ARM Sembolik Makine Dili
(80X86 ve ARM Sembolik Makine Dilleri Kursu 2. Bölüm)
Kurs Notları

Kaan ASLAN

C ve Sistem Programcıları Derneği

Güncelleme Tarihi: 13/10/2020

---

1. ARM Mimarisine Giriş

1.1. ARM İşlemcilerinin Tarihsel Gelişimi

ARM işlemcilerinin tarihi "Acorn Computer" isimli İngiliz firmasına dayanmaktadır. Bu firma 80'li yılların başlarında "BBC Micro" isimli 64K'lık ev bilgisayarlarını yapmıştı. Bu bilgisayarlarda Rockwell'in 8 bitlik 6502 işlemcileri kullanılıyordu. Firma daha sonra "Bekeley RISC" projesinden etkilenerek kendi RISC işlemcilerini yapmaya heveslendi. Böylece ilk ARM modelleri tasarlanmış oldu. 1990'da "Apple" ve "VLSI Technology" şirketleriyle ortaklık kurarak ARM ismini aldı. (Eskiden ARM "Acorn RISC Machine" isminden kısaltlıyordu. Fakat daha sonra bu firma kurulunca bu kısaltma "Advanced RISC Machine" haline geldi.) Apple firması bu yeni firmaya maddi destek sağlamıştır. "VLSI Technology" firması ekipman tedarik etmiştir. Acorn ise az sayıda tasarım mühendisini firmaya aktarmıştır. ARM şirketi daha sonra çok uluslu bir biçimde yaşamını sürdürmüştür. Şirket ARM işlemcilerinin pek çok sürümünü tasarlamıştır. Bugün mobil telefonların %95'inde ARM işlemcileri kullanılmaktadır. Göülü sistemlerin %90'ına yakınında yine ARM işlemcileri bulunmaktadır.

ARM firması üretim değil tasarım firmasıdır. Firma işlemcileri mimari olarak tasarlayıp lisanslarını üretici firmalara satmaktadır. ARM mimarileri için Genel olarak iki tür lisans sözleşmesi bulunmaktadır. Bir firma ARM'dan yalnızca işlemcinin üretim lisansını almış olabilir. Bu durumda bu firma işlemcide tasarım değişikliği yapamamakta yalnızca onu üretmektedir. Tabii üretim genel olarak artık SOC (System On Chip) biçiminde içerisinde RAM ve ROM ünitleri de bulunacak biçimde yapılmaktadır. İkinci lisans türü geliştirme lisansıdır. Bu lisansa sahip olan firmalar işlemci tasarımını değiştirip, geliştirip üretim yapabilmektedir. Bugün örneğin Apple'ın A8, A9, A10 gibi SOC chiplerinde, Qualcomm'un Snapdragon chiplerinde, Samsung'un Exynos SOC chipleri tasarım lisansıyla geliştirilip üretilmişleridir.

ARM firması birkaç ay önce Japon Softbank firması tarafından satın alınmıştır.

1.2. ARM Mimarisinde Cortex'ler ve Versiyonlar

ARM şirketi tarihsel gelişimde pek çok işlemci tasarlamıştır. Bu işlemcile eskiden ARM1, ARM2,.., ARM-11 gibi kodlar veriliyordu. Sonra şirket bunların kod numaralarını değiştirdi. Bugün modern ARM işlemcilerinde iki önemli kavram kullanılmaktadır: Cortex ve Versiyonlar.

ARM terminolojisinde Cortex işlemcinin donanımsal mimarisini belirtir. Yani "işlemcinin içerisindeki birimler, çekirdek sayısı, güç harcaması, hızı vs. gibi özellikleri cortex ile ilgilidir. Versiyon ise arayüz özellikleridir. Versiyon en temelde makine komutlarının kümesini belirtir. Yani bir versiyondaki komutlar ile diğer versiyondaki komutlar ortak noktaların dışında farklı olabilmektedir. Farklı Cortex'ler aynı versiyonları kullanabilmektedir. Genel olarak ARM'ın ARM1'den ARM11'e kadar olan tasarımlarına klasik model denilmektedir. ARM firması daha sonra isimlendirmede "cortex" terimini kullanmaya başlamıştır. ARM'ın üç cortex grubu vardır: A serisi cortex'ler, R serisi cortex'ler ve M serisi cortex'ler. A serisi cortex'ler (buradaki

