R0,A                 ;A yı R0 içine yaz
      INC   R1                   ;Bir dahaki sayı için R1 i arttır
      MOV   EN_BUYUK,R0          ;En büyük sayı adresine R0 ı ata
      DJNZ  SAYAC,CIKAR          ;Bir dahaki sayıyı deneme işlemine geç

      RET                        ;Alt işlemi bitir
```

5.4.4. VERİ FORMATI DÖNÜŞTÜRME PROGRAMLARI

5.4.4.1. BINARY/BCD DÖNÜŞTÜRME PROGRAMLARI; BIN_TO_BCD; BCD_TO_BIN

Binary BCD dönüşümleri

Burada binary sayılardan BCD (Binary Coded Decimal) sayılara ve BCD sayılardan binary sayılara dönüşüm işlemi için iki alt program yazılmıştır. BCD sayılarla işlem bildiğimiz 10 tabanında yapılan işlemlerin 16 tabanındaki mikrodenetleyici sistemlerle yapılan işlemler arasında dönüşüm için kullanılır.

Binaryden BCD ye dönüşüm yapan alt programda 30H de bulunan sayı dönüşüm işleminden sonra onlar ve birler basamağı 40H de yüzler basamağı ise 41H de saklanmaktadır.

BCD den binary e dönüşüm işleminde 30H adresindeki sayı dönüşüm işlemi ardından 40H adresinde saklanmaktadır.

```
BINARY2BCD:

      MOV A,30H         ;
      MOV B,#0AH  ;
      DIV AB            ;
      MOV 40H,B         ; birler basamağını 40h adresine yaz
      MOV B,#0AH  ;

      DIV AB            ;
      MOV 41H,A         ; yüzler basamağını 41H adresine yaz
      MOV A,B           ; onlar basamağını akümülatöre yaz
      SWAP A            ; onlar basamağı olarak değiştir
      ADD A,40H         ; onlar basamağını birler basamağıyla
      MOV 40H,A         ; topla ve sonucu 40H adresine yaz
      RET               ;

BCD2BINARY:

      MOV A,30H         ;
      ANL A,#0F0H       ;alt nibble ını sil
      SWAP A            ;nibble ları değiştir
      MOV B,#0AH  ;
      MUL AB            ;
      MOV A,B           ;
      MOV 32H,A         ;A'daki sayı geçici olarak 32H yaz
      MOV A,30H         ;
      ANL A,#0FH        ;üst nibble ını sil
      ADD A,32H         ;geçici değerle topla
      MOV 40H,A         ; sonucu yaz
      RET
```

5.4.4.2. BINARY VE ASCII DÖNÜŞÜMLERİ

Haberleşme sistemlerinde ve değişik uygulamalarda sıklıkla kullanılan binary ve ASCII dönüşümleri için kullanılan iki alt program yazılmıştır. Bilindiği gibi ASCII tablosu 256 karakterden oluşmaktadır. Bunlardan 30H ve 39H arasındaki semboller 0-9 sayılarıdır. Hexadecimal sistemde kullanılan A-F ise 41H ve 46H arasında bulunmaktadır. Ayrıca a-f küçük harfleri de 61H ve 66H arasında bulunmaktadır.

Binaryden ASCII ye dönüşüm işleminde önce R0 alınan binary değerin sayı mı harf mi olduğuna bakılmakta ardından buna göre sayı ise o sayıya 30H, harf ise 61H eklenmektedir. Bu kontrol işlemi o akümülatördeki değerden 10 çıkarılmaktadır. Eğer sayı ise çıkarma işleminde çıkarılan sayı çıkan sayıdan büyük olacaktır ve elde bayrağı set edilmeyecektir, çıkarılan sayı çıkan sayıdan küçük ise elde bayrağı set edilerek o değere 41H değeri eklenecektir. Burada A-F arasındaki harf değerlerine 41H eklenerek ASCII tablosundaki büyük harfler kullanılmıştır. Gerektiğinde bu değerlere 61H yerine 41H eklenerek harflerin ASCII tablosundaki küçük harf karşılıkları da bulunabilir.

ASCII den binary e dönüşümde ise yukarda anlatılanların tam tersi yapılmaktadır. Ancak burada ek olarak harf ASCII karakterlerin büyük ve küçük olması durumları için ek

bir kontrol daha yapılmaktadır. İki alt programda da girilen değer R0 a çıkan sonuç ise akümülatörde saklanmaktadır.

