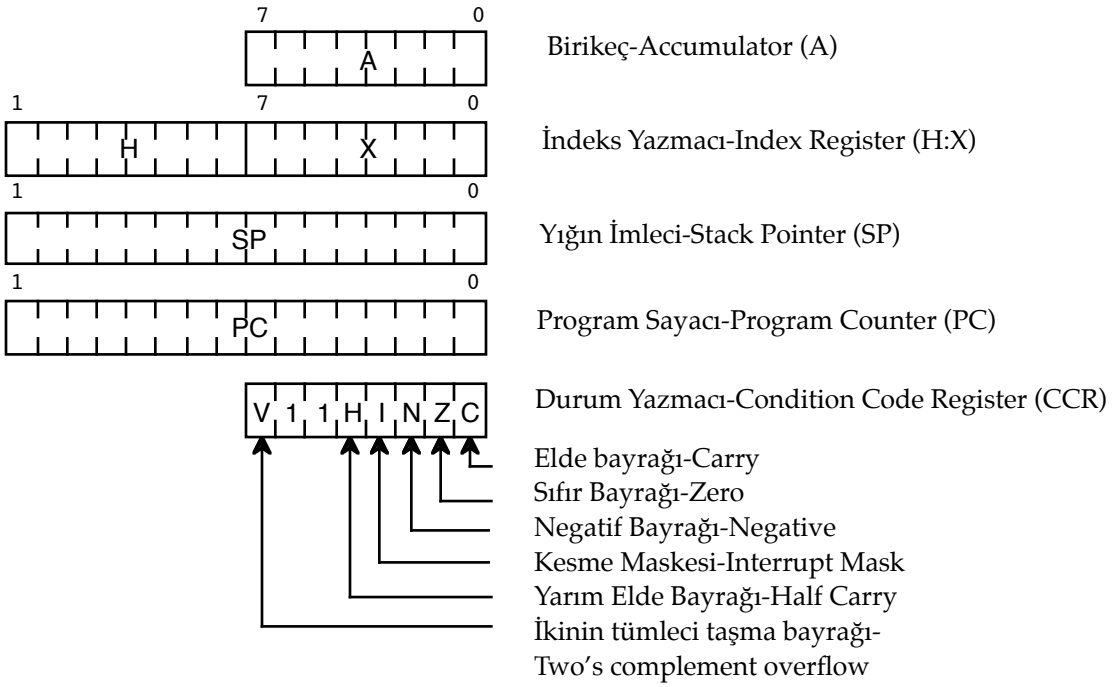
Komutlar ve Adresleme Kipleri

2-1 Programlama Modeli

Komutların ve adresleme kiplerinin ayrıntılarına girmeden HC08 ailesi mikroişlemcilerin programlama modelini tanıyalım. HC08 ailesi bazı 16-bit genişlemelere sahip 8-bitlik mikroişlemcilerdir. Programcının erişimine açık yazmaçlar (register) Şekil 2-1 de gösterilmiştir. Burada uzun olanlar 16 bit, kısalar ise 8 bit bilgi tutabilirler. Bu yazmaçların kısa bir tarifini ve kullanma yerlerini özetliyelim.



Şekil 2-1. HC08 ailesi programlama modeli

Birikeç (Accumulator)

Şekil 2-1 gösterilen Birikeç A genel amaçlı 8-bitlik bir yazmaçtır. Merkezi işlem birimi (CPU) birikeci komut verisi tutmaya ve aritmetik/mantık tipi işlemlerin girdi ve çıkış verilerini saklamaya kullanır.

İndeks Yazmacı veya İmleci (Index Register H:X)

16-bitlik İndeks Yazmacı ile 64 Kbayt bellek aralığının tümünde imleçli erişim yapılabilir. Bu yazmaç H ve X yarılarının birleştirilmesinden oluşur. HC08 ailesi öncesi HC05 ailesinde yalnız 8-bitlik X yazmacı vardı. HC05 ailesi yazılımların HC08 ailesine taşınabilinmesini kolaylaştırmak amacıyla bu mimariye karar verilmiştir.

HC05 kod eşdeğerliliği için donanım sıfırlama (hardware reset) sonrası bu yazmacın üst kısmı (H) sıfırlanır. İndeks Yazmacı tablo ve dizi elemanları erişimi türü işlemlerde imleç görevini yerine getirir.

Yığın İmleci (Stack Pointer SP)

Yığın imleci (SP) yığın üzerinde bir sonraki (kullanılmamış bellek gözü) belleğin adresini tutan 16-bitlik bir yazmaçtır. Yığın geçici verilerin sıralı erişimli bir şekilde saklanması amacıyla ayrılmış bir bellek bölümüdür. Geçici veri, yığına itilerek (push işlemiyle) saklanır. Burada yığın imleci otomatik bir adres yaratıcısı olarak çalışır ve her veri yazma (yığına itme) işleminden sonra bir azalarak bir sonraki boş veya diğer anlamıyla kullanılmamış yere işaret eder. Yığından okuma (pull işlemi) sırasında yazma işleminin tam tersi sıralısı yapılır; önce yığın imleci otomatik olarak bir artar , sonra yığın imlecinin gösterdiği (adreslediği) bellek gözü okunur.

Birçok komutta yığın imleci index yazmacı gibi kullanılabilir. Bu şekilde Yığın İmleçli İndeksli adresleme gerçekleştirilir. Bu tip adresleme yığında dizi şeklinde sıralanmış geçici verilerin işlenmesinde büyük programlama kolaylığı sağlar.

Donanım sıfırlama (hardware reset) sonrası HC05 ailesi kod eşdeğerliliği amacıyla yığın imleci $00FF ‘e eşitlenir. Çok dikkat edilmesi gereken bir husus, RSP komutunun yığın imlecinin alt yarısına $FF yüklemesi ve üst yarısını değiştirmemesidir.

Program Sayacı (Program Counter PC)

Program sayacı (PC) 16-bitlik imleç işlemli bir yazmaçtır ve bir sonraki komuta veya komut verisine işaret eder. Normal şartlar altında program sayacı her bellekten komut okuma veya komut veri okuma işlemi sırasında otomatik olarak bir artar. Atla (Jump), Dallan (Branch), Altyordam çağır (subroutine call) komutları ve kesme (interrupt) işlemleri, program sayacına bir sonraki değerden başka değer yükler. Donanım sıfırlama sırasında program sayacı $FFFE ve $FFFF adresli bellek gözlerinde bulunan 16-bitlik vektör adresiyle yüklenir. Bu vektör donanım sıfırlama sonrası ana programın başlangıç adresidir.

Durum Yazmacı (Condition Code Register CCR)

8-bitlik Durum Yazmacı (CCR) kesme maskeleme biti ve beş işlem sonucu bayrağından (flag) oluşur. Beşinci ve altıncı bit kullanılmamaktadır ve her zaman bir konumundadır. Bu bitlerin işlevlerini kısaca tanıyalım:

V - İkinin Tümlevi Taşma Bayrağı (Overflow Flag)

Bu bit, ikili gösterim sisteminde ikinin tümlevi taşma sırasında bir konumuna gelir. Taşma bayrağı V, işaretli aritmetik işlemler için gereklidir.

H - Yarım Elde Bayrağı (Half-Carry Flag)

Eldesiz toplama (ADD) ve eldeli toplama (ADC) işlemleri sırasında birikecin (accumulator) üçüncü bitinden dördüncü bitine bir elde işlemi oluştuğunda bir'e eşitlenir (bayrak kalkar). Bu yarım elde ikili kodlanmış onlu sayı sistemi sayılar için gereklidir. Mikroişlemciler normalde sayıların ikili (binary) gösterimde olduğunu var sayarlar. Birikeci onlu düzelt (DAA) komutu H ve C bayraklarını kullanarak toplamı yine ikili kodlanmış onlu sayı sistemine çevirir.

I - Kesme Maskeleme (Interrupt Mask)

Kesme Maskeleme bir konumunda ise, bütün kesmeler etkisizdir. Kesmeleri etkinleştirmek için bu bit sıfırlanmalıdır. Bir kesme sırasında bu bit otomatik olarak bir konumuna gelir.

N - Negatif Bayrağı (Negative Flag)

Bir aritmetik, mantık veya veri işleme komutu sonucunda netice veri negatif ise (verinin en önemli biti bir ise) bu bayrak kalkar (N biti bir olur).

Z - Sıfır Bayrağı

Bir aritmetik, mantık veya veri işleme komutu sonucunda netice veri sıfıra eşit ise bu bayrak kalkar.

C - Elde Bayrağı (Carry/Borrow Flag)

Bir toplama veya çıkarma işlemi sonucu netice sekiz bite sığmıyorsa C biti bir olur. Kaydırma/yuvarlama (shift/rotate), bit sına ve dallan işlemleri de bu biti değiştirir.

Bayrakların ayrıntılı kullanımı ileri bölümlerde programlama örnekleri sırasında kapsanacaktır..

2-2 Komut (Instruction)

Şimdi bir merkezi işlem biriminin (CPU) işlemlerinden biri olan komut işleme kavramını araştıracağız. Komut durağan olarak bellekte bir bitler dizisi veya bir program satırı veya dinamik olarak denetleme kısmının bir dizi işlemi olarak açıklanabilir. Denetim ünitesinin ne yapacağı arka arkaya gelen bellek gözlerinde saklı olan komut dizileriyle belirlenir. CPU'nun denetleyicisi programın koşulması sırasında her bir komut için tekrar tekrar şu işlemleri yapar:

1. Bir sonraki komutu bellekten oku.
2. Okunan komutu çöz.
3. Çözülmüş komutu adım adım yap.

İleride de göreceğimiz gibi HC08 ailesi mikrodenetleyicilerde bellekten bir komutun okunması bazı durumlarda ilave bir veya birkaç baytın daha okunmasını gerektirebilir. Komutun koşulması sırasında buna ilaveten bir veya birden fazla bayt bellekten okunabilir veya yazılabilir. Komutu bellekten okuma getirme işlemi (fetch cycle) olarak adlandırılır. Bu işlem birden fazla bellek okuma işleminden oluşabilir. İlk okunan bayt işlem kodudur ve bu kodun çözümü sonucu denetleyici bu işlem için ilave baytların okunmasının gerekip gerekmediğine karar verir. Bir komutun işlem kodu dışındaki ilave veri baytlarına işlenen (operand) denir. Bir komutun tek veya daha çok bayttan oluşması adresleme kipleri tarafından belirlenir. İşlem verisine ne şekilde erişileciği yine adresleme kipleri ile belirlenir.

Bir mikroişlemcinin komut çeşitleri aşağıdaki gibi gruplandırılabilir:

1. Veri aktarma (Move).
2. Aritmetik (Arithmetic).
3. Mantık (Logical).
4. Denetim (Control).

5. Giriş/Çıkış (Input/Output).

Bu gruplara giren komutları ileride ayrıntılı şekilde işleyeceğiz. Şimdi komutların bellekte nasıl saklandığına ve HC08 ailesi tarafından nasıl işlendiğine bakalım. Örnek olarak veri aktarma gurubu komutlarından birikece veri yüklemeyi ele alalım. Bu komut ile bellekten bir bayt veri birikeç yazmacına yüklenecektir.

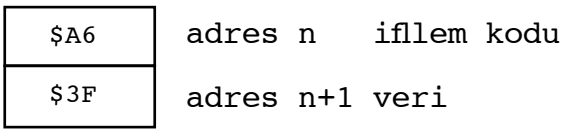
Birikece belli bir sayıyı, örneğin onaltılı tabanda 3F yüklemeyi arzuluyorsak, assembler dilinde komut şöyle yazılmalı idi:

```
LDA     #$3F
```

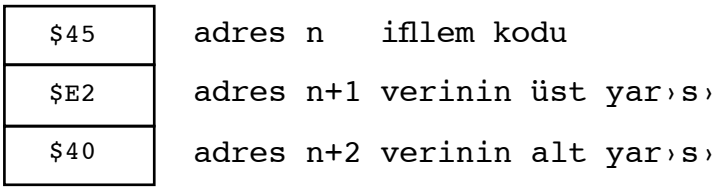
Burada "#" sembolü hemen adreslemeyi, "$" ise sayının onaltılı sayı tabanında olduğunu gösterir. Yazmacın boyuna göre bellekten okunacak veri bir veya iki bayttan oluşur. İndex imlecine (H:X) belli bir sayıyı, örneğin $E240 yüklemek istiyorsak, komut şöyle yazılmalı idi:

```
LDHX    #$E240
```

LDA #$3F komutunun bellekte makine dili karşılığının nasıl saklandığına bakarsak,



görürüz. Bunu LDHX #$E240 komutu için tekrarlarsak bellekte aşağıdakini görürüz:



Burada hemen adresleme kipinde LDA komutunun $A6 ve LDHX komutunun $45 ile makine diline dönüştüğünü görürüz. İndex imlecine yüklenen 16-bitlik $E240 sayısının 8-bitlik bellekte peşpeşe iki gözde ve önce üst yarısı, sonra alt yarısı olarak bulunduğuna dikkat ediniz. Çok baytlı sayıların bayt-genişlikte bellkte önemli (üst) bayt en düşük adreste, önemsiz (alt) bayt en yüksek adreste şekide saklı tutulmasına

büyükte sonlanan (big-endian) format denir. Bütün Freescale mikroişlemciler ve mikrodenetleyiciler bu formatı kullanır. Yukarıdaki ve bundan sonraki şekillerde en küçük adresli bellek gözü en tepede olacaktır.

2-3 Adresleme Kipleri (Addressing Modes)

Bir komut bir işlem kodu (opcode) ve seçmeli olarak bir veriden (operand) oluşur. Veri ya bir kaynak verinin adresi, ya bir neticenin yazılacağı hedef adresinden yada yüklenec verinin kendisinden oluşur. HC08 ailesi mikroişlemciler tek bir komut (MOV) hariç, veri için tek bir etkin adres tanımlayan tek-adres bilgisayardır. Örneğin, komutla 1000 adresli bellek gözündeki veri birikece yükleniyorsa, 1000 etkin adrestir. Adresleme kipi bu etkin adresin nasıl tanımlandığını belirtir ve işlem kodunun bazı bitlerinde saklı tutulur. Gerektiğinde komutun veri kısmında bulunan bilgi adresin belirlenmesine yarar. HC08 ailesi 7 değişik adresleme kipi kullanır.