A “Application” sözcüğünden gelmektedir.) tablet, masaüstü, dizüstü ve mobil aygıtlarda tercih edilen cortex’lerdir. Örneğin kullandığımız cep telefonlarında, tabletlerde Raspberry Pi gibi, Orange Pi gibi geliştirme kitlerinde hep A serisi cortex’ler kullanılmaktadır. R serisi cortex’ler gerçek zamanlı projelerde kullanılmaktadır. (Buradaki R harfi “Real Time” sözcüklerinden gelmektedir) Örneğin hard disk devrelerindeki işlemciler, kritik işlemleri izleyen donanımsal aygıtlarda R serisi cortex’ler kullanılmkatdır. M serisi cortex’ler kendi içerisinde RAM’i ve ROM’u olan IO işlemlerine açık tipik mikrodenetleyici tarzı cortex’lerdir. (Buradaki M “Micrıcontroller” sözcüğünden gelmektedir.) M serisi cortex’ler çok az güç harcamasıyla ve düşük bellek miktarlarına sahip olmasıyla ve tabii ki ucuz olmasıyla karkaterizedir. A serisi, R serisi ve M serisi cortex’ler aynı versiyonları kullanıyor olabilir. Yani örneğin A serisi bir cortex ile M serisibir mikrodenetleyici cortex’i Version 7 arayüzüne sahip olabilir. Bu durumda bunların makine komutları aynıdır.
ortex terimi genel mimariyi beelirtmektedir. ARM firması cortex’in dışında aynı donanım mimarisine sahip olan cortex’leri ayırmak için ayrıca bir de “core” terimini kullannmaktadır. O halde “cortex” genel mimariyi belirtirken “core” o mimarideki özel modeli belirtmektedir. Örneğin Cortex-A isimli cortex ailesinde “Cortex A-5”, “Cortex-A8”, “Cortex A-9” gibi “core”lar vardır.

ARM versiyonları cortex’lere göre de değişebilmektedir. Gerçekten de örneğin Cortex-M serisi ile Cortex-A serisi tasarımlar Versiyon 7’yi kullanmakla birlikte yine de bunların makine komutları arasında farklılıklar bulunabilmektedir. Bu nedenle ARM firması bu versiyonları “Version 7-A”, “Version 7-M” gibi isimlerle dokümante etmiştir. Bu terimlere ilişkin özet olarak şunları söyleyebiliriz:

- ARM’ın ARM1’den ARM11’e kadar olan tasarımlarına “Klasik ARM” denilmektedir. ARM şirketi bu tasarımlarda henüz cortex terimini kullanmamıştır.

- Cortex genel donanımsal mimariyi belirtir. Modern ARM işlemcilerinin donanımsal mimarileri “Cortex” terimiyle kodlanmaktadır. Üç temel cortex grubu vardır: “Cortex-A”, “Cortex-R” ve Cortex-M”. Bu cortex’ler kendi aralarında modellere sahiptir. Bunlara da ARM firması “core” demektedir. Örneğin “Cortex A8”, “Cortex A9” gibi.

- Versiyon temel olarak arayüzü belirtir. Yani tipik olarak makine komutlarının biçimi, yazmaç yapısı, komutların kodlanma formatı versiyonu oluşturmaktadır. Bugün için en tipi iki versiyon “Versiyon-7” ve “Version-8”dir. Bu versiyonlar da kendi aralarında “Versiyon 7-M”, Version 7-R” gibi gruplara ayrılmaktadır. Versiyon-7 tipik 32 bit yeni ARM işlemcilerinin kullandığığı versiyondur. Versiyon-8 ise 64 bit yazmaçları ve komutları barındırmaktadır.

1.3. Çok Kullanılan Bazı Cihazlardaki Cortex ve Versiyonlar

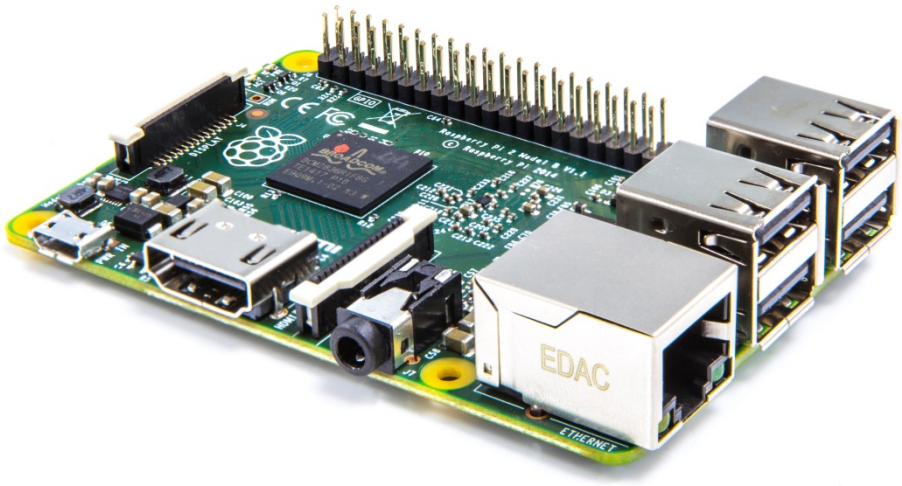
Kursumuzda ARM denemeleri Raspbery Pi 2 ve Raspberry Pi 3 üzerinde yapılacaktır. Bu aygıtlara Raspbian isimli Linux sürümü yüklenmiştir ve uzak masaüstü VNC bağlantısıyla bunlara bağlanılmaktadır. Bilindiği gibi Raspberry Pi düşük fiyatlı bilgisayar oluşturmak için geliştirilmiş bir projedir. Bu geliştirme kartlarının çok sayıda benzerleri başka firmalar tarafından üretilmiştir. (Banana Pi, Orange Pi, Beagle Bone Black gibi...) Raspberry Pi’ın birinci sürümünde klasik donanımsal mimari olarak ARM11, arayüz olarak Version-6 kullanılmıştır. Ancak Raspberry Pi’ın ikinci sürümünde artık Cortex-A serisine geçilmiş ve Cortex A7 core, arayüz olarak da Version-7 kullanılmıştır. Raspberry Pi’ın üçüncü versiyonunda Cortex-A53 ve arayüz olarak da ARM Version-8 (64 bit) kullanılmıştır.