```
BINARY2ASCII:
      MOV A,R0              ;R0  daki değeri akümülatöre yükle
      CLR C                 ;sonraki işlemler için elde bayrağını
                            ;temizle
      SUBB A,#0AH           ;sayı mı harf mi olduğunu bulmak için
                            ;10 çıkar

      JNC SAYI              ;elde set edilmemişse sayı işlemine geç
      ADD A,#41H            ;elde set edilmişse 41H ekle
      JMP SON               ;işlemi bitir

SAYI:                       ;elde set edilmemişse sayı işlemi
      MOV A,R0              ;değeri akümülatöre yükle
      ADD A,#30H            ;sayı için 30H ekle

      SON:                  ;bitir
      RET                   ;alt programdan geri dön

ASCII2BINARY:
      MOV A,R0              ;R0  daki değeri akümülatöre yükle
      CLR C                 ;sonraki işlemler için elde bayrağını
                            ;temizle
      SUBB A,#41H           ;sayı mı büyük harf mi olduğunu bulmak
                            ;için 41H çıkar
      JNC SAYI2             ;elde set edilmemişse sayı2 işlemine geç
      ADD A,#0AH            ;büyük harf için çıkan değere 10 ekle
      JMP SON2              ;işlemi bitir

      MOV A,R0              ;R0  daki değeri akümülatöre yükle
      SUBB A,#61H           ;sayı mı küçük harf mi olduğunu bulmak
                            ;için  61H çıkar
      ADD A,#0AH            ;küçük harf için çıkan değere 10 ekle
      JNC SAYI2             ;işlemi bitir

SAYI2:                      ;elde set edilmişse sayı işlemi
      MOV A,R0              ;R0  daki değeri akümülatöre yükle
      SUBB A,#30H           ;sayı değeri için 30H ekle