1) İçsel (Inherent)
2) Hemen (Immediate)
3) Genişletilmiş (Extended)
4) Doğrudan (Direct)
5) Dizinli veya İndeksli (Indexed)
6) Göreceli (Relative)
7) İki adresli (Two-address)

Bu adresleme kiplerini aşağıdaki bölümlerde inceleyeceğiz.

İçsel adresleme (Inherent addressing)

Bazı komutlar kaynak ve hedef adreslerini kendiliğinden içerir. Örneğin CLRA komutu, yani birikecin içeriğini sıfırla komutunda kaynak verisi sıfır, hedef ise birikecin kendisidir.

Hemen adresleme (Immediate addressing)

Bölüm 2-1 de ilk örnek olarak verilen hemen adresleme en basit adresleme kipidir. Bu adresleme kipinde verinin kendisi işlem kodunun hemen bir sonraki bellek gözünde bulunmasından dolayı hemen sıfatı verilmiştir. Bu adresleme kipi bir yazmaca sabit bir sayı yükleme amacıyla kullanılır. İşlemde kullanılan yazmacın boyuna bağlı olarak veri ya 8-bit yada 16-bittir. Sekiz bitlik bir mikroişlemcide 16-bitlik veri ardarda gelen iki bellek gözünde tutulur. Hemen adresleme kipi yalnız bellekten yazmaca veri yüklemede kullanılabilir.

Genişletilmiş adresleme (Extended addressing)

HC08 ailesi mikroişlemcilerin Program Sayacı (PC) 16-bitlik bir yazmaç olduğundan en fazla 2^{16} = 65536 bellek gözüne erişilebilir. Genişletilmiş adreslemede veri adresi 16-bit (iki bayt) ile tanımlanır. Daha önce de belirtildiği gibi 16-bit bilginin önemli yarısı düşük sayılı adreste bulunur. $0180 adres gözündeki veriyi birikece yüklemek istiyorsak, komut

```
LDA     $0180
```

olacaktır. LDA $0180 komutunun makine dilinde bellekte ne şekilde durduğunu araştırdığımızda aşağıdakini görürüz:

$C6	adres n ifllem kodu
$01	adres n+1 adresin üst yar›s›
$80	adres n+2 adresin alt yar›s›

Bu komut CPU tarafından yürütüldüğünde $0180 adresli belleğe erişilir, içeriği okunur, ve birikece yazılır.

Doğrudan adresleme (Direct addressing)

Mikroişlemci kullanımında deneyim veri erişim işleminin çoğunlukla bellek bölgesinin küçük bir kısmına yoğunlaştığını göstermiştir. HC08 ailesinde hem statik hemde dinamik kod randımanını arttırmak amacıyla daha yoğun ve hızlı bir adresleme kipi vardır. Bu adresleme kipinin adı doğrudan adreslemedir ve etkin adresin üst (önemli) yarısının sıfıra eşit olduğunu var sayar. Bu şekilde adres iki bayt yerine yalnız tek bir bayt (alt yarısı) ile belirtilir. Bu adresleme kipine bazen belleğin ilk 256 baytına eriştiği için sıfırıncı sayfa (zero-page) adresleme de denir. Bir önceki örneğe benzer şekilde $0080 adresli bellekteki veriyi birikece yüklemeyi arzularsak komutu şu şekilde yazabiliriz:

```
LDA     $80
```

Burada LDA $0080 yerine LDA $80 yazıldığına dikkat ediniz. LDA $80 komutunun makine dilinde bellekte ne şekilde durduğunu araştırdığımızda aşağıdakini görürüz:

$B6	adres n ifllem kodu
$80	adres n+1 adresin alt yar›s›

Bu örneği genişletilmiş adresle kıyasladığımızda makine dili işlemin üç bayt yerine iki bayta indiğini görürüz. Komutun yürütülmesi esnasında bir bayt daha az okunduğundan işlem bir çevrim saat darbesi daha kısa olmaktadır.

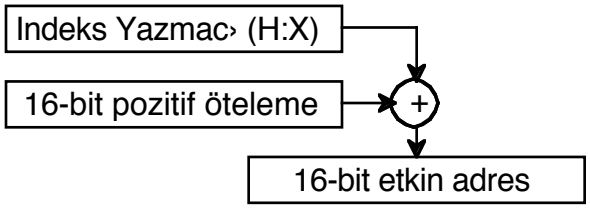
Dizinli veya İndeksli adresleme (Indexed addressing)

Genişletilmiş adreslemenin adres tanımı için sürekli olarak iki bayt gerektirmesi özellikle dizi halinde bellekte duran verilerin erişiminde toplam randımanı düşürdüğü bilgisayar tasarımcılarınca görülmüştür. Denetleyicinin veriye erişim randımanını arttırmak amacıyla veriye dolaylı olarak imleç veya indeks yazmacı yardımıyla erişen tasarım gerçekleştirilmiştir. Bir imleç veya indeks yazmacı aracılığıyla veriye erişim çoğunlukta işlem kodu artı bir bayt ile gerçekleştirilebilir. Böylelikle karakter dizileri, vektörler, başvuru tabloları ve diğer tip veri yapılarına daha az bayt harcıyarak ve daha hızlı erişilebilinmektedir.

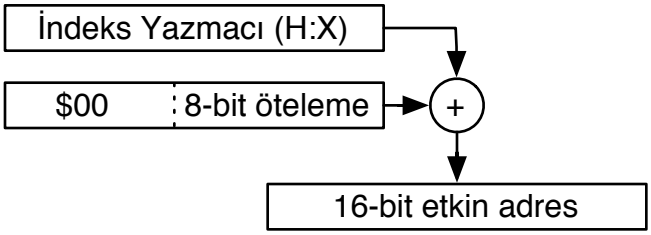
HC08 ailesinde H:X ve SP adında iki imleç vardır. H:X yazmacı indeks yazmacı veya imleci, SP ise yığın imlecidir. İndeks yazmacı indeksli adresleme, yığın imleci ise öncelikle yığına otomatik olan veri yazmaya veya yığından veri okumaya kullanılır. İndeksli adreslemenin 5 ayrı kipi vardır. İlk üçü temel kiplerdir ve diziler, vektörler ve başvuru tabloları tipi veri yapılarına erişimde kullanılır. Böylesi bir veri yapısına erişmek amacıyla indeks yazmacı veri bloğunun başlangıç adresine eşitlenmelidir. Etkin erişim adresi indeks yazmacının (H:X) içeriği ile öteleme (offset) olarak tanımladığımız bir verinin toplamından oluşur. Bu öteleme sıfır, $00 ila $FF arası işaretsiz (pozitif) bir baytlık bir değer, veya $0000 ila $FFFF arası işaretsiz iki baytlık bir sayı olabilir. Örneğin, birikece $E400 de başlayan bir başvuru tablosunun (mesala sinüs tablosu) n'inci elemanını yüklemek istiyorsak komut şu şekilde olmalıdır:

```
LDA     $E400,X
```

Bu işlemi yapmadan önce indeks yazmacının değeri n'e eşit olmalıdır. 16-bit ötelemeli etkin adres aşağıdaki gibi hesaplanır:



8-bitlik ötelemede, öteleme sayısının önüne sıfır ilave ederek 16-bite dönüştürülür, sonra toplama işlemi gerçekleştirilir.



Öteleme değeri sıfırsa, toplama yapılmadan indeks yazmacının içeriği etkin adres olarak kullanılır. Bu durumda işlem hızı maksimum, kod boyu minimumdur. 16-bit öteleme en yavaş ve en çok yer (bayt) tutanıdır ama buna karşın sınırlamasız her yere ulaşabilir. Etkin adres hesabı indeks yazmacının değerini değiştirmez.

Sıfır öteleme ve 8-bitlik ötelemeye bir birikece okuma ve birikeçten yazma komutlarıyla örnek verelim

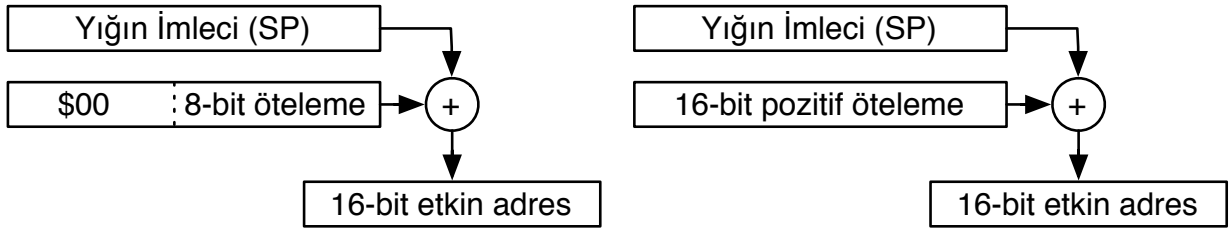
```
LDA     ,X
STA     $80,X
```

ve bellek gözlerinde makine dili karşılıklarını görelim.

$F6	adres n	LDA ,X komutu
$E7	adres n+1	STA $80,X işlem komutu
$80	adres n+2	$80 öteleme

Bu komut ikilisiyle indeks yazmacıxnın gösterdiği bellek gözündeki veri LDA X komutu ile birikece yüklenir, STA $80,X komutu ile ise birikeçteki veri indeks yazmacının gösterdiği bellek gözünden 128 ($80) göz ilerisine yazılır. LDA ,X komutunun yalnız bir bayt yer tuttuğuna dikkat edelim.

İndeks yazmacına ilaveten yığın imleci de indeks yazmacı gibi 8-bit ve 16-bit ötelemeyle etkin adres yaratmada kullanılabilir. Bu şekilde, yığına itilerek saklanmış veri üzerinde bir veri dizisiymiş gibi işlem yapma imkanı doğar.



Yığın imleçli indeksli adreslemeye olarak bir örnek olarak

```
LDA     2,S
```

kodunu verebiliriz. Bellek haritasına baktığımızda 0,SP son yığına itme işlemi sonrası yığın imlecinin işaret ettiği boş veya kullanılmamış göz, 2,SP ise içine daha önce burada göstermediğimiz program satırlarında $E2 itilmiş (yazılmış) bellek gözüdür.

0,SP	boş	adres n
1,SP	$10	adres n+1
2,SP	$E2	adres n+2

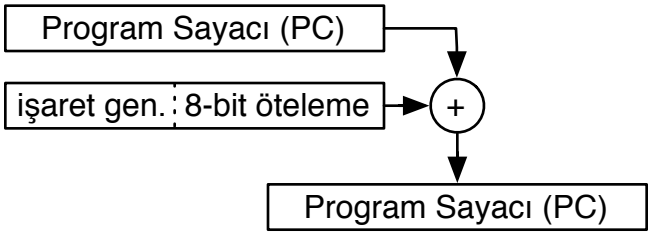
Dördüncü ve beşinci indeksli adresleme “İndeksli ve işlem sonrası indeks arttırımlı” (Indexed with Post Increment) ve “8-bit ötelemeli indeksli ve işlem sonrası indeks arttırımlı” (8-Bit Offset with Post Increment) kipleridir. Bu iki adresleme kipi yalnız CBEQ ve MOV komutlarında kullanılabilir. Bu özel ama güçlü kipleri ileri kısımlarda ayrıntılı bir şekilde işleyeceğiz.

Göreceli adresleme (Relative addressing)

Mikrodenetleyiciler bilgisayarlara benzerler ama mikrodenetleyicilerin içindeki ROM veya EPROMların yarattığı ilginç bir problem bu son programlama kipiyle çözümlenebilir. Birilerin başka birileri tarafından aynı aile mikroişlemci için yazılmış makine dili bir yazılımı satın aldığını varsayalım. Bu yazılımın çalışma adres aralığı alıcının kullandığı mikrodenetleyici tarafından desteklenmiyorsa program kullanılamaz. Bu yazılımın kaynak kodu alınamıyorsa, ve salt adresler kullanıyorsa son kullanıcı yazılımı kendine uygun adres aralığına kaydıramaz. Bu yazılım salt

adresler yerine göreceli adresler kullanmış olsaydı, herhangi bir başlangıç adresinden itibaren çalışabilir ve yukarıdaki problemleri yaratmazdı. Böylesi yazılımlara konumdan bağımsız denir. Program sayacı göreceli, veya kısaca göreceli adresleme bize konumdan bağımsız yazılım üretme imkanı sağlar.

Program sayacı göreceli, veya kısaca göreceli adresleme etkin adres üretmek için program sayacına ikinin tümlevi gösterim bir baytlık sayıyı (işaretli 8-bit sayı) toplar. Aşağıda gösterilen bu işlem sırasında önce 8-bitlik ikinin tümlevi gösterim sayıyı 16-bitlik ikinin tümlevi bir sayıya genişletir, sonra program sayacına toplanır ve toplam program sayacına yüklenir.



HC08 ailesinde göreceli adresleme koşullu ve koşulsuz dallanma ve altyordam çağırma komutlarında kullanılır. Bellekten yazmaca yükleme (LDA ve LDHX) komutlarında bu adresleme kipinin olmaması önemli bir eksikliktir ama bu bölümün sonuna doğru bir yazılım marifetiyle bunu gerçekleştirmek mümkündür.

Aşağıdaki kısa program örneği koşullu dallanma komutlarından birini kullanarak birikeçteki sayıya orantılı bir gecikme sağlar.

```
        LDA     #$10
Döngü   DECA
        BNE     Döngü
```

Bu üç satırlık programda “DECA” birikeci bir azaltır, “BNE Döngü” ise durum yazmacındaki sıfır bayrağını sınar ve bayrak bir’e eşit değilse programın “Döngü” etiketine dallanmasını sağlar. Programın yukarıdaki haliyle 16 kere bu döngüyü yapacağı bellidir, çünkü birikeç ilk satırda $10 = 16 ile yüklenmiştir. Programın saklı olduğu bellek gözlerini incelediğimizde

$A6	$EE21 LDA #$10
$10	
$4A	$EE23 DECA
$26	BNE etiket "Döngü"
$FD	
——	$EE26 Bir sonraki komut

görmekteyiz. Burada $26 BNE komutunun makine kodu, $FD ise eksi üçe eşit dallanma ötelemesidir. $FD ($EE23 - $EE26 = $FD) aritmetik işleminden elde edilmiştir. Burada hatırlanacak önemli husus daha önce belirtildiği gibi program sayacının her zaman yapılan işlemden bir adım ileride olduğudur. $FD bellekten okunurken program sayacı $EE26 olmuştur ve bu yüzden $4A'nın bulunduğu bellek gözü üç adım geridedir.