Galaxy S7’de de yine Cortex-A53 ve arayüz olarak da Version-8 kullanılmıştır. Iphone 7’de Cortex-A53 türvi Apple A10-Fusion işlemcisi kullanılmıştır. Arayüz de yine Version-8’dir. Benzer biçimde IPad Pro modellerinde de Apple’ın Cortex-A53 türevi A9X işlemcileri kullanılmıştır. Bunların arayüzleri de yine Version 8’dir.

1.4. Raspberry Pi’ın ARM Sembolik Makine Dili Çalışması İçin Uygun Hale Getirilmesi

Yukarıda da belirtildiği gibi kursumuzda denemeler için Raspberry Pi’ın 2 numaralı modeli (3 de olabilir) kullanılmaktadır. Biz örnek kodları burada GASM (as) kullanarak derleyeceğiz. Her ne kadar Raspberry Pi kitlerine HDMI ile monitor, USB ile de fare ve klavye bağlanabiliyorsa da uzak masaüstü çalışması çok daha kolaydır. Böylece Raspberry Pi’ı yalnızca güç kaynağına ve Ethernet kablosu ile ağa bağlamak yeterli olmaktadır. Zaten bunların server amaçlı kullanılmalarında da aynı yöntem uygulanmaktadır. Raspberry Pi 3 kendi içerisinde kablosuz ve bluetooth bağlantı modüllerine de sahiptir. O halde Raspberry Pi 3

kullanıldığı taktirde ethernet bağlantısına da gerek kalmayacaktır.



Raspberry Pi kitlerine pek çok işletim sistemi kurulabilmektedir. Bunların listesi gittikçe artmaktadır. Ancak Raspberry Pi için en gelişkin işletim sistemlerinde biri Raspbian'dır. Raspbian aynı zamanda en çok tercih edilen sistemdir.

Raspberry Pi'a işletim sistemi kurmak oldukça kolaydır. www.raspberrypi.org sitesine girilip (burası Raspberry Pi'ın resmi sitesidir. Download kısmına gelinir. Burada NOOBS paketi indirilir. Bu zip dosyası açılarak Fat32 Micro SD karta çekilir. Sonra Raspberry Pi HDMI kablo ile monitöre bağlanır. Tabii USB klavye ve fare bağlantısı da gerekebilmektedir. Sonra güç uygulandığında karşımıza bir kurulum ekranı çıkacaktır. Oradan işletim sistemi olarak Raspbian'ı seçip kuruluma başlayabiliriz. Raspian kurulduktan sonra apt-get paket yönetici pronm ile gerekli yazılımlar Internet bağlantısıyla kurulabilir. Aşağıdaki yazılımlar kursta kullanılan Raspberry Pi 2'ye sorunsuz olarak yüklenmiştir:

- Libre Office
- gcc, g++ derleyicileri
- R yorumlayıcısı ve server programı
- Python derleyicisi, yorumlayısıcı
- Eclipse
- Mono framework
- Mono Develop IDE'si
- MySql Veritabanı Yönetim Sistemi
- Apache Web Server
- SVN server
- GIT server
- Qt Framework
- Qt Creator IDE'si
- Bazı tarayıcılar
- Code Blocks, Blufish gibi editörler
- SSH server ve Putty client program
- FTP Server
- Ve diğerleri

Anahtar Notlar: Raspberry Pi'a VNC uzak masaüstü kurulumu da şöyle yapılabilir:

1) Komut satırında aşağıdaki komut uygulanır:

sudo apt-get install tightvncserver

2) Install işleminden sonra vncserver programı çalıştırılarak password girilir:

sudo vncserver

Burada bizden bir parola istenecek. Biz de bağlanırken o paraloyı kullanacağız.

3) Sonra ilgili terminali hazırlamak için Raspberry pi'da şu komut uygulanır:

vncserver -geometry 1600x900 -depth 24 :1

Geometry çözünürlüğü belirlemek için kullanılıyor. 1920x1080 yapılabiliyor. "depth" ise renk yoğunluğunu belirtiyor (24bit).

Yukarıdaki işlemler boot sırasında otomatik olarak da yapılabilir.

4) Windows makinaya VNC client program kurulur. İki önemli client program var: TightVNC Viewer ve Real VNC viewer.

5) VNC viewer da bağlantı yapılırken ip numarası ile ':' ekran numarası girilir. Örneğin:

6) Üçüncü adımı otomatize etmek için /home/pi dizininde .config/autstart dizinine geçilir:

cd .config/autostart

Sonra bir editörde (örneğin nano) tightvnc.desktop isimli dosya oluşturulur:

nano tightvnc.desktop

Dosyanın içersine şunlar yazılır:

[Desktop Entry]
Type=Application
Name=TightVNC
Exec=vncserver –geometry 1280x800 –depth 24 :1
StartupNotify=false