SON2:                       ;bitir
      RET                   ;geri dön
```

CPE 323 MSP430 INSTRUCTION SET ARCHITECTURE

Aleksandar Milenkovic

The MSP430 is a 16-bit, byte-addressable, RISC-like architecture. The main characteristics of the instruction set architecture are as follows.

Registers

The MSP430 has a register file with 16 registers (R0-R15) that are all visible to programmers. Register R0 is reserved for the program counter (PC), register R1 serves as the stack pointer, and register R2 serves as the status register. Register R3 can be used for constant generation, while the remaining registers R4-R15 serve as general-purpose registers.

A relatively large number of general-purpose registers compared to other microcontrollers, allows the majority of program computation to take place on operands in general-purpose registers, rather than operands in main memory. This helps improve performance and reduce code size.

A block diagram of the processor core is shown in Figure 1. Register-register operations are performed in a single clock cycle.

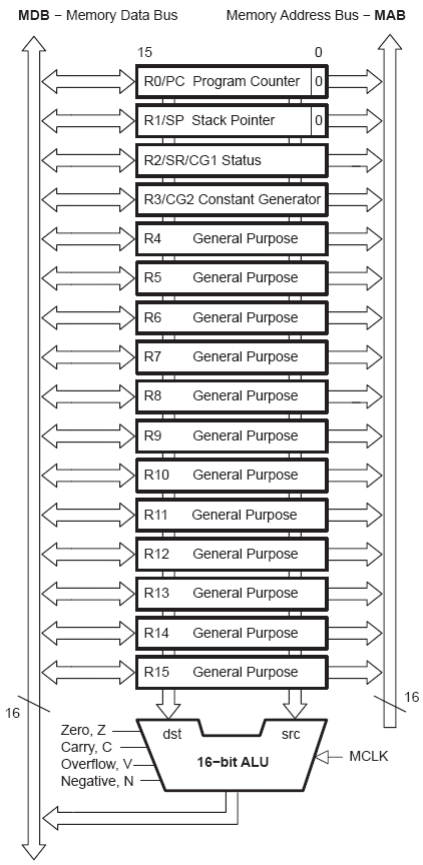


Figure 1. MSP430 CPU Block Diagram.

Program counter (PC/R0). PC always points to the next instruction to be executed. MS430 instructions can be encoded with 2 bytes, 4 bytes, or 6 bytes depending on addressing modes used for source (src) and source/destination (src/dest) operands. Hence, the instructions have always an even number of bytes (they are word-aligned), so the least significant bit of the PC is always zero.

The PC can be addressed by all instructions. Let us consider several examples:

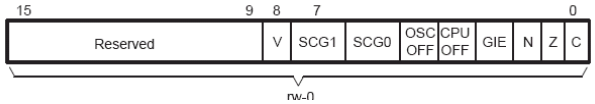
```
MOV #LABEL,PC ; Branch to address LABEL
MOV LABEL,PC ; Branch to address contained in LABEL
MOV @R14,PC ; Branch indirect to address in R14
```

Stack pointer (SP/R1). The program stack is a dynamic LIFO (Last-In-First-Out) structure allocated in RAM memory. The stack is used to store the return addresses of subroutine calls and interrupts, as well as the storage for local data and passing parameters. The MSP430 architecture assumes the following stack convention: the SP points to the last full location on the top of the stack, and the stack grows toward lower addresses in memory. The stack is also word-aligned, so the LSB bit of the SP is always 0.

Two main stack operations are PUSH (the SP is first decremented by 2, and then the operand is stored in memory at the location addressed by the SP), and POP (the content from the top of the stack is retrieved; the SP is incremented by 2).

[[Illustrate PUSH and POP operations on the stack.]]

Status register (SR/R2). The status register keeps the content of arithmetic flags (C, V, N, Z), as well as some control bits such as SCG1, SCG0, OSCOFF, CPUOFF, and GIE. The exact format of the status register and the meaning of the individual bits is show in Figure 2.



Bit	Description	
V	Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-variable range.	
	`ADD(.B),ADDC(.B)`	Set when: Positive + Positive = Negative Negative + Negative = Positive, otherwise reset
	`SUB(.B),SUBC(.B),CMP(.B)`	Set when: Positive − Negative = Negative Negative − Positive = Positive, otherwise reset
SCG1	System clock generator 1. This bit, when set, turns off the SMCLK.	
SCG0	System clock generator 0. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.	
OSCOFF	Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not use for MCLK or SMCLK	
CPUOFF	CPU off. This bit, when set, turns off the CPU.	
GIE	General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.	
N	Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative.	
	Word operation:	N is set to the value of bit 15 of the result
	Byte operation:	N is set to the value of bit 7 of the result
Z	Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.	
C	Carry bit. This bit is set when the result of a byte or word operation produced a carry and cleared when no carry occurred.	

Figure 2. Status register format (top) and bits description (bottom).