DECA komutu işlemi sonucu birikeç sıfıra eşitlenmediği durumda durum yazmacında sıfır bayrağı sıfır konumunda olur ve BNE komutuyla program sayacı "Döngü" etiketinin adresine eşitlenir. Buna karşın DECA 'nın neticesinde birikeç sıfıra eşitlendiğinde durum yazmacında sıfır bayrağı bir olur ve BNE komutu sonrası program sayacı değiştirilmez ve bellekteki bir sonraki komut işlenir.

Dallanma komutlarındaki öteleme bir bayt ile sınırlı olduğundan, ikinin tümlevi sayı gösteriminde en büyük sayı +127, en küçük sayı ise -128 dir. Bu yüzden dallanma uzaklığı da bu sayılarla sınırlıdır. Daha büyük uzaklıklar kat edilmek isteniyorsa kodun arasına BRA komutları yerleştirilmelidir.

İki adresli adresleme (Two-address addressing)

Bir veriyi iki değişik bellek gözü arasında taşımak için, veri önce bir LDA komutuyla kaynak bellek gözünden birikece, arkasından da bir STA komutuyla birikeçten hedef bellek gözüne aktarılır. Bu şekilde veri aktarımı genelde birçok mikroişlemcı mikrodenetleyici için standart yöntemdir. Mikrodenetleyicilerin yazılımlarında işlemlerin çoğunluğu ilk 256 bayt adreslerde bulunan giriş/çıkış üniteleri ile bellek arasında veri aktarımıdır. Bu tür işlemler için HC08 ailesi mikrodenetleyicilerde MOV komutu tanımlanmıştır. İleri bölümlerde MOV komutunu ayrıntılarıyla inceleyeceğiz. MOV komutu iki adresli (kaynak + hedef) bir komuttur ve dört değişik adresleme şekli mümkündür:

1. IMM/DIR bir baytı hemen adresten, doğrudan (ilk 256) bir adrese aktarır.

2. DIR/DIR bir baytı doğrudan bir adresten, doğrudan bir adrese aktarır.
3. IX+/DIR H:X 'in gösterdiği bellek gözündeki bir baytı, doğrudan bir adrese aktarır. Bu işlemin ardından H:X bir arttırılır.
4. DIR/IX+ doğrudan adreslenebilir bir bellekteki bir baytlık veriyi H:X 'in gösterdiği bellek gözüne aktarır. Bu işlemin ardından H:X bir arttırılır.

MOV komutuna basit bir örnek verelim:

```
MOV     #$55,$00        $55 hemen verisini $00 adresli bellek gözüne yaz
                        (IMM/DIR) tipi adresleme
```

HC08 ailesi mikrodenetleyicilerde $00 adresinde A giriş/çıkış kapısı (Port A) bulunmaktadır. Yukarıdaki işlemi LDA ve STA komutlarıyla yapmak zorunda kalsaydık, şu komutları kullanmamız gerekirdi:

```
LDA     #$55            $55 hemen sabit verisini birikece yükle
STA     $00             birkeçteki veriyi $00 adresine yaz
```

2-4 Komut Kümesi (The Instruction Set)

Freescale HC08 ailesi mikrodenetleyicilerin 89 tanımlanmış değişik komutu vardır. Bu komutlar 8 ve 16-bit ikili ve onlu aritmetik, mantık, kaydırma/yuvarlama, yükleme, yazma (saklama), veri aktarma, koşullu ve koşulsuz dallanma, altyordam çağırma ve kesme işlemlerinden oluşur.

Freescale HC08 ailesi mikrodenetleyicilerde, işlem kodu tek bayt olan bir komutta işlem kodu komutun ne tip bir işlem olduğunu ve hangi adresleme kipini kullanacağını içerir. İşlem kodu iki bayt olan komutlarda birinci bayt $9E, ikinci bayt komutun ne tip bir işlem olduğunu ve hangi adresleme kipini kullanacağını içerir. Tanımlanmış olan 89 komutun bütün geçerli adresleme kiplerindeki ikili işlem kodları Ek-1 de verilmiştir.

İlerki bölümlerde HC08 ailesinin komutlarını sınıflarına ayırarak örnek ve açıklamalarla inceleyeceğiz.

Bu bölümün sonunda HC08 ailesi Assembler dilinde program yazabilmek için bütün bilgilere sahip olacaksınız. Kolaylıkla 25 satır boyunda küçük programları

yazabileceksiniz. Bu kitabı kullanan derse ilave bir laboratuvarınız olduğunda, yazdığınız küçük programları mikrodenetleyiciye yükleyip koşturabilir, hatalarınızı bulup düzeltebilirsiniz. Bundan sonra laboratuvar çalışmalarını sürdürerek daha uzun programlar yazmak ve denemek size kalmıştır.

2-4-1 Veri Aktarma Komutları (Move Instructions)

Veri aktarma sınıfı komutlar bir veya iki baytlık verileri bellekten yazmaca, yazmaçtan belleğe, yazmaçtan yazmaca, veya bellekten belleğe aktarmaya yarar. Bu sınıftaki en kolay iki komut bellekten yazmaca veri aktar "Load", ve yazmaçtan belleğe veri aktar "Store" dan oluşur. Burada yazmaçlar ya birikeç veya indeks yazmacı H:X veya onun alt yarısı X tir. "Load" tipi komutlar hemen, doğrudan, genişletilmiş ve indeksli adresleme kiplerinde oluşur. "Store" tipi komutlar ise hemen adresleme kipi dışında doğrudan, genişletilmiş ve indeksli adresleme kiplerini kullanabilirler. Hemen adresleme kullanılabilinse idi, mikrodenetleyici çok sakıncalı olan kendi kodunun içine yazma işlemine kalkışırdı.

LDA	Load Accumulator	Birikece veri yükle
LDHX	Load index register (H:X)	İndeks yazmacına (H:X) veri yükle
LDX	Load index reg. low (X)	İndeks yazmacı alt yarısına (X) veri yükle
STA	Store Accumulator	Birikeçteki veriyi bellekte sakla
STHX	Store index register (H:X)	İndeks yazmacındaki (H:X) veriyi bellekte sakla
STX	Store index reg. low (X)	X yazmacındaki veriyi bellekte sakla

Yükle (Load) ve Sakla (Store) komutları kullanılan yazmacın boyuna bağlı olarak 8 veya 16 bitlik veri aktarırlar. HC08 ailesi mikrodenetleyicilerinin veri yolu (data bus) 8 bit olduğundan bellek te 8 bitlik gözlerden oluşur ve 8 bitlik bir veri aktarımı tek bir çevrimde gerçekleşir. Buna karşın 16 bitlik bir veri aktarımında peşpeşe bulunan iki bellek gözüne iki çevrimde ulaşılır. Yükle ve Sakla tipi komutlar için örneklere daha karmaşık komutlarla beraber ileride yer vereceğiz.

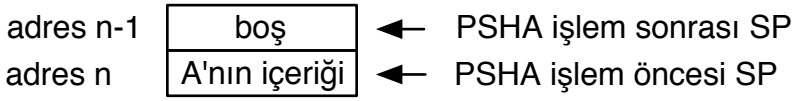
Bellekten yazmaca (ve yazmaçtan belleğe) veri aktarımında yığın imlecinin (SP) kullanılması özel bir durumdur. Yığın imleci (SP) "Push" ve "Pull" komutlarında bir imleç olarak çalışmaktadır. "Push" komutu yazmaçtaki veriyi belleğe, "Pull" ise bellekteki veriyi yazmaca aktarır.

PSHA	Push Accumulator	Birikeçteki veriyi yığına it (push)

PULA	Pull Accumulator	Yığındaki veriyi birikece çek (pull)
PSHH	Push Index Register High	İndeks yazmacı üst yarı veriyi yığına it
PULH	Pull Index Register High	Yığındaki veriyi indeks yazmacı üst yarısına çek
PSHX	Push Index Register Low	İndeks yazmacı alt yarı veriyi yığına it
PULX	Pull Index Register Low	Yığındaki veriyi indeks yazmacı alt yarısına çek

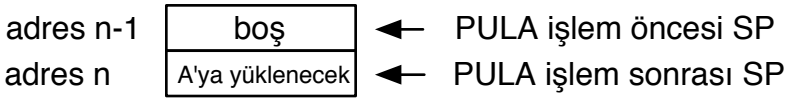
Push komutlarında verinin geçici olarak saklandığı bellek bölgesine yığın adı verilir. Yığın bir mikroişlemci mikrodenetleyici sisteminde bulunan RAM belleğin küçük bir kısmını oluşturur. Yığına geçici süre saklanmak amacıyla itilmiş verinin yazılımdaki bir hatadan dolayı bozulmamasına özen gösterilmelidir.

Bir "Push" işlemi önce yazmaçtaki veriyi yığın imlecinin (SP) gösterdiği bellek gözüne aktarır, sonra yığın imlecinin içeriğini bir azaltır. Şimdi bir PSHA komutu sonrası bellek içeriklerine bir göz atalım:



Açıkca görüldüğü gibi yığın imleci her zaman boş veya kullanılmamış bir bellek gözüne işaret etmektedir.

Bir "Pull" işleminde önce yığın imleci (SP) bir arttırılır, sonra imlecin işaret ettiği bellek gözündeki bilgi okunur ve komutta adı geçen yazmaca yüklenir. Şimdi bir PULA komutu sonrası bellek içeriklerine bir göz atalım:



Yazmaçtan yazmaca veri aktarımı komutları, kaynak ve hedef adresleri komutun kendisiyle belirlendiğinden, içsel adresleme kipini kullanırlar.

TAP	A' daki veriyi CCR' ye aktar	Transfer A to CCR
TPA	CCR' deki veriyi A' ya aktar	Transfer CCR to A
TAX	A' daki veriyi X' e aktar	Transfer A to X
TXA	X' deki veriyi A' ya aktar	Transfer X to A
TSX	SP' deki veriye bir topla ve H:X' e aktar	Transfer (SP)+1 to H:X
TXS	H:X' deki veriden bir çıkar ve SP' ye aktar	Transfer (H:X)-1 to SP

TAP komutu birikecin içeriğini durum yazmacına (CCR) aktarır. Durum yazmacının beşinci ve altıncı bitleri her zaman bire eşit olduklarından, TAP komutu amcak geri kalan bitleri değiştirebilir. TPA komutu ise durum yazmacının bütün bitlerini birikece aktarır. TPA TAP ile birlikte durum yazmacını geçici bir süre için saklamaya yarar. Aşağıdaki komut parçası TPA ile durum yazmacını birikece aktarır, PSHA birikeci yığında saklar, //// araya giren komut satırlarına eşdeğerdir, PULA daha önce yığında saklanan birikeci geri yükler, TAP ile birikeç içeriği durum yazmacına geri aktarılır. Böylece //// satırlarındaki komutların durum yazmacının içeriğini değiştirmesine izin verilmemiş olur.

```
TPA
PSHA
////
PULA
TAP
```

TAX komutu birikeç içeriğini indeks yazmacının alt yarısına (X) aktarılır. TXA komutu ise ters yönde işlem yapar. TAX komutu HC05 ailesinde "birikeç ötelemeli indeksli" adresleme kipi için oluşturulmuştur. HC05 ailesinde indeks yazmacı HC08 ailesine karşın 8-bitliktir. Bu komuta bir örnek vermek için bir başvuru tablosundaki n'ninci veriyi birikece yükleme işlemini gerçekleştirelim. Birikeç (A) içeriği tablodaki kaçıncı elemanın okunacağını belirtmektedir. Başvuru tablosunun $E400 adresinden başladığını varsayarsak program aşağıdaki iki satırdan oluşacaktır.

```
TAX
LDA     $E400,X
```

HC08 ailesi mikroişlemcilerin indeks yazmacı H:X 16-bitlik olduğundan, yukarıdaki komutların doğru çalışması için LDA $E400,X işleminden önce indeks yazmacının üst yarısı bir CLRH komutu ile sıfırlanmalıdır.

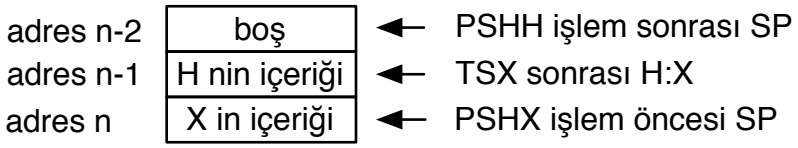
```
TAX
CLRH
LDA     $E400,X
```

TSX ve TXS yığın imleci ile indeks yazmacı arasında 16 bitlik veri transferi komutlarıdır. Burada çok dikkat edilmesi gereken indeks yazmacına transfer edilen değerin yığın imleci değeri artı bir olmasıdır. Bu bir artırarak transfer işlemi, indeks

yazmacının yığına en son itilen veriye sıfır öteleme ile işaret etmesi içindir. İyi programlama kurallarından biri, geçici değişkenlerin ve çalışma alanlarının, normal bellek yerine yığında açılan bir bölgede saklanmasıdır. Böylelikle bu geçici verilerin ve çalışma alanlarındaki bilgilerin başka bir program tarafından bozulması önlenir. Dikkat edilecek husus yığına yapılan itme "Push" ve çekme "Pull" sayılarının birbirine eşit olmasıdır. Aşağıdaki örnekte

```
PSHX
PSHH
TSX
LDA     0,X
PULH
PULX
```

PSHX ve PSHH ile H:X yığına itilir, TSX ile indeks imlecinin yığına en son itilen H'nin saklı olduğu bellek gözüne işaret etmesini sağlar. LDA 0,X komutuyla H 'nin yığında saklı değeri birikece aktarılır. PULH ve PULX ile hem indeks imlecinin orijinal değeri geri yüklenir hem de yığına yapılan itme çekme işlemleriyle dengelenmiş olur. Bu altı satırlık kod ile komut setinde olmayan bir işlem, H yazmacındaki veri birikece aktarılmış olur. Yığındaki saklı bilgiler ve imleçlerin işaret ettiği bellek gözleri aşağıdaki gibidir.