Eğer birden fazla sanal pencere bağlantısı kurulmak isteniyorsa aynı dosyadan farklı isimle bir tane daha oluşturmak gerekiyor (örneğin tightvnc2.desktop gibi) Dosya save edilir ve sistem reboot edilir.
Not: Ayrıca ilk açılışta LXPOLKIT ile ilgili bir uyarı mesajı görüntüleniyor. Bunu engellemek için:

sudo nano /etc/xdg/autostart/lxpolkit.desktop

ile lxpolkit.desktop dosyası açılır. Dosyanın sonundaki,

NotShowIn: GNOM;KDE

satırına LXDE de eklenir:

NotShowIn: GNOM;KDE;LXDE

VNC Server için Router port yönlendirmesi yapılırken 5900, 5901, 5902 vs. porrtlarının açılması gerekir. Eğer bağlantı :1 biçiminde 1'inci sanal desktop'tan yapılacaksa 5901, ikinci sanal desktop'tan yapılacaksa 5902 vs. portları TCP için yönlendirilmelidir. (Örneğin kursumuzda :1 sanal desktop'undan bağlantıl yapıldığı için yalnızca 3901 portu yönlendirildi.

1.5. GNU ARM Çapraz Derleyici Araçlarının Kurulması

80x86 Tabanlı Windows, Linux ve Mac OS X sistemleri için GNU'nun ARM çağraz derleyici (cross compiler) paketi vardır. Bu paket içerisinde gcc, g++ ve as gibi temel derleyiciler ve bağlayıcılar bulunur. Bu araçlar derleme ve bağlama işlemini ARM mimarisi için yapmaktadır. Bu araçların kurulumu için Google'da "GNU ARM toolchain" araması yapılabilir. https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads sayfasından paket indirilip kurulabilir. Örnek bir derleme şöyle yapılabilir:

```
arm-none-eabi-gcc  -S sample.c
```

Burada sample.s isimli ARM kodu içeren sembolik makine dili dosyası elde edilecektir.

2. 32 Bit ARM İşlemcilerinde Temel Makine Komutları

Bu bölümde 32 bit ARM İşlemcilerindeki temel makine komutları ele alınacaktır.

2.1. 32 Bit ARM İşlemcilerinin Temel Mimari Özellikleri

ARM işlemcileri RISC mimarisine göre tasarlanmışlardır. Bu nedenle genel RISC prensiplerinin pek çoğu bu işlemcilerde uygulanmıştır.

1) ARM mimarisinde makine komutları Intel mimarisine göre oldukça azdır. Yalnızca çok temel komutlar vardır ve bunlar etkin olarak işletilmektedir.

2) ARM diğer RISC işlemcilerinde olduğu gibi Load/Store tarzda mimariye sahiptir. Yani bellekten yazmaçlara (load işlei) ve yazmaçlardan belleğe (store) aktarım yapan makine komutlarının dışındaki komutlarda bellek operandı yoktur. Load/Store komutlarının dışındaki komutlar yazmaçlar arasında işlemler yapmaktadır.

3) ARM işlemcilerinde 15 temel yazmaç bulunmaktadır. Bu yazmaçlardan 3 tanesi özel işlevlere sahiptir. Ancak geri kalan 12 tanesi ile her türlü işlem yapılabilmektedir. Bu anlamda yazmaçlar arasında bir kullanım farklılığı yopktur.

4) ARM mimarisinde Load/Store komutları dışındaki komutların çoğu üç operand almaktadır. Böylece iki kaynak operand işleme sokulduğunda sonucun yerleştirilmesi için bunlardan birinin bozulmasına gerek kalmaz. Örneğin:

```
ADD R0, R1, R2
```
Burada R0 = R1 + R2 işlemi yapılmaktadır. Tabii operand'larda aynı yazmaçlar birden fazla kez kullanılabilir. Örneğin aşağıdaki komut da geçerlidir:

```
ADD R0, R0, R0
```

5) ARM işlemcileri düşük güç harcamasıyla ünlüdür. Bu nedenle zaten gömülü sistemlerde ve mobil aygıtlarda tercih edilmektedir.

6) ARM mimarisinde diğer RISC mimarilerinde olduğu gibi makine komutları eşit uzunluktadır (örneğin hep 4 byte). Komutların kodlanması (encoding) oldukça basittir. Bu durum işlemcinin komutu alıp (fetch) yorumlama (decode) sürecini hızlandırmaktadır.

7) ARM işlemcileri teemel olarak tamsayılar üzerinde işlem yapmakla birlikte bunlar aynı zamanda VFP ve NEON gibi birleşik matematik işlemcilerle kullanılabilmektedir. Yani ARM mimarisinde gerçek sayı işlemleri donansımsal olarak da yapılabilmektedir.