Constant generator (R2-R3). Profiling common programs for constants shows that just a few constants, such as 0, +1, +2, +4, +8, -1, are responsible for majority of program constants. However, to encode such a constant we will need 16 bits in our instruction. In order to reduce the number of bits spent for encoding frequently used constants, a trick called constant generation is used. By specifying dedicated registers R2 and R3 in combination with certain addressing modes, we tell hardware to generate certain constants. This results in shorter instructions (we need less bits to encode such an instruction). **Error! Reference source not found.** describes values of constant generators.

Register	As	Constant	Remarks
R2	00	– – – – –	Register mode
R2	01	(0)	Absolute address mode
R2	10	00004h	+4, bit processing
R2	11	00008h	+8, bit processing
R3	00	00000h	0, word processing
R3	01	00001h	+1
R3	10	00002h	+2, bit processing
R3	11	0FFFFh	−1, word processing

Figure 3. Constant generation.

An example: Let's say you want to clear a word in memory at the address *dst*. To do this, a MOVE instruction could be used:

MOVE #0, *dst*

This instruction would have 3 words: the first contains the *opcode* and addressing mode specifiers. The second word keeps the constant zero, and the third word contains the address of the memory location. Alternatively, the instruction

MOVE R3, dst

performs the same task, but we need only 2 words to encode it.

General-purpose registers (R4-R15). These registers can be used to store temporary data values, addresses of memory locations, or index values, and can be accessed with BYTE or WORD instructions.

Let us consider a register-byte operation using the following instruction:

ADD.B R5, 0(R6)

This instruction specifies that a source operand is the register R5, and src/dest operand is in memory at the address (R6+0). The suffix .B indicates that the operation should be performed on byte-size operands. Thus, a lower byte from the register R5, 0x8F, is added to the byte from the memory location Mem(0x0203)=0x12, and the result is written back, so the new value of Mem(0x0203)=0xA1. The content of the register R5 is intact.

Let us now consider a byte-register operation using the following instruction:

ADD.B @R6, R5.

This instruction specifies a source operand in memory at the address contained in R6, and the destination operand is in the register R5. A suffix .B is used to indicate that the operation uses byte-sized operands. A suffix .W indicates that operations are performed on word-sized operands and is default (i.e., by omitting .W we imply word-sized operands). As shown below, a byte value Mem(0x0223)=0x5F is added to the lower byte of R5, 0x02. The result of 0x61 is zero extended to whole word, and the result is written back to register R6. So, the upper byte is always cleared in case of byte-register operations.

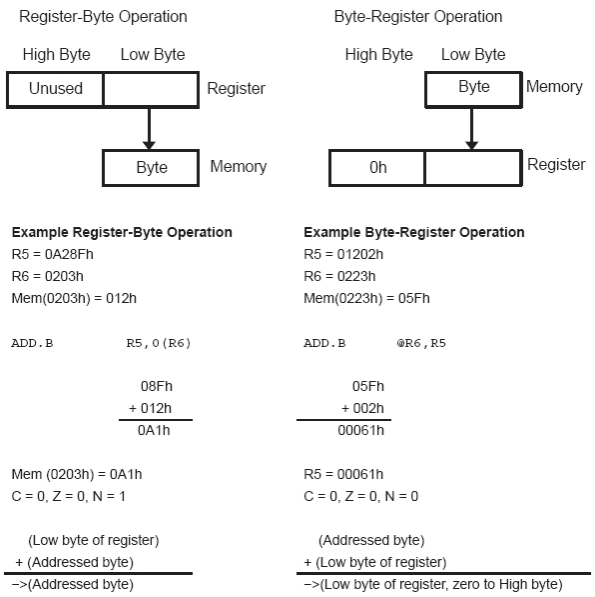


Figure 4. Register-byte (left) and byte-register (right) operations.

Addressing Modes

The MSP430 architecture supports a relatively rich set of addressing modes. Seven of addressing modes can be used to specify a source operand in any location in memory (Figure 5), and the first four of these can be used to specify the source/destination operand. Figure 5 also illustrates the syntax and give a short description of the addressing modes. The addressing modes are encoded using As and Ad address specifiers in the instruction word, and the first column shows how they are encoded.

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register contents are operand
01/1	Indexed mode	X(Rn)	(Rn + X) points to the operand. X is stored in the next word.
01/1	Symbolic mode	ADDR	(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/–	Indirect register mode	@Rn	Rn is used as a pointer to the operand.
11/–	Indirect autoincrement	@Rn+	Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/–	Immediate mode	#N	The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.

Figure 5. Addressing Modes.