TXS komutu indeks yazmacı içeriği eksi bir değerini yığın imlecine aktarır. Bu işlem çok az kullanılır ve asıl amacı yığın imlecini tanımlamaya yöneliktir. HC08 ailesi mikroişlemciler donanım sıfırlama (hardware reset) işlemiyle yığın imlecine $00FF yükler. Küçük HC08 ailesi mikroişlemcilerde RAM bellek genelde $0080 ile $00FF arasında bulunur. Bu yüzden donanım sıfırlama yığın imlecini doğru olan en üst bellek gözüne işaret edecek şekilde tanımlar. Buna karşın daha çok RAM belleği bulunan HC08 ailesi mikroişlemcilerde, örneğin 68HC908GP32 mikrodenetleyicisinde RAM bellek $0040 ila $023F arasında yer alır. Burada yığın imlecini $023F değerine getirmeyi arzuluyorsak, aşağıdaki kodu kullanmamız gerekir:

```
LDHX    #$0240          68HC908GP32 tepe RAM değeri artı bir
TXS                     H:X - 1 => SP
```

2-3 bölümünde ikili adresleme kısmında MOV komutunu ve uygulanabilen adresleme kiplerini tanıtmıştık. HC08 ailesi mikrodenetleyicilerde giriş/çıkış üniteleri ve belleğin bir kısmı ilk 256 bayt adreslerde bulunduğundan bu üniteler arasında veri aktarımı MOV komutu ile daha kısa yazılımla ve daha hızlı işlemle gerçekleşmektedir. Aynı örneğe bir daha göz atalım:

A6 55	LDA	#$55	Birikece veriyi yükle	2 ~/2 bayt
B7 00	STA	$00	Birikeçteki veriyi hedefe yaz	3 ~/2 bayt

LDA / STA komutları bellekte 4 bayt harcar ve işlem 5 saat darbesi gerektirir. Buna karşı MOV komutuyla aynı işlem 3 bayt ve 4 saat darbesi gerektirir.

6E 00 55	MOV	#$55,$00	$55 verisini $00 adresine yaz	4 ~/3 bayt

LDA / STA komutları birikeci kullandığından, bazı hallerde birikecin LDA #$55 komutu öncesi değeri saklı kalmalıdır. Bu durumda bütün kodu daha da uzatan ve yavaşlatan LDA komutundan önce bir PSHA, STA $00 komutundan sonra da bir PULA komutu gerekirdi.

Bellekte ILK adıyla tanımlanmış bir başlangıç adresinden SON adıyla anılan adrese kadar bulunan verileri peşpeşe HC08 mikroişlemcilerin Port A giriş/çıkış kapısına gönderen bir kısa programı analiz edelim:

	LDHX	#ILK	Bellekteki dizi başına işaret et	3 ~/3 bayt
L1	MOV	X+,$00	veriyi bellekten Port A'ya yolla	4 ~/2 bayt
	CPHX	#SON	imleç SON 'a ulaştı mı ?	3 ~/3 bayt
	BLS	L1	hayırsa, bir sonrakini yolla	3 ~/2 bayt

Önce LDHX #ILK ile indeks imlecinin veri dizisinin başına işaret etmesi sağlanır. MOV X+,$00 komutu indeks imlecinin işaret ettiği bellekteki veriyi $00 adresine (Port A giriş/çıkış kapısı) aktarır ve ardından indeks imlecini bir arttırır. CPHX #SON komutu indeks imlecini SON değeri ile karşılaştırır ve durum yazmacının N, Z, ve C bitlerini güncelleştirir. BLS L1 komutu (Branch if Lower or Same) "daha az veya eşitse dallan" ile CPHX #SON komutunun durum yazmacının (CCR) C ve Z bitleri durumuna göre L1 etiketine dallanır veya bir sonraki komuta devam eder (programın sonu). Aynı programı LDA / STA komutlarıyla yazacak olursak

```
        LDHX    #ILK    Bellekteki dizi başına işaret et    3 ~/3 bayt
L1      LDA     ,X      veriyi bellekten oku                2 ~/1 bayt
        STA     $00     veriyi Port A'ya yaz                3 ~/2 bayt
        AIX     #1      H:X 'i bir arttır                   2 ~/2 bayt
        CPHX    #SON    imleç SON 'a ulaştı mı ?            3 ~/3 bayt
        BLS     L1      hayırsa, bir sonrakini oku/yolla    3 ~/2 bayt
```

hem daha fazla bayt harcar ve daha fazla saat darbesi gerektirir.

2-4-2 Aritmetik Komutları (Arithmetic Instructions)

Bilgisayarlar genellikle sayısal verilerin üzerinde matematiksel işlemler yapmaya veya süreç ve makine denetiminde kullanılırlar. Bu işlemler şimdi işleyeceğimiz aritmetik komutları gerektirir. Bilgisayarlar tasarımlanırken ve programlar yazılırken bellek kullanımı ve işlem hızı dikkate alınır. Bu yüzden temel aritmetik dört işlem (toplama, çıkartma, çarpma, bölme) dışında daha çok kullanılan, bir arttır, bir azalt gibi basit aritmetik işlemlere gerek vardır. İlk tasarımlanan bilgisayar ve mikroişlemcilerde çarpma ve bölme komutları tümleşik devre teknoliojisi imkan sağlamadığından gerçekleştirilememiştir ve örneğin çarpma bir dizi toplama ve sola kaydırma işlemleriyle yazılımla elde edilmekteydi. Bu yüzden çarpma ve bölme işlemleri toplama ve çıkartmaya oranla çok yavaştı. Tümleşik devre teknolojisinin gelişmesine paralel olarak mikroişlemcilerde önce çarpma sonrada bölme komutları yer bulmuştur. Denetim ve veri toplama işlemleri çarpma ve bölme işlemlerini sıkça kullandığından, Freescale HC08 ailesi mikrodenetleyicilerde 8 x 8 çarpma ve 16 / 8 bölme komutları gerçekleştirilmiştir. Aritmetik komutlar hemen, doğrudan, genişletilmiş, indeksli ve içsel adresleme kiplerini kullanır. Önce toplama ve çıkartma komutlarına bir göz atalım.

ADD	Birikece topla (ADD to accumulator)
ADC	Birikece elde ile topla (ADd with Carry to accumulator)
SUB	Biriketen çıkart (SUBtract from accumulator)
SBC	Birikeçten elde ile çıkart (SuBtract with Carry from accumulator)

Görüldüğü gibi eldeli ve eldesiz olarak iki tip toplama ve çıkartma vardır. Eldesiz toplama bellekteki veriyi birikeçtekine toplar. Bu toplama işlemi sırasında netice $FF ten büyükse durum yazmacındaki elde (C) bayrağı kalkar. Toplama komutu durum yazmacındaki V, H, N, Z ve C bitlerini gereken şekilde değiştirir. Eldeli toplamada ise

bellek içeriği ve elde (C) biti birikece toplanır. Bu toplama da V, H, N, Z ve C bitlerini gereken şekilde değiştirir. Toplanacak sayılar 8 bitten büyükse eldeli toplama kullanılmalıdır. Şimdi bellekte bulunan 24-bitlik iki sayının toplama işlemini üç değişik şekilde assembler dilinde gerçekleştirelim. Aşağıda assembler dili kaynak kodu ve üretilen makine dili kodu her üç toplama için verilmiştir.

```
EE00 C6 0182  ADD24  LDA   $0182   1. sayı önemsiz baytı (LSB)
EE03 CB 0185         ADD   $0185   2. sayı önemsiz baytı
EE06 C7 0188         STA   $0188   Toplamın önemsiz baytı
EE09 C6 0181         LDA   $0181   1. sayı orta baytı (NSB)
EE0C C9 0184         ADC   $0184   2. sayı orta baytı
EE0F C7 0187         STA   $0187   Toplamın orta baytı
EE12 C6 0180         LDA   $0180   1. sayı önemli baytı (MSB)
EE15 C9 0183         ADC   $0183   2. sayı önemli baytı
EE18 C7 0186         STA   $0186   Toplamın önemli baytı
*
EE00 45 0180  ADD24  LDHX  #$0180  1. sayı MSB ye işaret et
EE03 E6 02           LDA   2,X     1. sayı LSB yi oku
EE05 EB 05           ADD   5,X     2. sayı LSB ile topla
EE07 E7 08           STA   8,X     Toplam LSB yi yaz
EE09 E6 01           LDA   1,X     1. sayı NSB yi oku
EE0B E9 04           ADC   4,X     2. sayı NSB ile eldeli topla
EE0D E7 07           STA   7,X     Toplam NSB yi yaz
EE0F F6              LDA   ,X      1. sayı MSB yi oku
EE10 E9 03           ADC   3,X     2. sayı MSB ile eldeli topla
EE12 E7 06           STA   6,X     Toplam MSB yi yaz
*
0100 AE 03    ADD24  LDX   #3       döngü sayacını belirle
0102 98              CLC            elde bayrağını sıfırla
0103 D6 017F  ALOOP  LDA   $017F,X  1. sayıyı oku
0106 D9 0182         ADC   $0182,X  2. sayıyla eldeli topla
0109 D7 0185         STA   $0185,X  toplamı yaz
010C 5B F5           DBNZX ALOOP    sayacı (X) bir azalt,
*                                   0 değilse ALOOP'a dallan
```

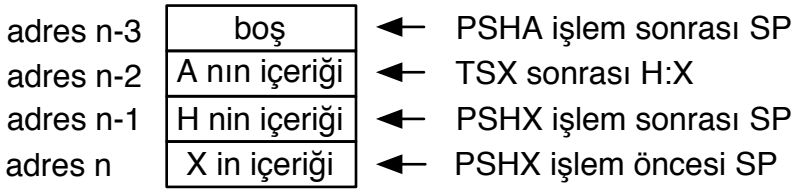
İlk iki örnekte sayıların en önemsiz baytı (LSB) eldesiz toplamayla, orta (NSB) ve en önemli bayt (MSB) eldeli toplamayla toplanmıştır. Burada dikkat edilecek olan okuma "Load" ve yazma "Store" işlemlerinin elde bayrağını (C) değiştirmediğidir. Genişletilmiş adresleme kullanan birinci örnek 27 bayt ve 36 saat darbesi gerektirir. Buna karşı indeksli adreslemeden yararlanan ikinci örnek 20 bayt ve 29 saat darbesi gerektirir. Üçüncü örnek 16-bit ötelemeli indeksli adresleme ve X yazmacını da döngü

sayacı olarak kullanmaktadır. İleride daha ayrıntılı göreceğimiz DBNZX komutu her seferinde X'i bir azaltır ve sıfıra eşitlenip eşitlenmediğini CCR deki Z bayrağına bakarak sınar. Z bayrağı kalkık değilse, yani X sıfıra erişmediyse, komutta verilen etikete bir dallanma yapılır. Z bayrağı kalkıksa dallanma yapılmaz, bir sonraki komut bellekten okunur ve işlenir. Bu kod 14 bayt uzunluğundadır ve diğer iki örnektekinden kısadır fakat 16-bit ötelemeli indeksli adresleme ve DBNZX komutu toplam 47 saat darbesi gerektirerek işlem hızını düşürür. İlk eldeli toplama işleminden önce CLC komutuyla CCRdeki C biti sıfırlanır. Bu döngülü toplama ancak büyük döngü sayılarında verimli olur.

Bir başka örnek ise aşağıda gösterilen yığında bulunan sayılar üzerinde işlem yapılmasıdır:

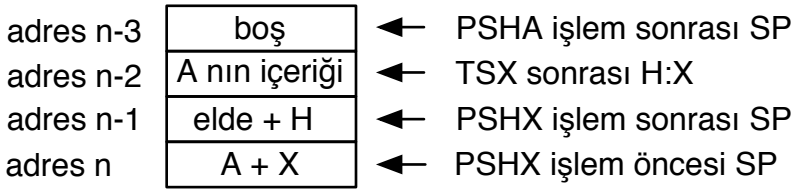
```
PSHX
PSHH
PSHA
TSX
ADD     2,X
STA     2,X
CLRA
ADC     1,X
STA     1,X
PULA
PULH
PULX
```

TSX komutundan sonra yığına bir göz atacak olursak şunları görürüz:



Bu küçük program parçası birikeçteki işaretsiz 8-bitlik bir sayıyı 16-bitlik indeks yazmacına (H:X) toplayarak bir etkin adres hesaplama işlemi yapar. HC08 ailesinde H:X = H:X + A toplama işlemini gerçekleştirebilen bir komut yoktur. Toplama işlemleri bellekteki veriyi birikece topladığından, H:X in içeriği önce bir bellekte saklanmalı, arkasından toplama işlemi gerçekleştirilmelidir. Bu amaçla H:X önce

yığında saklanmalıdır. TSX komutundan sonra indeks yazmacı PSHA işlemiyle birikecin (A) içeriğinin saklı olduğu bellek gözüne işaret eder. Şimdi, ADD 2,X komutuyla A = A + X işlemi, yani birikece yığında bulunan H:X in alt yarısı X'in toplanması yapılır ve toplam STA 2,X ile geri yazılır. Bu işlem sırasında oluşabilecek elde H:X in üst yarısına toplanmalıdır; bunun için önce CLRA ile birikeç sıfırlanır, sonra ADC 1,X ile birikece yığındaki H 'nin içeriği ve elde toplanır. CLRA işlemi durum yazmacındaki C bitini değiştirmez. Toplam yine STA 1,X ile yığına geri yazılır. Bu işlemlerden sonra yığın içeriği şöyledir:



Son olarak PULA, PULH ve PULX işlemleriyle birikeç eski haline, birikeç (A) ile H:X in toplamı H:X 'e geri yüklenir ve yığın programa başlarken ki duruma geri gelir.

Toplama işlemlerini ayrıntılarla anlattıktan sonra çıkartma işlemleri üzerinde aynı mantığı kullandıklarından durulması gerekmez. Çıkartma (Compare) işleminde az farklı olan karşılaştırma işlemi ise ayrı dikkat gerektirir.

Karşılaştırma (Compare) komutları yazmaç içeriğini bellek içeriği ile karşılaştırır. Bu komutlar aslında çıkartma işlemleri yapar ama farkı yazmaca yazmazlar. Buna karşın çıkartma işlemi sonucu durum yazmacındaki V, Z, N, ve C bitlerini güncelleştirirler.