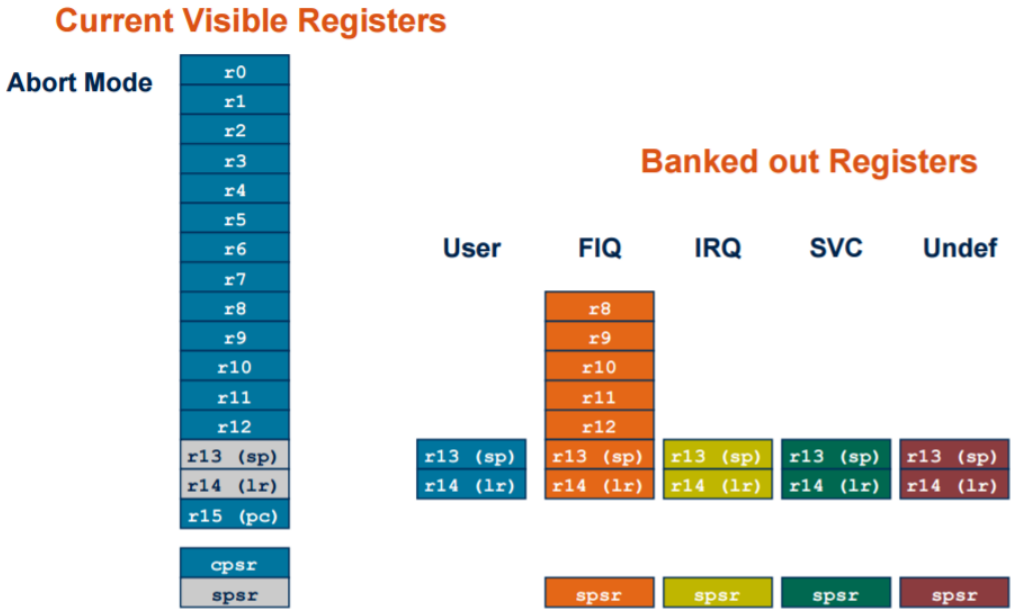
8) ARM işlemcileri hem "Big Endian" hem de "Little Endian" formatta çalışabilmektedir. İşlemci reset edildiğinde default çalışma modu "Litte Edian"dır.

9) ARM işlemcilerinde kod ve verinin hizalanması (alignment) önemlidir.

10) ARM mimarisinde komutların çoğu koşullu (condition coded) biçimdedir. Komut kodlarındaki yüksek anlamlı 4 bit koşulu belirtir. Koşullar bayraklarla ilgilidir. Eğer o andakş bayraklar komutun koşulunda belirtilen durumu karşılamıyorsa komut hiçbir etkiye yol açmaz (yani yapılmaz). Çalışma sonraki makine komutundan devam eder. Tabii koşullardan biri de AL (Always) koşuludur. Bu durum aslında koşulun olmadığı anlamına gelir. Düz komutlar aslında koşulsuz komutlardır.

2.2. 32 Bit ARM İşlemcilerinin Yazmaç Yapısı

32 bit ARM işlemcilerinin yazmaç yapısı şöyledir:

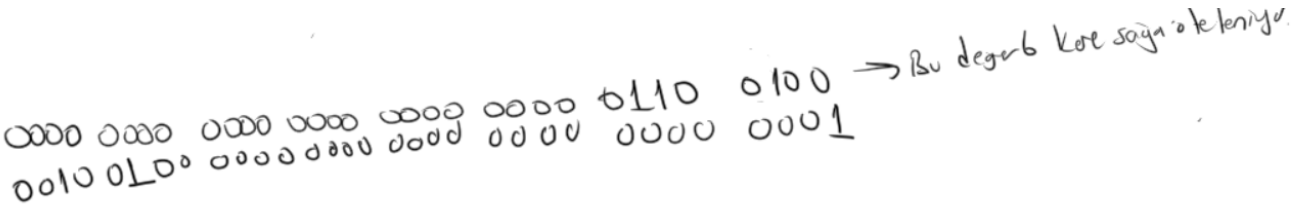


ARM mimarisinde yazmaçların isimlendirilmesi R harfinin yanına numara getirilerek yapılmaktadır. R0-R12 arasındaki 13 yazmaç genel amaçlı olarak her işlemde kullanılabilir. R13 yazmacı stack göstericisini belirtir. (Yani Intel'deki ESP gibi). R14 yazmacına "Link Register" denilmektedir. Bu yazmacın işlevi ileride ele alınacaktır. R15 yazmacı ise program sayacıdır (program counter). Yani bu yazmaç Intel'deki EIP yazmacı gibidir. ARM mimarisinde bayrak yazmacının ismi CPSR biçimindedir. Buna kısaca "Status Register" denilmektedir. Yine bu yazmaç Intel'de olduğu gibi bitlerden yani bayraklardan oluşmaktadır. Intel mimarisindeki pek çok bayrak burada da benzer biçimde vardır.