Register mode. The fastest and shortest mode is used to specify operands in registers. The address field specifies the register number (4 bits).

Indexed mode. The operand is located in memory and its address is calculated as a sum of the specified address register and the displacement X, which is specified in the next instruction word. The effective address of the operand is ea, ea=Rn+X.

Symbolic mode. This addressing mode can be considered as a subset of the indexed mode. The only difference is that the address register is the PC, and thus ea=PC+X.

Absolute mode. The instruction specifies the absolute (or direct) address of the operand in memory. The instruction includes a word that specifies this address.

Indirect register mode. It can be used only for source operands, and the instruction specifies the address register Rn, and the ea=Rn.

Indirect autoincrement. The effective address is the content of the specified address register Rn, but the content of the register is incremented afterwards by +1 for byte-size operations and by +2 for word-size operations.

Immediate mode. The instruction specifies the immediate constant that is operand, and is encoded directly in the instruction.

One should notice a smart encoding of the addressing modes. Only a 2-bit address specifier As is sufficient in encoding 7 addressing modes. How does it work? For example, please note that the absolute addressing mode is encoded in the same way as the indexed and the symbolic modes, As=01. However, the absolute mode specifies the SR register as the address register. It is never used in the indexed mode as an address register, so this combination indicates the absolute addressing. Next, the immediate mode uses the same As=11 as the autoincrement mode. It is distinguished from the autoincrement mode because the specified register is the PC, which is never used in the autoincrement mode. Similarly, we can explain how only a single bit *Ad* suffices in distinguishing 4 addressing modes for the destination operand.

[[Examples]]

Instruction Set

The MSP430 instruction set consists of 27 core instructions and 24 emulated instructions. The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves, instead they are replaced automatically by the assembler with an equivalent core instruction. There is no code or performance penalty for using emulated instruction.
There are three core-instruction formats:

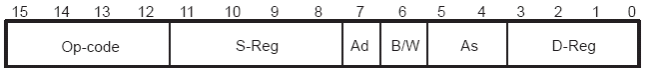
- Double-operand
- Single-operand
- Jump

All single-operand and double-operand instructions can be byte or word instructions by using .B or .W extensions. Byte instructions are used to access byte data or byte peripherals. Word instructions are used to access word data or word peripherals. If no extension is used, the instruction is a word instruction.

The source and destination of an instruction are defined by the following fields:

- src - The source operand defined by As and S-reg
  - As - The addressing bits responsible for the addressing mode used for the source (src)
  - S-reg The working register used for the source (src)
- dst - The destination operand defined by Ad and D-reg
  - Ad - The addressing bits responsible for the addressing mode used for the destination (dst)
  - D-reg - The working register used for the destination (dst)
- B/W Byte or word operation:
  - 0: word operation
  - 1: byte operation

Figure 6 shows the double-operand instruction format and the list of all double-operand core instructions.

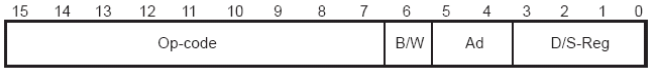


Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
MOV(.B)	src,dst	src → dst	–	–	–	–
ADD(.B)	src,dst	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	src + dst + C → dst	*	*	*	*
SUB(.B)	src,dst	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	dst + .not.src + C → dst	*	*	*	*
CMP(.B)	src,dst	dst – src	*	*	*	*
DADD(.B)	src,dst	src + dst + C → dst (decimally)	*	*	*	*
BIT(.B)	src,dst	src .and. dst	0	*	*	*
BIC(.B)	src,dst	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	src .or. dst → dst	–	–	–	–
XOR(.B)	src,dst	src .xor. dst → dst	*	*	*	*