CMP	Birikeç içeriğini bellek içeriğiyle karşılaştır. (A) - (M)
CPHX	H:X yazmacı içeriğini bellek içeriğiyle karşılaştır. (H:X) - (M:M+1)
CPX	X yazmacı içeriğini bellek içeriğiyle karşılaştır. (X) - (M)

CMP ve CPX komutları 8-bit karşılaştırmalardır, CPHX ise 16-bitlik bir karşılaştırmadır. Basit bir örnekte Port A giriş/çıkış kapısı içeriğini bir sınır değerle karşılaştıralım. Şayet Port A değeri sınır değerinden az veya eşit ise Port B ye bir yazalım, aksi halde iki yazalım.

```
LDA     #$7F        Karşılaştırma değerini birikece yükle
CMP     $00         Port A içeriği ile karşılaştır
BHI     High        yüksekse "High" etiketine dallan
```

```
        LDA     #1          değilse A = 1 yap
        BRA     Save        ve yazmaya "Save" dallan
High    LDA     #2          A = 2 yap
Save    STA     $01         birikeci (A) Port B ye yaz
```

Bu program parçası Port A nın giriş, Port B nin ise çıkış kapısı olarak koşullandırılmış olduğunu varsaymaktadır. $0080 ile $009F arası bellek gözlerine sıfır yazmak istiyorsak şu basit program parçasını yazabiliz:

```
CLRM    LDHX    #$0080      H:X 'i birinci bellek gözüne işaret ettir
CLOOP   MOV     #0,X+       belleğe $00 yaz, H:X 'i bir arttır
        CPHX    #$009F      H:X 'i son bellek adresiyle karşılaştır
        BLS     CLOOP       sınır değerden az veya eşitse CLOOP'a dallan
```

Karşılaştırma komutlarının özel kısmını TST komutları oluşturur:

TSTA	Birikeci sına	TeST Accumulator
TSTX	X yazmacını sına	TeST X register
TST	Bellek içeriğini sına	TeST content of memory

Bu komutlar birikeci veya indeks yazmacının alt yarısını (X) veya bellek içeriğini sıfır değeriyle karşılaştırırlar. Bu komutlar CCR deki N ve Z bayraklarını güncelleştirir ve V bitini sıfırlarlar.

Daha önce de değindiğimiz gibi programların bellek kullanımını azaltmak ve işlem hızlarını arttırmak amacıyla bazı basit aritmetik komutlar ilave edilmiştir. Yazılımda, örneğin bir program parçasını kaç kere koşacağımızı sayan bir döngü sayacı yaratmak amacıyla, çoğu zaman bir yazmaca veya bellek gözüne 1 toplamak veya 1 çıkartmak isteriz. Burada ADD #1 komutu yerine INCA ve SUB #1 yerine DECA komutları getirilmiştir. Arttırma ve azaltma komutları

INCA	Birikeci (A) 1 arttır	INCrement A
INCX	X yazmacını 1 arttır	INCrement X
INC	Bellek içeriğini 1 arttır	INCrement content of memory

DECA	Birikeci (A) 1 azalt	DECrement A
DECX	X yazmacını 1 azalt	DECrement X
DEC	Bellek içeriğini 1 azalt	DECrement content of memory

A, X yazmaçlarına ve belleğe 1 toplar veya onlardan 1 çıkarır. Komut ayrıntılarına bakıldığında biraz şaşırtıcı olarak CCR deki C bitinin değişmediği gözükür. INC ve DEC komutları genellikle döngü sayaçlarını değiştirmek amacıyla kullanıldığından, bir aritmetik işlem yerine sayma işlemi olarak kullanılması öngörülmüştür. Bellek gözü içeriği bir sayaç olarak kullanılabildiğinden A ve X kullanılmadan birçok sayaççı kolaylıkla yaratmak mümkündür.

Toplama işlemine benzer iki komut daha vardır:

AIX	H:X e hemen topla	Add immediate to H:X register
AIS	SP ye hemen topla	Add immediate to SP register

Bu komutlar H:X veya SP ye işaretli 8-bitlik bir sayıyı hemen toplarlar. Böylece H:X ve SP nin içeriğinden en fazla 128 çıkarabilir veya 127 toplayabiliriz. Bu komutların CCR deki bitleri değiştirmediğine özellikle dikkat etmek gerekir çünkü bu iki komut imleç değeri değiştirme amacıyla konulmuştur. AIS yığında geçici çalışma alanı açmak ve kapamak amacıyla kullanılabilir. Aşağıdaki program parçası A birikecine A + H:X in işaret ettiği bellek gözündeki veriyi H:X in içeriğini bozmadan yükleme işlemini göstermektedir.

```
PSHX            H:X in değerini yığında sakla
PSHH
PSHX            H:X i bir daha yığında sakla
PSHH
ADD     2,SP    A ya yığındaki X i topla
TAX             Toplamı X e aktar
PULA            Yığındaki H yi A ya yükle
ADC     #0      Toplama işleminden oluşabilen eldeyi A ya kat
PSHA            Değişmiş olabilen H yi yığında sakla
PULH            Bu veriyi H ye yükle
AIS     #1      Yığın imlecini düzelt
LDA     ,X      Veri dizisindeki A'ncı elemanı A ya yükle
PULH            İşlem öncesi H:X i geri yükle
PULX
```

Bu program parçası dizi halinde bulunan verilerin erişiminde, örneğin bir başvurma tablosundaki sıralı değerlere ulaşırken kullanılır ve HC08 ailesinde bulunmayan LDA

A,X komutunu öykünür (emulate). Daha karmaşık mikroişlemcilerde, örneğin HC12 ve 68000 ailesinde bu adresleme kipi vardır ve adına birikeç ötelemeli indeksli adresleme kipi denir.

Denetim, veri toplama ve sinyal işleme algoritmalarının çoğu toplama ve çıkartma işlemlerinin dışında çarpma ve bölme işlemlerini kullanır. Çarpma ve bölme işlemlerinin donanımda gerçekleştirilmeleri aritmetik-mantık biriminde (ALU) çok büyük sayıda kapı devresi gerektirdiğinden tümleşik devrenin maliyetini önemli şekilde etkiler. Donanımda çarpma işlemini gerçekleştiren ilk mikroişlemci/ mikrodenetleyici 1976'da piyasaya çıkan Motorola MC6801 idi. İlerliyen yıllarda, tümleşik devre teknolojisinin gelişmesiyle daha karmaşık mikroişlemci/ mikrodenetleyiciler geliştirildi. MC68000 ve MC68HC11 gibi mikroişlemci/ mikrodenetleyiciler çarpma işlemine ilaveten bölme işlemini de donanımda gerçekleştirdiler. Gelişen teknolojiyle bugün artık HC08 ailesi gibi sade mikrodenetleyicilerde bile çarpma ve bölme mevcuttur. MUL komutu A ile X 8-bitlik yazmaç içeriklerini çarpar ve 16-bitlik sonucu X:A da saklar. Elde bayrağı bu işlem sonucu sıfırlanır.

DIV komutu H ve A birleştirilmiş yazmaçlarında (H:A) bulunan 16-bitlik işaretsiz (pozitif) bir sayıyı X yazmacında bulunan 8-bitlik bir sayıya böler. Bölme sonucu A yazmacında, bölmeden arda kalan H yazmacında saklanır. X yazmacı içeriği değişmez. Sıfıra bölme işleminde veya bölme sonucu 8-bitten büyükse (A>$FF) durum yazmacındaki C biti bir olur ve bölme sonucu ve kalan belirsiz olur.

Mikroişlemcilerde aritmetik bayt randımanı yüksekliğinden ikili veya onaltılı gösterimle yapılmaktadır. Buna karşın insanların eğitimin başından büyük yanlışlık sonucu onlu gösterime alıştırılmıştır. Birikeci onlu gösterime uyarla (DAA) komutu ile ikili kodlanmış onlu sayıları toplama işleminde kullanılmaktadır. Kısaca ikili kodlanmış onlu sistemde bir bayta iki onlu sayı (00 ila 99) sığdırabilinmektedir. Bu sırada 4-bitlik sıradaki 10 ila 15 aralığı kullanılamamaktadır ve neticede bir bayta 0 ila 255 aralığı yerine ancak 0 ila 99 aralığı sığmaktadır. DAA komutu bir toplama sonucu durum yazmacındaki H ve C bitlerinin değerine göre toplam neticesini ikili kodlanmış onlu gösterime çevirir. Komutun nasıl işlediğine örnek vermek için birikecin içeriğinin $46 ve $90 adresli belleğin içeriğinin $27 ye eşit olduğunu kabul edelim.

ADD $90

komutu sonunda birikecin (A) içeriği $6D ye eşitlenir. Halbuki toplamaya giren sayıların ikili kodlanmış onlu olmaları durumunda toplam sonucu $73 olmalıydı.

```
ADD $0140
DAA
```

komut sırası işlendiğinde DAA komutu toplama işlemi sonrası oluşan $6D sayısını $73 'e dönüştürür. Dikkat edilecek husus DAA işleminin yalnız ADD ve ADC komutlarından sonra kullanılabilinmesidir.

Bir sayının işaretinin değiştirilmesi veya bir başka deyimle sayının sıfırdan çıkarılması işlemi çokça yapılan özel bir işlemdir.

NEGA	A 'nın işaretini değiştir	NEGate A
NEGX	X 'in işaretini değiştir	NEGate X
NEG	Bellek içeriğinin işaretini değiştir	NEGate content of memory

komutları A, X veya bellekteki 8-bitlik sayıyı sıfırdan çıkarır ve neticeyi aynı yere geri yazar. Bu işlem C, N, Z, ve V bayraklarını güncelleştirir.

Bir bellek gözünü veya yazmacı boşaltmak veya bir başka değişle oraya sıfır yazmak çok önemli bir işlemdir. Bu amaçla

CLRA	A 'yı sıfırla	CLeaR A
CLRX	X 'i sıfırla	CLeaR X
CLR	Bellek gözünü sıfırla	CLeaR content of memory

komutları tanımlanmıştır. Bu komutlar durum yazmacındaki N ve V bitlerini sıfırlar, Z bitini bire eşitler ve C bitini değiştirmezler.

2-4-3 Mantık İşlemleri (Logic Instructions)

Mantık işlemleri birikeç, yazmaç ve bellek içeriklerinde bir veya birden fazla bitin sıfırlanması, bire eşitlenmesi veya değiştirilmesini sağlar. Mantık komutları çoğunlukla ve en basit haliyle bir mikroişlemci/mikrodenetleyicide belirli bellek gözlerinde bulunan giriş/çıkış kapılarına bağlı donanımı açıp kapamaya veya bağlı bulunan anahtarların durumlarını saptamaya yarar. HC08 ailesinin birleşimsel mantık

komutları

AND	birikeç ve bellek içeriğini “ve” le	AND A
ORA	birikeç ve bellek içeriğini “veya” la	OR A
EOR	birikeç ve bellek içeriğini “dışlayıcı ve” le	Exclusive OR A

ve birikecin veya bellek içeriğinin bir’in tümlevini yani bir olan bitleri sıfır ve sıfır olan bitleri bir yapan

COMA	birikeçi tümle	COMplement A
COMX	X yazmacını tümle	COMplement X
COM	bellek içeriğini tümle	COMplement content of memory

komutlarıdır. Aritmetik işlemlerde SUB ve CMP komutlarının benzeştiği şekilde AND komutunun BIT komutuyla benzeri vardır.

BIT	Birikecin bitlerini test et	BIT test A

BIT komutu AND komutu gibi bellek içeriği ile birikeç içeriğini mantık ve işlemine tabi tutar ama neticeyi birikece yazmaz. Yalnızca durum yazmacında N ve Z bitlerini güncelleştirir ve V bayrağını sıfırlar.

Mikroişlemci uygulamalarında sıkça giriş/çıkış kapılarının belirli bitlerini bir veya sıfır konumuna değiştirmemiz gerekmektedir. Bu işlemleri ORA ve AND komutlarıyla gerçekleştirebiliriz. PortA giriş/çıkış kapısının 3 numaralı bitini sıfırlayan ve PortB giriş/çıkış kapısının sıfırıncı bitini bir konumuna getiren basit komut sıralarını bu örnek için verelim:

LDA	$00	PortA verisini oku	3~/2 bayt
AND	#$FB	bit 3’ü sıfırlamak için $FB ile AND’la	2~/2 bayt
STA	$00	Sonucu PortA ya yaz	3~/2 bayt
LDA	$01	PortB verisini oku	3~/2 bayt
ORA	#$01	bit 0’ı birlemek için $01 ile OR’la	2~/2 bayt
STA	$01	Sonucu PortB ye yaz	3~/2 bayt

Görüldüğü üzere her bir işlem üç satır koddan oluşmaktadır ve bu işlemler toplamda 16 saat darbesi ve 12 bayt bellek gerktirmektedir. Bu tip işlemleri hızlandırmak ve bellek gereksinimini azaltmak amacıyla aşağıdaki iki yeni komut tanımlanmıştır

BCLR n	Bellekteki n bitini sıfırla	Clear Bit n in Memory
BSET n	Bellekteki n bitinin birle	Set Bit n in Memory

Yukarıdaki altı satırlık kodu bu yeni komutlarla yazdığımızda

BCLR	3,$00	Sıfır adresli bellekteki bit 3'ü sıfırla	4~/2 bayt
BSET	0,$01	Bir adresli bellekteki bit 0'ı birle	4~/2 bayt

bellek gereksinimi üçte birine ve işlem hızı iki katına çıkmaktadır. Bu iki komut durum yazmacında (CCR) hiçbir biti değiştirmemektedir ve yalnızca ilk 256 bellek gözünde çalışmaktadır. HC08 ailesi mikroişlemcilerin giriş/çıkış kapıları ilk 256 bayt bellek aralığında bulunmaktadır.