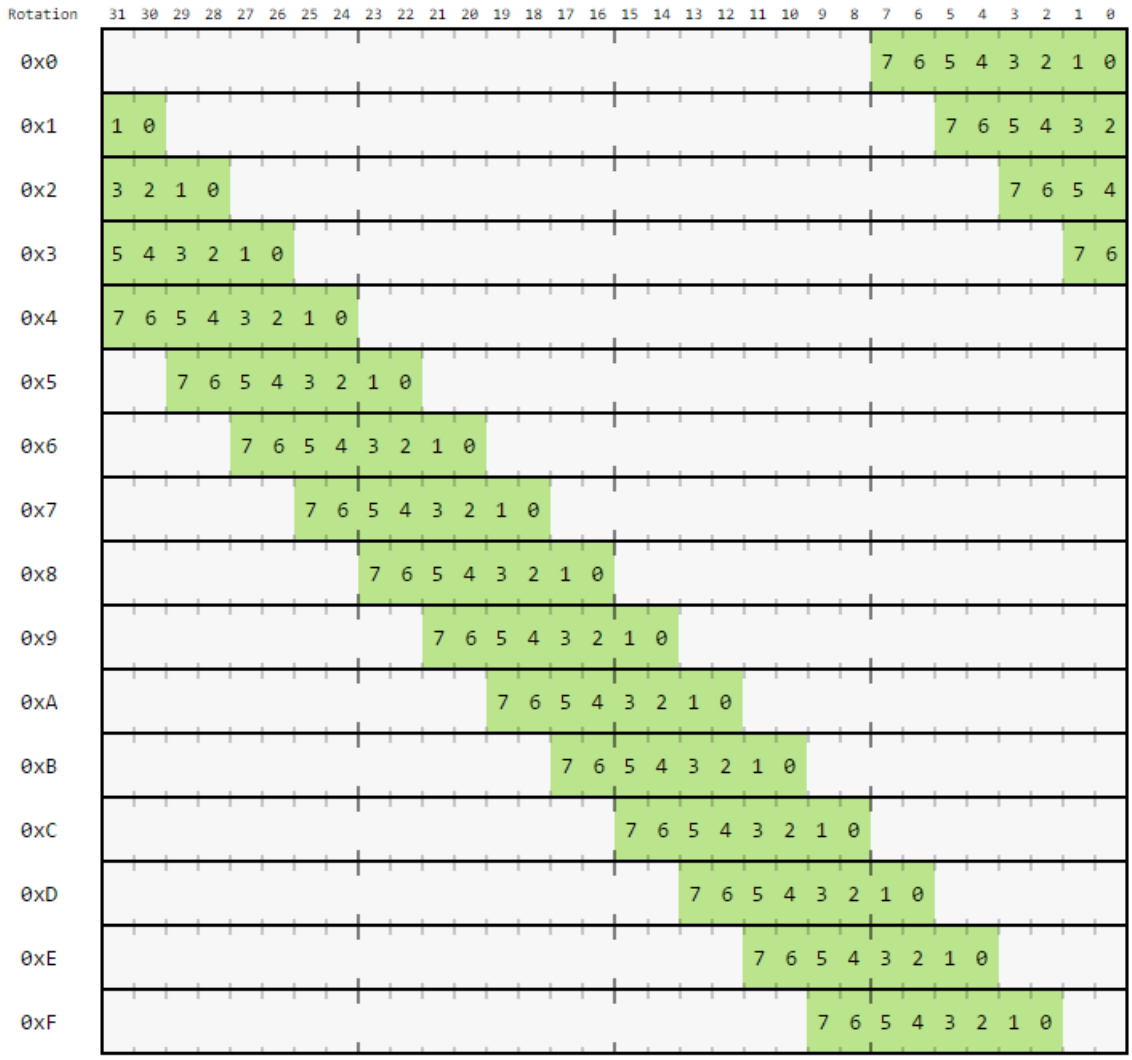
2.3. MOV Komutları

Yukarıda da belirtildiği gibi ARM tipik bir RISC işlemcisidir. Diğer RISC işlemcilerinde olduğu gibi yazmaç bellek arasında aktarım için bir grup Load/Store komutları bulunur. Diğer komutlar bellek operandı almazlar. ARM'daki MOV komutu Intel'in MOV komutuyla karıştırılmamalıdır. ARM'ın MOV komutları load/store komutlar değildir. Yani bunlar bellek ile yazmaç arasında transfer yapmazlar. MOV komutları yazmaç-yazmaç ve yazmaç-sabit (immediate) işlemlerini yapmaktadır.

MOV komutu ile bir yazmaca sabit değer atanabilir. Ancak 32 bit ARM mimarisinde tüm komutlar 32 bit (4 byte) olduğu için sabit değer komutun içerisinde 4 byte olarak kodlanamamaktadır. Bu da her değerin MOV komutlarıyla bir yazmaca atanamayacağı anlamına gelir. MOV komutunun opcode'unda sabit değer için 12 bit ayrılmıştır. Bu 12 bitin yüksek anlamlı 4 biti "sağa döndürme (rotate right)" değeridir. Düşük anlamlı 8 bit ise gerçek sabit değerdir. 8 bitlik değer sağa doğru belirtilen miktarda döndürüldükten sonra elde edilen değer yazmaca yerleştirilir. Döndürme için 4 bit ayrıldığından dolayı çift döndürmeler dikkate alınmıştır. Yani döndürme miktarındaki değer ikili çarpılır ve bu 8 bitlik değer o miktarda sağa döndürülür. Örneğin 8 bitlik değer 100 olsun ve 4 bitlik döndürme değeri de 3 olsun. Elde edilen sabit şu olacaktır:



Bu sabitin 10'luk sistemdeki karşılığı 603979777'dir.