AND(.B)	src,dst	src .and. dst → dst	0	*	*	*

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

Figure 6. Double-operand instruction format (top) and instruction table (bottom).

Figure 7shows the single-operand instruction format and the list of all single-operand core instructions.

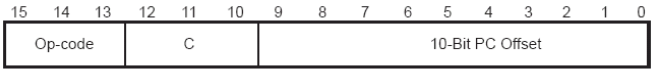


Mnemonic	S-Reg, D-Reg	Operation	Status Bits			
			V	N	Z	C
RRC(.B)	dst	C → MSB →.....LSB → C	*	*	*	*
RRA(.B)	dst	MSB → MSB →....LSB → C	0	*	*	*
PUSH(.B)	src	SP – 2 → SP, src → @SP	–	–	–	–
SWPB	dst	Swap bytes	–	–	–	–
CALL	dst	SP – 2 → SP, PC+2 → @SP	–	–	–	–
		dst → PC				
RETI		TOS → SR, SP + 2 → SP	*	*	*	*
		TOS → PC,SP + 2 → SP				
SXT	dst	Bit 7 → Bit 8.....Bit 15	0	*	*	*

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

Figure 7. Single-operand instruction format (top) and instruction table (bottom).

Figure 7shows the jump instruction format and the list of all jump core instructions. Conditional jumps support program branching relative to the PC and do not affect the status bits. The possible jump range is from −511 to +512 words relative to the PC value at the jump instruction. The 10-bit program-counter offset is treated as a signed 10-bit value that is doubled and added to the program counter:
PCnew = PCold + 2 + PCoffset × 2



Mnemonic	S-Reg, D-Reg	Operation
JEQ/JZ	Label	Jump to label if zero bit is set
JNE/JNZ	Label	Jump to label if zero bit is reset
JC	Label	Jump to label if carry bit is set
JNC	Label	Jump to label if carry bit is reset
JN	Label	Jump to label if negative bit is set
JGE	Label	Jump to label if (N .XOR. V) = 0
JL	Label	Jump to label if (N .XOR. V) = 1
JMP	Label	Jump to label unconditionally

Figure 8. Jump instruction format (top) and instruction table (bottom).

Figure 9 shows a complete list of the MSP430 core and emulated instructions.

Mnemonic		Description		V	N	Z	C
ADC(.B)†	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR†	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Clear destination	0 → dst	–	–	–	–
CLRC†		Clear C	0 → C	–	–	–	0
CLRN†		Clear N	0 → N	–	0	–	–
CLRZ†		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)†	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)†	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)†	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT†		Disable interrupts	0 → GIE	–	–	–	–
EINT†		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)†	dst	Increment destination	dst +1 → dst	*	*	*	*
INCD(.B)†	dst	Double-increment destination	dst+2 → dst	*	*	*	*
INV(.B)†	dst	Invert destination	.not.dst → dst	*	*	*	*
JC/JHS	label	Jump if C set/Jump if higher or same		–	–	–	–
JEQ/JZ	label	Jump if equal/Jump if Z set		–	–	–	–
JGE	label	Jump if greater or equal		–	–	–	–
JL	label	Jump if less		–	–	–	–
JMP	label	Jump	PC + 2 x offset → PC	–	–	–	–
JN	label	Jump if N set		–	–	–	–
JNC/JLO	label	Jump if C not set/Jump if lower		–	–	–	–
JNE/JNZ	label	Jump if not equal/Jump if Z not set		–	–	–	–
MOV(.B)	src,dst	Move source to destination	src → dst	–	–	–	–
NOP†		No operation		–	–	–	–
POP(.B)†	dst	Pop item from stack to destination	@SP → dst, SP+2 → SP	–	–	–	–
PUSH(.B)	src	Push source onto stack	SP – 2 → SP, src → @SP	–	–	–	–
RET†		Return from subroutine	@SP → PC, SP + 2 → SP	–	–	–	–
RETI		Return from interrupt		*	*	*	*
RLA(.B)†	dst	Rotate left arithmetically		*	*	*	*
RLC(.B)†	dst	Rotate left through C		*	*	*	*
RRA(.B)	dst	Rotate right arithmetically		0	*	*	*
RRC(.B)	dst	Rotate right through C		*	*	*	*
SBC(.B)†	dst	Subtract not(C) from destination	dst + 0FFFFh + C → dst	*	*	*	*
SETC†		Set C	1 → C	–	–	–	1
SETN†		Set N	1 → N	–	1	–	–
SETZ†		Set Z	1 → C	–	–	1	–
SUB(.B)	src,dst	Subtract source from destination	dst + .not.src + 1 → dst	*	*	*	*
SUBC(.B)	src,dst	Subtract source and not(C) from dst.	dst + .not.src + C → dst	*	*	*	*
SWPB	dst	Swap bytes		–	–	–	–
SXT	dst	Extend sign		0	*	*	*
TST(.B)†	dst	Test destination	dst + 0FFFFh + 1	0	*	*	1
XOR(.B)	src,dst	Exclusive OR source and destination	src .xor. dst → dst	*	*	*	*

† Emulated Instruction

Figure 9. The complete MSP430 Instruction Set (core + emulated instructions).