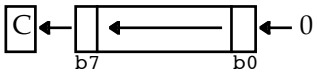
Bir bellek gözündeki bir biti veya birkaç biti sıfırdan bire veya birden sıfıra değiştirmek (evirmek) istiyorsak EOR komutundan aşağıdaki örnekteki gibi faydalanabiliriz:

LDA	PORTA	A giriş/çıkış kapısını oku	3~/2 bayt
EOR	#$04	Bit 2 yi evir	2~/2 bayt
STAA	PORTA	Neticeyi A giriş/çıkış kapısına yaz	3~/2 bayt

Kaydırma (shift) ve yuvarlama (rotate) komutları mantık işlemlerinin özel bir kısmını oluşturmaktadır. Bu komutlarla birikeç, X yazmacı ve bellekteki bitlerin yerlerini değiştirebiliriz. Örneğin aritmetik sola kaydır (arithmetic shift-left) komutu

ASLA	A'yı aritmetik sola kaydır	Arithmetic Shift Left A
ASLX	X'i aritmetik sola kaydır	Arithmetic Shift Left X
ASL	Belleği arit. sola kaydır	Arithmetic Shift Left memory

bütün bitleri bir göz sola kaydırır. Bu işlemde kullanılan yazmaç veya belleğin bit 7 içeriği CCR nin elde bitine ve bit 0 gözüne bir sıfır yerleştirilir.



LSLA	A'yı sola kaydır	Logic Shift Left A
LSLX	X'i sola kaydır	Logic Shift Left X

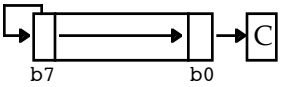
LSL	Bellek içeriğini sola kaydır	Logic Shift Left content of memory

komutları ASLx komutlarıyla eş anlamlıdır, çünkü aritmetik ve mantıksal sola kaydırma işlemleri aynıdır. Bütün bitlerin bir göz sola kaydırılması verinin iki ile çarpılmasına eşittir.

Birikecin, X yazmacının veya bellek içeriğinin sağa kaydırılmasında aritmetik ve mantık işlemleri arasında önemli farklılık vardır. Aritmetik sağa kaydırma işlemleri

ASRA	A'yı aritmetik sağa kaydır	Arithmetic shift right A
ASRX	X'i aritmetik sağa kaydır	Arithmetic shift right X
ASR	Belleği arit. sağa kaydır	Arithmetic shift right memory

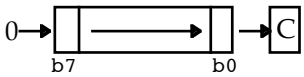
bütün bitleri bir göz sağa kaydırırken en önemli biti (bit 7) yerinde tutar ve bit 0 CCR nin elde bitine geçer.



En önemli bitin (bit 7) yerinde tutulması verinin işaretini korur ve işlem bu yüzden işaretli ikiye bölme işlemi gibi çalışır. ASRA işleminden önce birikecin içeriğinin $80 (ondalık -128) olduğunu varsayarsak, işlem sonrası $C0 (ondalık -64) olduğunu görürüz.

Mantıksal sağa kaydırma mantıksal sola kaydırma işleminin tümlevidir. Mantıksal sağa kaydırma bütün bitleri bir göz sağa kaydırır, en önemli bite (bit 7) sıfır yazar ve bit 0 CCR nin elde bitine geçer. Bu işlem işaretsiz bir ikiye bölme işlemiyle eşdeğerdir.

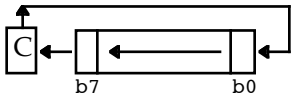
LSRA	A'yı mantıksak sağa kaydır	Logic Shift Right A
LSRX	X'i mantıksak sağa kaydır	Logic Shift Right X
LSR	Belleği mantık. sağa kaydır	Logic Shift Right content of memory



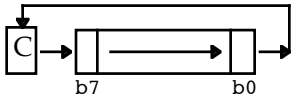
Yuvarlama komutları kaydırma komutları gibi birikeç, X yazmacı veya bellekteki bitleri bir göz kaydırırlar. Buna ilaveten CCR'deki elde biti bir taraftan içeriye, diğer taraftan da çıkan bit CCR'ye çıkarılır. Bu dairesel işlemden dolayı bu komutlara

yuvarlama komutları denir.

ROLA	A'yı sola doğru yuvarla	ROtate Left A
ROLX	X'i sola doğru yuvarla	ROtate Left X
ROL	Belleği sola doğru yuvarla	ROtate Left content of memory



RORA	A'yı sağa doğru yuvarla	ROtate Right A
RORB	X'i sağa doğru yuvarla	ROtate Right X
ROR	Belleği sağa doğru yuvarla	ROtate Right content of memory



Yuvarlama komutları çoklu baytlı ikiyle çarpma ve bölme işlemlerinde kullanılır. Aşağıdaki komut sırası

```
ASL   $A2
ROL   $A1
ROL   $A0
```

$A0 - $A2 bellek gözlerindeki 24-bitlik sayıyı ikiyle çarpar.

Yazılımı sadeleştiren bir başka komut ise birikeçteki yarım baytların yerini değiştiren (NSA) dır. Bu komutla birikecin alt yarısı ile üst yarısı yer değiştirir. İkili gösterimli ondalık sayı (BCD) sisteminde aritmetik işlemlerde kolaylık sağlar. Aşağıdaki örnekte, BCD1 ve BCD2 bellek gözlerinin sıfır ila üçüncü bitlerinde bulunan tek hane BCD sayılar birleştirilerek tek bir bayt haline getirilmektedir. BCD1 ve BCD2 bellek gözlerindeki sayıların bit 4-7 aralığı sıfırdır.

```
LDA   BCD1    Birinci BCD'yi oku
NSA           Yerlerini değiştir
ADD   BCD2    İkinci BCD'ye topla
```

NSA komutu olmasa idi aynı işlem şöyle gelişirdi:

```
LDA     BCD1
LSLA
LSLA
LSLA
LSLA
ADD     BCD2
```

Geriye kalan mantık işlemleri komutları

CLC	Elde bitini (C) sıfırla	SEC	Elde bitini bir yap
CLI	Kesme bitini (I) sıfırla	SEI	Kesme bitini bir yap
CLV	V bitini sıfırla	SEV	V bitini bir yap

durum yazmacındaki (CCR) bazı bitleri değiştirmemize yarar.

2-4-4 Denetim komutları (Control instructions)

Bir sonraki komut sınıfı, denetim veya program akış denetim komutları olarak adlandırılır. Bu komutlar program sayacını etkiler. Veri aktarma sınıfından sonra bu sınıf komutlar kullanım sıklığında ikinci sırada yer alırlar. Denetim komutları dallanma, atlama, altyordam ve kesme alt grupları olarak sınıflandırılabilir.

Öncelikle dallanma komutlarını inceliyelim. Dallanma komutları program sayacı göreceli adresleme kipini kullanır. Bir kaynak program, dallanma komutundaki hedef adresi birleştirici (assembler) programının hesaplayarak değerlendirebileceği sayısal bir değer, bir sembol veya veya bir tanımla belirler. Birleştirici hedef adres ile program sayacının o anki değeri arasındaki fark değeri hesaplayarak 8-bitlik ötelemeyi bulur. Program koşum sırasında sınanan koşul yerinde ise 8-bitlik işaretli öteleme değeri 16-bite çevrilir ve o anki program sayacı değerine toplanır. Bu toplama sonucu program sayacına yüklenerek programın dallanma komutunda belirlenen hedef değerinden devam etmesi sağlanır. Sınanan koşul yerine getirilmiyorsa, program dallanma komutunun hemen ardından gelen komutun işlenmesiyle devam eder. Tablo 2-1 bütün dallanma komutlarının bir özetini vermektedir.

İki adet koşulsuz ve 18 adet koşullu dallanma komutu mevcuttur. Örneğin

BRA Etiket

komutuyla koşulsuz olarak (branch always) Etiket değerine dallanma gerçekleşecektir. Bu işlemde Etiket'in değeri program sayacına yüklenir. BRA komutunun tümlevi

BRN Etiket

(branch never) hiçbir zaman Etiket'e dallanmıyacaktır. İlk bakışta bu anlamsız gelecektir, fakat BRN komutunun makine kodu programlayıcı tarafından program deneme ve kontrolu sırasında kolaylıkla diğer bir dallanma koduna çevrilebilecektir. Bu şekilde program işlem kontrolu kolaylaştırılabilir. BRN komutu iki bayt yer tutar ve buna karşı 3 saat darbesinde işlenir. Yazılımda gecikme amacıyla da kullanılabilir.

Tablo 2-1. Dallanma Komutları Özeti

Dallanma (Branch)				Tümlev dallanma			Tipi
Test	Boole işlem	Mnemonic	Kodu	Test	Mnemonic	Kodu	
r>m	(Z)∩(N⊕V)=0	BGT	92	r≤m	BLE	93	işaretli
r≥m	(N⊕V)=0	BGE	90	r<m	BLT	91	işaretli
r≤m	(Z)∩(N⊕V)=1	BLE	93	r>m	BGT	92	işaretli
r<m	(N⊕V)=1	BLT	91	r≥m	BGE	90	işaretli
r>m	(C)∩(Z)=0	BHI	22	r≤m	BLS	23	işaretsiz
r≥m	(C)=0	BHS/BCC	24	r<m	BLO/BCS	25	işaretsiz
r=m	(Z)=1	BEQ	27	r≠m	BNE	26	işaretsiz
r≠m	(Z)=0	BNE	26	r=m	BEQ	27	işaretsiz
r≤m	(C)∩(Z)=1	BLS	23	r>m	BHI	22	işaretsiz
r<m	(C)=1	BLO/BCS	25	r≥m	BHS/BCC	24	işaretsiz
elde	(C)=1	BCS	25	elde yok	BCC	24	basit
sonuç=0	(Z)=1	BEQ	27	sonuç≠0	BNE	26	basit
negatif	(N)=1	BMI	2B	pozitif	BPL	2A	basit
I mask	(I)=1	BMS	2D	I mask=0	BMC	2C	basit
H-bit	(H)=1	BHCS	29	H=0	BHCC	28	basit
IRQ=1	–	BIH	2F	IRQ=0	BIL	2E	basit
her zaman	–	BRA	20	hiçbir zaman	BRN	21	koşulsuz

açıklamalar : ∩ mantık ve işlemi; ⊕ dışlayıcı veya

Koşullu dallanma komutları durum yazmacı bitlerini sınar. Daha önce de belirtildiği gibi bu bitlere çok dikkat edilmesi gerekir, çünkü programdaki mantık hatası kolay kolay gözükmez. Durum yazmacındaki bitler genellikle yazılımdaki hatanın sorumlusudur, çünkü programı yazan hangi bitin ne koşul altında değiştiğini ve kullanılan dallanma komutunun da hangi bitlere bakarak karar verdiği doğru anımsayamabilir. Bu amaçla Ek 1 de komutlarin en sağ kolonunda CCR deki bitlerin

nasıl değiştiğini kontrol etmekte fayda vardır. Genelde veri aktarma komutları ya hiçbir biti değiştirmez, ya da N ve Z bayraklarını değiştirir ve C bitini değiştirmezler. Aritmetik komutlar ise genellikle H, N, Z, V ve C bitini, mantık işlemleri ise N, Z, ve C bitlerini değiştirir. Her işlemde hangi bitin ne şekilde değiştiği bir mantık sonucudur.

Durum yazmacının CCR yalnız bir bitini sınayan 12 adet basit dallanma komutu vardır.

BNE	Etiket	Z = 0 ise Etikete dallan
BEQ	Etiket	Z = 1 ise Etikete dallan
BPL	Etiket	N = 0 ise Etikete dallan
BMI	Etiket	N = 1 ise Etikete dallan
BCC	Etiket	C = 0 ise Etikete dallan
BCS	Etiket	C = 1 ise Etikete dallan
BHCC	Etiket	H = 0 ise Etikete dallan
BHCS	Etiket	H = 1 ise Etikete dallan
BIL	Etiket	IRQ pini sıfır seviyesinde ise dallan
BIH	Etiket	IRQ pini bir seviyesinde ise dallan
BMC	Etiket	I = 0 ise Etikete dallan
BMS	Etiket	I = 1 ise Etikete dallan

Çoğu zaman karşılaştırma (compare) ve çıkartma (subtracht) işlemlerinde iki sayı karşılaştırılır. Genellikle bu işlemlerden sonra neticenin pozitif, negatif, küçük v.s. durumlarına göre bir dallanma gerekir. Aşağıdaki tabloda Test sütununda R bir yazmaç, M ise bir bellek içeriği (veya iki bellek gözü içeriği) olarak düşünülürse işlemdeki sayıların işaretli ve işaretsiz olmalarına bağlı olarak gereken dallanma komutları verilmiştir.

Test	**İşaretli**	**İşaretsiz**
$R < M$	BLT	BLO (veya BCS)
$R \leq M$	BLE	BLS
$R \geq M$	BGE	BHS (veya BCC)
$R > M$	BGT	BHI

İkinin tümlevi gösterim veya işaretli sayılar için dallanma komutları :

BLT	Daha küçük ise dallan	$N \oplus V = 1$ ise dallan

BLE	Daha küçük veya eşitse dallan	$Z + (N \oplus V) = 1$ ise dallan
BGE	Daha büyük veya eşitse dallan	$N \oplus V = 0$ ise dallan
BGT	Daha büyükse dallan	$Z + (N \oplus V) = 0$ ise dallan

İsaretsiz sayılar için dallanma komutları :

BLO	Daha azsa dallan	$C = 1$ ise dallan
BLS	Daha az veya eşitse dallan	$C + Z = 1$ ise dallan
BHI	Daha çoksa dallan	$C + Z = 0$ ise dallan
BHS	Daha çok veya eşitse dallan	$C = 0$ ise dallan

CCR nin C bitine göre karar veren BLO ile BCS komutları ve BHS ile BCC komutlarının aynı olduğuna dikkat edin. Komutların hangi CCR bitlerini etkilediklerini Ek 1 deki komut özetinin dikkatle analiz edin. Her dallanma komutu bir 8-bitlik öteleme baytı ile beraber çalışır Bu ikinin tümlevi gösterimli bir baytlık öteleme ile en büyük öteleme +127 ile -128 ile sınırlıdır. Genellikle az da rastlansa, dallanma etiketi bu mesafenin dışında kalırsa, ilave BRA komutları kullanarak bu sorun çözülebilir. (tıpkı geniş bir dereyi aşmak için derenin ortasına yerleştirilen atlama taşları gibi)

İkinin tümlevi taşma bayrağını sınayan ve buna göre dallanma yapabilen iki komutun (bu komutlar başka mikroişlemcilerde BVC ve BVS olarak var) olmadığına dikkat edin. Bu komutları aşağıdaki örnek ile gerçekleştirebiliriz :

```
        TPA
        TSTA
        BMI     V_SET
        ////                    V bir değilse gereken komut(lar)
V_SET   ////                    V bir ise gereken komut
```

V bayrağının CCR nin en önemli biti olduğunu hatırlıyalım.