Buradan da görüldüğü gibi biz tüm 32 bit değerleri bu algoritmayla elde edemeyiz. Ancak 16 * 256 tane değeri (üstelik bunlar da ardışıl değildir) bu algoritmayla elde edebiliriz. İşte ARM sembolik dili derleyicileri bu biçimde ifade edilmesi mümkün olmayan sabitler için derleme aşamasında error vermektedir. Örneğin aşağıdaki makine komutu ifadesi mümkün olmadığı için error oluşturacaktır.

```
MOV R0, #12345678
```

Pekiyi madem ki biz MOV komutuyla bir yazmaca istediğimiz bir sabiti yerleştiremeyebiliriz, bunu sağlamanın başka bir yolu olabilir mi? Evet bunun birkaç yolu vardır. Birincisi sabiti derleme aşamasında belleğe yerleştirip, LDR makine komutu ile onu yazmaca yerleştirmektir. ARM sembolik makine dili derleyicileri özel bir sentaksla bu işlemi programcının daha kolay yapmasını sağlamaktadır. Örneğin:

```
LDR R0, =12345678
```

Burada aslında 12345678 değeri belleğe yerleştirilip bu bellek içeriği R0 yazmacına yüklenmektedir. (Sayının solundaki '=' sembolüne dikkat ediniz.) İkincisi ise mimariye sonradan eklenmiş olan MOVW ve MOVT komutlarını kullanmaktır.

ARM mimarisine Vesiyon-6'nın sonraki sürümlerinde yazmaca sabit yükleyen MOVW ve MOVT isimli iki komut daha eklenmiştir. MOVW doğrudan 16 bit sabiti yazmaca yerleştirir. Herhangi bir döndürme işlemi yapılmaz. Örneğin:

```
MOVW    R0, #17314
```

MOVW komutu yazmacın yüksek anlamlı 16 bitini sıfırlamaktadır.

MOVT ise sabit değeri yazmacın düşük anlamlı 16 bitini bozmadan yüksek anlamlı 16 bitine yerleştirmektedir. Böylece biz 32 bit değeri istersek MOVW ve MOVT komutlarıyla iki hamlede yerleştirebiliriz. Örneğin:

```
MOVW    R0, #0x5678
MOVT    R0, #0x1234
```

işleminin sonucunda R0 yazmacının içeriisnde 0x12345678 değeri bulunacaktır. MOVW ve MOVT komutları yalnızca yazmaca sabit atamak için kullanılır.

MOV makine komutunun iki yazmaç operandlı biçimi bir yazmaçtaki değeri diğer bir yazmaca atamak için kullanılır. Örneğin:

```
MOV R3, R0
```

Burada R0'daki değer R3 yazmacına atanmıştır.

2.4. ARM İşlemcilerinde Load Store Komutları ve Adresleme Modları

ARM mimarisinde yazmaç ve bellek arasında atama yapan iki komut vardır: LDR (Load Register) ve STR (Store Register). LDR bellekten yazmaca atama yaparken, STR yazmaçtan belleğe atama yapmaktadır. ARM sembolik makine dilinde bellek operandları diğer sembolik makine dillerinde olduğu gibi yine köşeli parantezlerle gösterilmektedir. Köşeli parantez içerisine yalnızca sabit değer yerleştirilemez. Köşeli parantez içerisine yerleştirilecek öğelerin iki temel kalıbı vardır. Genel olarak ARM'ın sembolik makine dilinde köşeli parantezin içerisine toplama anlamında '+' karakteri getirilmez. Bunun yerine köşeli parantez içerisindeki öğeler ',' karakteriyle ayrılır. Bu da aslında toplama anlamına gelmektedir. Şimdi bu bu kalıpları (ardes modlarını) tek tek ele alalım.

1) Yazmaç + Sabit Kalıbı: Bu kalıpta bellek operandı yazmaç ve 12 bitlik pozitif ya da negatif bir sabitin toplamından oluşur. ([-4095, +4095]) Örneğin:

```
LDR     R0, [R1, #8]
```

Burada R1 + 8 değeri ile belirtilen bellek bölgesindeki 32 bit değer R0 yazmacına atanmaktadır. Örneğin:

```
LDR     R0, [R1]
```

Bu komut aşağıdaki eşdeğerdir:

```
LDR     R0, [R1, #0]
```

Load/Store komutlarının ön ve son indeksli (pre-indexed, post-indexed) biçimleri de vardır. Ön indeksli, biçimlerde '!' karakteri köşeli parantezin sonuna yerleştirilir:

```
LDR     R0, [R1, #8]!
```

Son indeksli (post-indexed) biçimde ise sabit değer köşeli parantezin dışına yazılır:

```
LDR     R0, [R1], #8
```

O halde normal biçimle birlikte load/store komutlarının üç farklı varsyasyonu olduğunu söyleyebiliriz:

```
LDR     R0, [R1, #8]       Normal biçim
LDR     R0, [R1, #8]!      Ön indeksli biçim
LDR     R0, [R1], #8       Son indeksli biçim
```

Normal biçimde R1 + 8 adresindeki 32 bit değer R0 yazmacına atanır. Bu atama işleminden sonra R1'in değeri değişmez. Ön indeksli biçimde R1 + 8 adresindeki 32 bit değer R0 yazmacına yerleştirildikten sonra R1'in değeri 8 artırılmaktadır. Son indeksli biçimde ise R1 adresindeki 32 bit değer R0'a atandandıktan

sonra R1'in içerisindeki değer 8 artırılmaktadır. Ön indeksli biçim ile son indeksli biçim arasındaki fark ön indeksli biçimde toplam değere ilişkin adrese erişilmesi, son indeksli biçimde ise yalnızca yazmacın belirttiği adrese erişilmesidir. Her iki biçimde de köşeli parabtez içerisindeki yazmaç işlem sonucunda artırılmaktadır. Örneğin R1 yazmacının bir int dizinin adresini gösterdiğini düşünelim:

```
LDR     R0, [R1], #4
```

Bu komut bir döngü içerisinde uygulanırsa her defasında diziinn bir sonraki elemanı R0 yazmacına çekilecektir. Aynı işlemi Intel mimarisinde yapacak olsaydık köşeli parantez içerisindeki yazmacı ayrıca artırmamız gerekirdi.