Dallanma komutlarına ilaveten dallanma ile bir başka komutun birleşimi olan komutlar mevcuttur. Bunlar

BRCLR n	Bellekteki n'ninci bit sıfırsa dallan
BRSET n	Bellekteki n'ninci bit bir ise dallan
CBEQ	Birikeç ile bellek içeriği eşitse dallan

CBEQA	Birikeç ile hemen adreslemedeki değer eşitse dallan
CBEQX	X yazmaç ile hemen adreslemedeki değer eşitse dallan
DBNZ	Bellek gözündeki sayıyı bir azalt, sıfır değilse dallan
DBNZA	Birikeci bir azalt, sıfır değilse dallan
DBNZX	X yazmacı bir azalt, sıfır değilse dallan (H'nin içeriği değişmez)

BRCLR n komutu ilk 256 adresteki bellek gözünün n'ninci bitinin sınar, bit sıfıra eşitse komutta verilen etikete dallanır. Sınanan bit sıfır değilse bir sonraki komut işlenir. BRSET n komutu ilk 256 adresteki bellek gözünün n'ninci bitinin sınar, bit bire eşitse komutta verilen etikete dallanır. Sınanan bit bir değilse bir sonraki komut işlenir.

CBEQ komutu adreslenmiş bellekteki veriyi birikeçle (CBEQX komutunda X yazmacıyla) karşılaştırır ve yazmaç (A veya X) içeriği bellek içeriğine eşitse dallanmayı yapar. CBEQ komutu CMP ve BEQ komutunun birleştirilmiş halidir ve başvurma tablolarının taranmasını önemli miktarda hızlandırır. CBEQ komutu CCR bitlerini değiştirmez.

CBEQ komutunun IX+ adresleme tipi H:X in gösterdiği bellek gözündeki veriyi A birikeci ile karşılaştırır ve veriler eşitse dallanma gerçekleşir. Dallanma yapılsın yapılmasın H:X her seferinde bir arttırılır. CBEQ komutunun IX1+ adresleme tipinde veriye erişmek amacıyla H:X yazmacına 8-bitlik bir öteleme ilave edilerek bellek adreslenir.

Bir karakter dizisi içinde boşluk (ASCII $20) üzerinden atlıyan basit bir kod dizisi yazalım. Dizinin en az bir adet boşluk olmayan bir karakter bulundurduğunu ve programa girişte H:X in dizinin başına işaret ettiğini varsayalım. Program çıkışında H:X dizinin içindeki ilk boşluk olmayan karaktere işaret edecektir.

```
        LDA     #$20        Boşluk karakterini birikece yükle
ATLA    CBEQ    X+,ATLA     Boşluk olmayan karakteri bulana
*                           kadar H:X 'i arttır.
        AIX     #-1         İmleci ilk boşluk olmayan karaktere getir
```

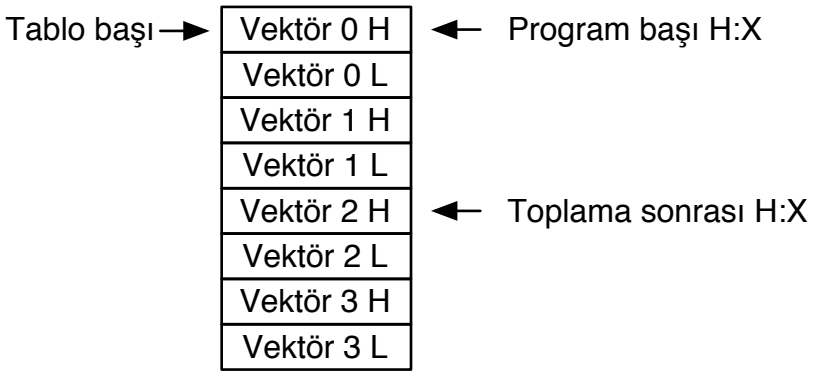
CBEQ komutunun H:X imlecini dallanma olsun olmasın arttırdığına dikkat edelim. Bu örnekte, CBEQ komutunun dallanmamasının hemen ardından H:X 'in boşluk olmayan karakter artı bire işaret edecektir. Bu yüzden AIX #-1 ile H:X bir azaltılarak düzeltilmektedir.

CBEQA ve CBEQX birikecin veya index yazmacının alt yarısının içeriğini bir hemen veriyle karşılaştırırlar ve veriler eşit ise dallanmayı gerçekleşir.

DBNZ, DBNZA ve DBNZX döngü mekanizmaları bellekten, birikeçten (A) veya X yazmacından bir çıkartır, daha sonra çıkartma işlemi sonucu $00 değilse program komutun gösterdiği etikete dallanır. DBNZX H:X yazmacının yalnız alt yarısını etkiler, üst yarı (H) içeriği değişmez. DBNZX komutu için bir örnek daha önce kısım 2-4-2 de ADD ve ADC komutlarının anlatımında verilmişti.

Program sayacı göreceli koşulsuz dallanma komutu BRA 'nın doğrudan, indeks ve genişletilmiş adres kipini kullanan eşdeğeri JMP "jump=atla" komutudur. Etkin atlama adresi, doğrudan adreslemede komutta belirtilen ilk 256 baytlık bellek aralığındaki yer, indeksli adreslemede H:X yazmacı artı belirtilen öteleme değeri, genişletilmiş adreslemede ise belirtilen 16 bitlik değerdir.

Atla (JMP) komutunun indeksli adresleme kipi atlama tablolarının kullanımını kolaylaştırdığı için özellikle önemlidir. Bu amaçla küçük bir program yazalım. Program başlangıcında H:X yazmacının atlama tablosunun bellekteki başlangıç adresine işaret ettiğine ve birikecin (A) kaçıncı sıradaki atlama değerine (vektörüne) eşit olduğunu var sayalım. Atlama adresleri (vektörleri) 16 bitlik değerler olduğundan her bir adres girdisi iki bellek gözünde saklıdır. Program dökümünden önce H:X in vektör tablosu başlangıcına (vektör sıfır) işaret ettiğini ve birikecin ikiye eşit olduğunu varsayalım.




```
LSLA                    A 'yı ikiyle çarp
PSHX                    X 'I sonra H 'yi yığına it
PSHH
ADD     2,SP            Yığındaki X 'i A 'ya topla
```

```
        TAX             Toplamı X 'e kopyala
        PULA            Yığındaki H 'yi A 'ya çek
        ADC     #0      Toplamada oluşabilmiş eldeyi kat
        PSHA            A 'daki toplamı yığın üzerinden
        PULH            H 'ye aktar
        PULA            Yığını düzelt
        LDX     1,X     Vektör alt baytını X 'e yükle
        LDA     ,X      Vektör üst baytını A 'ya yükle
        PSHA            A 'yı H 'ye kopyala
        PULH
        JMP     ,X      Vektör adresindeki programa atla
```

Vektör tablosu girdileri iki baytlık adres olduğundan birikeç H:X ile toplama işleminden önce ikiyle çarpılması gerekmektedir. LDHX komutunun indeksli adresleme kipinin bulunmaması yüzünden bu işlem için önce H:X in alt yarısı (X) tablodan bir ötelemeli indeksli olarak yüklenir, sonra birikeç (A) sıfır ötelemeyle indeksli olarak yüklenir , yığına itilir ve H olarak yığından çekilir. Bu uzun işlem sonucu H:X tablodan yüklenmiş olur ve indeksli atlama gerçekleştirilebilir.

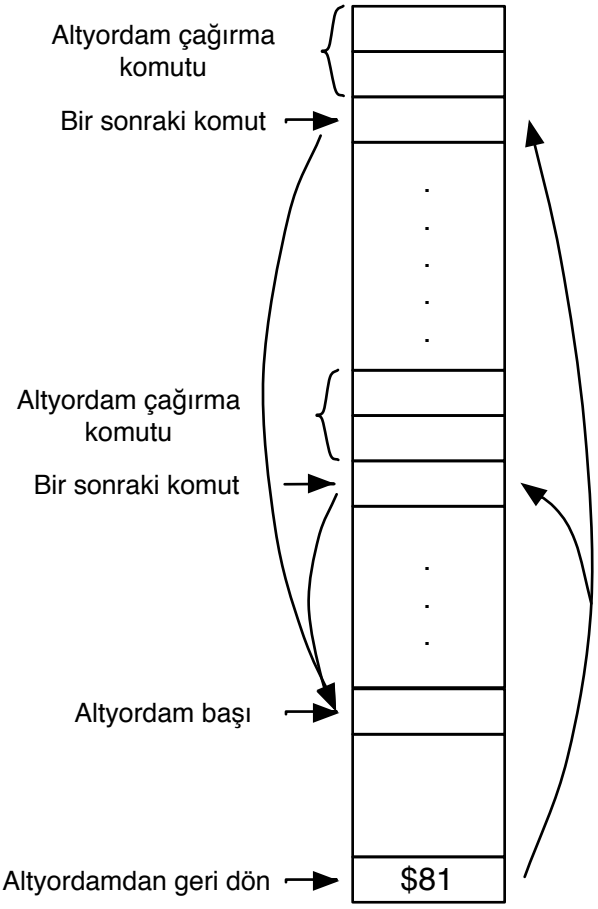
NOP "No operation" komutu da bir koşulsuz dallanmadır. Bu komut program sayacını bir arttırmaktan başka bir şey yapmaz. Bu komut bir bayt ve bir saat darbesi zaman harcar ve genellikle kısa gecikme döngülerinde kullanılır.

```
        CLRA            Döngü sayacını 0 = 256 yap
DLOOP   NOP
        NOP
        DECA
        BNE     DLOOP
```

NOP kullanılmadan elde edilecek gecikme 256 x 4 = 1024 saat darbesi, iki adet NOP kullanılarak gecikme 256 x 6 = 1536 saat darbesi olur.

Bugüne kadar yazdığınız programlarda bir program parçasını birkaç yerde tekrarladığınız olmuştur. Aynı program içinde aynı kod parçasını tekrar tekrar yazmaktan nasıl kurtulabileceğinizi merak ettiniz mi? Bunun için iki çözüm vardır, birincisi altyordam çağırma, ikincisi makro tanımlamadır. Biz birinci çözüm, altyordam çağırmayı araştıracağız. Altyordam program parçası sonuncu komutu kendisini çağıran ana programa geri dönmesini sağlar. Bu komut, program sayacına

ana programdaki geri dönüş adresini yüklemekle yükümlüdür. Bu amaçla alt yordam çağırma komutları kendinden sonra gelen komutun adresini (geri dönüş adresi) yığında saklarlar. Alt yordamdan geri dön (Return from subroutine) RTS komutu yığından iki bayt okur ve bu 16 bitlik veriyi program sayacına yükler. Alt yordamdan geri dönme RTS işleminin doğru çalışmasını sağlamak amacıyla yığın imlecinin RTS komutu öncesi alt yordam başındaki değere eşit olması gerekmektedir.



Şekil 2-2. Alt yordam çağırma ve geri dönme

Freescale HC08 ailesi mikroişlemcilerde iki altyordam çağırma komutu vardır:

BSR	Altyordama dallan	Branch to SubRoutine
JSR	Altyordama atla	Jump to SubRoutine

BSR komutu program sayacı göreceli adresleme, JSR komutu ise doğrudan, indeksli, ve genişletilmiş adresleme kiplerini kullanır.

Sekiz bitlik sayıların toplanması, çıkarılması, çarpılması ve hatta 16 bitlik bir sayının 8 bitlik bir sayıya bölünmesi işlemleri bu amaçlara uygun komutların mevcudiyetinden çok kolaydır. Buna karşın iki 16 bitlik sayıyı çarpmak ve 32 bitlik bir

sonuç elde etmek istediğimizde, kısa bir program parçası yazmamız gerekir. Bu çarpma işlemini yazdığımız programda birden fazla yerde kullanmamız gerekirse bu çarpma programı parçasını bir altyordam halinde yazmakta fayda vardır. Aşağıda iki işaretsiz 16-bitlik sayının çarpım altyordamı (mpy16) örnek olarak verilmiştir. Bu altyordam başlangıçta H:X in gösterdiği iki 16-bitlik sayıyı çarpar ve 32-bit sonucu baştakı sayıların üzerine yazar. İşlem sırasında ara neticelerin saklanması için gerekli bellek yığında gerçekleştirilmiştir.

```
*****************************************************
*
* 16-bit işaretsiz çarpma
*
* program girişinde:
*  2*16-bit veriye (çarpılan ve çarpan)
*  H:X yazmacı işaret eder
*
* program çıkışında:
*  H:X 32-bitlik sonuca işaret eder
*
* değişken:
*  accumulator
*
* bütün hesap ara verileri yığında saklanır
* Prof. Dr. Ömer Cerid 2002
*
*****************************************************

mpy16:
          pshx
          pshh
          ais    #-6T
          lda    3,x     ; çarpanın alt baytını oku
          psha           ; yığına sakla
          lda    2,x     ; çarpanın üst baytını oku
          psha           ; yığına sakla
          lda    1,x     ; çarpılanın alt baytını oku
          psha           ; yığına sakla
          lda    ,x      ; çarpılanın üst baytını oku
          psha           ; yığına sakla
          ldx    2,sp    ; x'e çarpılanın alt baytını yükle
          lda    4,spc
          mul            ; alt baytları çarp
```

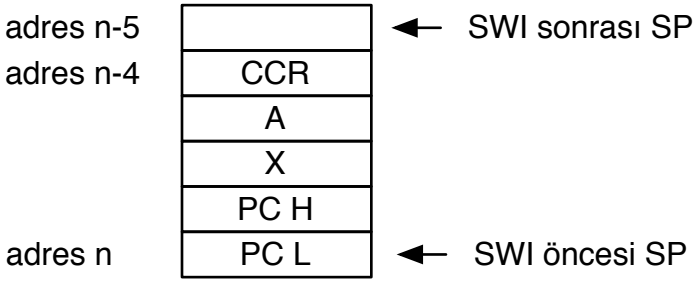
```
sta    8,sp   ; sonucu yığına yaz
stx    7,sp
ldx    2,sp   ; x'e çarpılanın alt baytını yükle
lda    3,sp   ; a'ya çarpanın üst baytını yükle
mul           ; çarp
add    7,sp   ; bir önceki çarpımın alt baytına topla
sta    7,sp   ; toplamı sakla
txac
adc    #0     ; son toplamanın eldesini a'ya topla
sta    6,sp   ; toplamı sakla
ldx    1,sp   ; x'e çarpılanın üst baytını yükle
lda    4,sp   ; a'ya çarpanın alt baytını yükle
mul           ; çarp
add    7,sp   ; bir önceki çarpımın alt baytına topla
sta    7,sp   ; toplamı sakla
txa           ; x ila a'yı değiştir
adc    6,sp   ; eldeyi topla
sta    6,sp   ; sakla
lda    #0     ; a'ya eldeyi bozmadan sıfır yükle
adc    #0     ; eldeyi a'ya topla
sta    5,sp   ; sakla
ldx    1,sp   ; x'e çarpılanın üst baytını yükle
lda    3,sp   ; a'ya çarpanın üst baytını yükle
mul           ; çarp
add    6,sp   ; ara toplamı yap
sta    6,sp   ; sakla
txa           ; x ila a'yı değiştir
adc    5,sp   ; eldeyi kat
sta    5,sp   ; sakla
tpa           ; durum yazmacını a'ya kopyala
sei           ; kesmelere mani ol!
ais    #10T   ; yığın imlecini on arttır
pulh          ; program başındaki H'yi yığından çek
pulx          ; program başındaki X'i yığından çek
ais    #-8T   ; yığın imlecini sekiz azalt
tap           ; durum yazmacını geri yükle
pula          ; 32-bit sonucu yığından çek
sta    ,x     ; H:X'in gösterdiği yere yaz
pula
sta    1,x
pula
sta    2,x
pula
```

```
        sta    3,x
        ais    #4      ; yığın imlecini düzelt
        rts            ; altyordamdan geri dön
```

Denetim komutlarının son gurubu kesme komutlarıdır. Bu komutlar ,altyordam komutları gibi ana programın akışını geçici süre için durdurur, kesme programını koşar, ve sonunda ana programa hiçbir şey olmamış gibi geri dönerler. Kesme komutları H yazmacı ve yığın imleci dışında bütün yazmaçları yığında otomatik olarak saklar. Şu kesme komutları vardır:

SWI	Yazılım kesme	SoftWare Interrupt
WAIT	Kesmeye hazırlan, beklemeye geç ve CPU'yu durdur Enable interrupts; stop CPU	
STOP	Kesmeye hazırlan, saati durdur Enable interrupts; Stop Clock Oscillator	
RTI	Kesmeden geri dön	ReTurn from Interrupt

Kesmeler aslında ya donanım veya yazılım kaynaklıdır. Bu kısımda komutları incelediğimizden yazılım kaynaklı kesmeleri inceleyeceğiz. İleri bölumlerde mikroişlemci mikrodenetleyicilerin donanımlarını incelediğimizde donanım kaynaklı kesmeleri de çok ayrıntılı olarak inceleyeceğiz. Yazılım kesme SWI kullanıcı tarafından programa bilerek sokulan bir kesme işlemidir. SWI komutu işlendiğinde merkezi işlem birimi CPU aşağida Şekil 2-3 de gösterildiği sırada yazmaçları yığına yazarak saklar, sonradan oluşabilecek kesmeleri önlemek amacıyla durum yazmacındaki I bitini bir konumuna getirir, ve sonunda $FFFC : $FFFD bellek gözlerindeki yazılım kesme vektör bilgisini okur ve program sayacına yükler.



Şekil 2-3. Kesme sırasında yığına yazma

$FFFC : $FFFD bellek gözlerindeki değer yazılım kesme servis yordamının başlangıç adresi veya vektörüdür. Başlangıç adresi her zaman aynı adresten okunduğu için SWI komutu içsel adresleme kipli bir komuttur. Kesme servis programı

her zaman yığında saklı yazmaç içeriklerini geri yükleyen RTI “Kesmeden geri dön” komutu ile bitmelidir. SWI komutu genelde kullanıcı programının akışını denetlemek ve hata ayıklamak amacıyla programa sokulur ve bu yere kesme noktası adı verilir. Kesme noktasında ana program durdurulur ve denetim özel bir pograma devredilir. Bu programla yazmaç ve bellek içerikleri görülebilir ve değiştirilebilir. Gerekli inceleme, düzeltmeden sonra kesme noktasından geri dönülüp ana programa devam edilebilir. HC05 ailesi mikrodenetleyiciler uyumluluğu için H yazmacı yığında otomatik olarak saklanmamaktadır. Kesme servis programı H yazmacının içeriğini değiştiriyorsa, kesme servis programı başında PSHH komutuyla H yazmacı içeriği yığında saklanmalı ve RTI komutundan önce bir PULH komutuyla geri yüklenmelidir.

WAIT ve STOP komutları iki değişik güç tüketimi azaltma kipi elde etmek amacıyla ilave edilmiştir. WAIT komutunu işleyen mikroişlemci önce CCR deki I bitini sıfırlayarak maskelenebilir kesmeleri işleyebilir hale gelir, sonra merkezi işlem biriminin (CPU) saat darbelerini keserek durdurur. Merkezi işlem biriminin durdurulması sonucu mikroişlemcinin elektrik tüketimi önemli oranda azalır. Mikrodenetleyicide bulunan giriş/çıkış ünitelerinin ve zamanlayıcının (Timer) saat darbeleri kesilmez ve kesme sinyali üretebildiklerinden işlemeye devam ederler. Bir kesme üzerine merkezi işlem biriminin saat darbeleri yeniden aktive edilir. Bu durumda CPU yazmaçlarını sırasıyla yığına saklar ve kesmeyi yaratan donanıma özgün kesme vektörünü yükleyerek kesme servis yordamını koşmaya başlar. STOP komutu işlenmesi durumunda ise CPU CCR deki I bitini sıfırladıktan sonra mikrodenetleyicinin saat osilatörünü durdurur. Mikrodenetleyici CMOS bir tümleşik devre olduğundan saat darbeleri olmadığından neredeyse sıfıra yakın bir akım tüketme kipine girer. Mikrodenetleyicinin IRQ girişine uygulanan bir kesme veya donanım sıfırlama (Reset) girişine sinyal uygulayarak mikrodenetleyici saat osilatörünü yeniden çalışır hale getirir. Saat osilatörünün yeniden çalışıp kararlı hale gelmesi uzunca bir zaman gerektirir. Güç tüketimi azalma ile ilgili ayrıntılı bilgi ileri kısımlarda verilecektir.

Tablo 2-2 MC68HC908JL16 mikrodenetleyicisinin bütün kesme vektörlerinin bir listesini vermektedir. Reset bütün kesmelerin en önemlisidir ve her mikrodenetleyici her ne durumda olursa olsun, hemen algılanır. SWI bir donanım kesmesi olduğundan ve CCR deki I bitinin durumundan bağımsız olduğundan Reset’ten sonra en yüksek önceliğe sahiptir. IRQ ve ondan sonra gelen tümleşik devredeki diğer çevre elemanlarının kesme vektörleri öncelikleri düşerek sıralanırlar. Aynı anda iki kesme oluştuğunda mikrodenetleyici daha yüksek önceliği olanının servis yordamını işleyecektir.

Tablo 2-2. MC68HC908JL16 MCU Kesme Vektör Adresleri

Vektör Önceliği	Adres	Vektör
En düşük	$FFDE	ADC çevrim tamam (üst)
	$FFDF	ADC çevrim tamam (alt)
	$FFE0	Tuş kesme vektörü (üst)
	$FFE1	Tuş kesme vektörü (alt)
	$FFE2	SCI gönderme vektörü (üst)
	$FFE3	SCI gönderme vektörü (alt)
	$FFE4	SCI alma vektörü (üst)
	$FFE5	SCI alma vektörü (alt)
	$FFE6	SCI hata vektörü (üst)
	$FFE7	SCI hata vektörü (alt)
	$FFE8	MMIIC vektörü (üst)
	FFE9	MMIIC vektörü (alt)
	–	Kullanılmamış
	$FFEC	TIM2 taşma vektörü (üst)
	$FFED	TIM2 taşma vektörü (alt)
	$FFEE	TIM2 kanal 1 vektörü (üst)
	$FFEF	TIM2 kanal 1 vektörü (alt)
	$FFF0	TIM2 kanal 0 vektörü (üst)
	$FFF1	TIM2 kanal 0 vektörü (alt)
	$FFF2	TIM1 taşma vektörü (üst)
	$FFF3	TIM1 taşma vektörü (alt)
	$FFF4	TIM1 kanal 1 vektörü (üst)
	$FFF5	TIM1 kanal 1 vektörü (alt)
	$FFF6	TIM1 kanal 0 vektörü (üst)
	$FFF7	TIM1 kanal 0 vektörü (alt)
	–	Kullanılmamış
	$FFFA	IRQ vektörü (üst)
	$FFFB	IRQ vektörü (alt)
	$FFFC	SWI vektörü (üst)
	FFFD	SWI vektörü (alt)
	$FFFE	Reset vektörü (üst)
En yüksek	$FFFF	Reset vektörü (alt)

2-4-5 Giriş/Çıkış Komutları

Son komut çeşidini oluşturan giriş/çıkış komutları Freescale mikroişlemci ve mikrodenetleyicilerde yoktur. Bütün Freescale mikrodenetleyiciler bağımsız giriş/çıkış yerine bellek adreslenmiş giriş/çıkış kullanırlar. Bellek adreslenmiş giriş/çıkış diğer tipe göre üstündür, çünkü özel komut gerektirmeden mevcut veri aktarımı, aritmetik, ve mantık işlemleri tipi bütün komutlar giriş/çıkış ünitelerinde kullanılabilir.

2-5 Birleştirici (Assembler) Komutları

HC08 ailesinin bütün komutlarını işledik. Birleştirici (Assembler) programının bir kaynak yazılımından makine kodunu üretebilmesi için assembler programına ilave bilgiler vermek gerekir. Bu amaçla assembler komutları adıyla anılan ilave komutlar tanımlanmıştır [2]. En sık kullanılanların listesi aşağıya çıkarılmıştır:

DS #n	Bellek tanımla	n sayısı kadar baytlık bir bölgeyi bellekte ayır,
END	Kaynak program sonu	
EQU	Belirle	Etiketi veriye eşitle,
FCB	Sabit bayt tanımla	İşlemin veri kısmında belirtilen sayıyı (sayıları) belleğe yaz,
FCC	Sabit karakter tanımla	İşlemin veri kısmında belirtilen ASCII karakter dizisini belleğe yaz,
FDB	İki bayt tanımla	İşlemin veri kısmında belirtilen iki baytlık sayıyı (sayıları) belleğe yaz,
MACR	Makro tanımlama	Bir makro tanımlama başı,
MEND	Makro tanımlama sonu	
ORG	Başlangıç noktası	Program sayacının veriye eşitlenmesini sağla,
RMB	Bellekte yer ayır	Veri kısmında tanımlanan miktarda baytı bellekte kullanıma ayır.

Bazı makroasembler programlarının açıklamalar kısmının ayırımı için boşluk karakteri yerine noktalı virgül gerektirdiğini belirtmekte fayda vardır. Daha ayrıntılı birleştirici komutları listesi ve açıklamaları "68HC08 In-Circuit Simulator Operator's Manual" [2] isimli kitapta bulunabilir. Birleştirici komutlarına örnek vermek için daha önce verdiğimiz bir programı yeniden yazalım:

```
PORTA   EQU     $00         PORTA'nın adresini tanımla
PORTB   EQU     $01         PORTB'nin adresini tanımla
*
        ORG     $0100       Program sayacını $0100 'e eşitle
        LDA     PORTA       PORTA 'daki veriyi oku
        AND     #$FB        veriyi $FB ile ve işlemine tabi tut
        STA     PORTA       Veriyi PORTA 'ya yaz
        LDA     PORTB       PORTB 'deki veriyi oku
        ORA     #$01        veriyi $01 ile veya işlemine tabi tut
        STA     PORTB       Veriyi PORTB 'ye yaz
        END                 Program sonunu işaretle
```

EQU komutuyla PORTA ve PORTB giriş/çıkış kapılarını programın en başından tanımlayarak, üretilen kodun daha kolay anlaşılması ve değiştirilebilmesi sağlanmıştır.

Aşağıda verilen kesme servis programı, seri iletişim arabirimi (SCI) tarafından bir bayt alındığında koşacaktır. SCI alma kesme vektörü $FFE4:$FFE5 adreslerindedir ve kesme programının başlangıç adresini (SCISER) bulundurmaktadır. Burada FDB komutuyla birleştirinin SCISER 'in değerini hesaplayıp $FFE4:$FFE5 adresine yazması sağlanmaktadır. RMB komutuyla birleştiri INBUF imlecinin RAM bölgesinin $80 ve $81 bellek gözlerinde saklanabilmesi için yer ayırmaktadır.

```
SCS1    EQU     $16         SCI durum yazmacı 1 (status register 1)
SCS2    EQU     $17         SCI durum yazmacı 2  (status register 2)
SCDR    EQU     $18         SCI veri yazmacı (data register)
*
        ORG     $80         $0080 'i belirle (RAM bölgesi)
INBUF   RMB     2           Giriş imlecini saklama yeri ayır (iki bayt)
*
        ORG     $E000       SCI kesme programı başlangıç adresini belirle
SCISER  LDA     SCSI1       SCI durum yazmacı biri oku
        PSHH                H yazmacını yığında sakla
        LDHX    INBUF       giriş tampon bellek imlecini yükle
        MOV     SCDR,X+     almacın aldığı veriyi belleğe yaz, imleci bir arttır
        STHX    INBUF       tampon bellek imlecini sakla
        PULH                H yazmacını geri yükle
```

```
        RTI                     kesmeden geri dön
*
        ORG     $FFE4
        FDB     SCISER          SCI kesme servis progamının adresini tanımla
        END                     Program sonunu işaretle
```

Kesme servis programı H yazmacının içeriğini değiştirdiğinden, kesme servis programı başinda PSHH komutuyla H yazmacı içeriği yığında saklanmakta ve RTI komutundan önce bir PULH komutuyla geri yüklenmektedir.

Kaynakça

1. Freescale Inc., “CPU08RM/AD CPU08 Central Processing Unit Reference Manual” Revision 3, 2001.
2. Freescale Inc., “M68ICS08SOM/D M68ICS08 68HC08 In-Circuit Simulator Operator’s Manual”, chapter 4.