Komutun sabit kısmı negatif bir değer de olabilir. Örneğin

```
LDR R0, [R1, #-16]
```

Köşeli parantez içerisindeki yazmaç PC de olabilir. Bu durumda PC'den göreli bir bellek bölgesinin içeriği yazmaca yüklenecektir. Örneğin:

```
LDR     R0, [PC, #8]
```

Burada işlenen komutun 8 byte ilerisindeki 4 byte'lık değer R0 yazmacına aktarılır. Ancak burada ince bir nokta söz konusudur. Pek çok işlemcide Program sayacı (Intel'deki EIP, ARM'daki PC) komutun alınması sırasında ("fetch" işlemi sırasında) artırılmaktadır. Böylece program sayacı bir komut çalışırken o komutun başlangıç adresini değil sonraki komutun başlangıç adresini gösteriyor durumdadır. (Zaten Intel'de program sayacına göreli jump komutlarında orijin noktasının sonraki komut olmasının nedeni budur.) Fakat ARM işlemcilerinde geçmişe doğru uyumu da korumak için program sayacı (PC yazmacı) çalıştırılan komuttan sonraki komutu değil ondan sonraki komutu göstermektedir. Başka bir deyişle ARM işlemcilerinde program sayacı sonraki komutun yerini değil iki sonraki komutun yerini göstermektedir. Dolayısıyla da komut aslında program sauyacının gösterdiği yerden değil onun bir öncesinden alınır. 32 bit ARM işlemcilerinde komutlar da hep 4 byte olduğu için program sayacı (PC) o anda çalıştırılan komutun 8 byte ilerisini gösterir durumda olmaktadır. Örneğin:

```
LDR     R0, [PC, #XXX]
<Komut>
number: .word  100
```

Burada number adresindeki değerin R0'a yüklenmesi için XXX değerinin kaç olması gerekir? Yanıt 0. Çünkü LDR komutu çalışırken PC yazmacı zaten 8 byte ileriyi yani number adresini göstermektedir.

```
LDR     R0, [PC, #0]
<Komut>
number: .word  100
```

ARM mimarisinde Intel'de olduğu gibi köşeli parantez içerisinde yalnızca bir sabit (bir adres) getirilememektedir. Bu mimaride en azından köşeli parantez içerisinde bir yazmaç + sabit olmak zorundadır. Fakat bu yazmaç da PC olabilmektedir.

Pekiyi o halde bu mimaride belli bir adresteki, bilgi yazmaca nasıl yüklenecektir? İşte bu işlem için LDR komutunun bu kalıbı kullanılır. ARM sembolik makine dili derleyicileri LDR komutunda ikinci operand olarak bir etiket gördüğünde onun köşeli parantez içerisinde PC + göreli uzaklık olduğunu varsaymaktadır. Yani örneğin:

```
number: .word 100
...
LDR     R0, number
```

Komutu sembolik makine dili derleyicisi için aşağıdakiyle eşdeğerdir:

```
number: .word 100
```

```
...
LDR R0, [PC, #göreli uzaklık]
```

Ancak ARM sembolik makine dili derleyicilerinin bu komutu kabul etmesi için sembolün (etiketin) aynı bölüm (section) içerisinde ve 4096 byte yakınlarda olması gerekmektedir. Eğer sembol aynı bölüm (tipik olarak .text bölümü) içerisinde değilse ARM derleyicileri bu sembolün belirttiği adresin bağlama sırasında 4096'nın dışında kalabileceği şüphesiyle error oluşturmaktadır.

Pekiyi aynı bölümde olmayan ya da aynı bölümde olduğu halde 4096 yakınlıkta olmayan sembol adreslerini nasıl yazmaçlara yerleştirebiliriz? İşte bu işlem ttipik olarak iki aşamada yapılmaktadır. Birinci aşamada adresi yerleştilecek sembolün adresi aynı bölümde 4096 yakınlıkta bir adrese yerleştirilir ve LDR komutuyla o adresten yükleme yapılır. ARM sembolik makine dili derleyicileri "= <etiket>" sentaksı kullanıldığında bunu zaten otomatik olarak yapmaktadır. Örneğin:

```
.arm

.data
             .word 1, 2, 3, 4, 5
    number: .word100
.text
foo:
    ldr r0, =number
    bx  lr
```

Burada sembolik makine dili derleyicisi aslında number sembolünün adresini komutun 4096 yakınlığında bir yere yerleştirip oradan PC göreli LDR ile yüklemeyi yapmaktadır. Yukarıdaki kod derlendikten sonra objdump -d ile disassembly çıktısı şöyle olacaktır:

```
00000000 <foo>:
   0:   e51f0000        ldr     r0, [pc, #-0]   ; 8 <foo+0x8>
   4:   e12fff1e        bx      lr
   8:   00000014        .word   0x00000014
```

Tabii burada aslında R0 yazmacına yüklenen değer number sembolünün adresidir, bizim bşr LDR daha kullanıp oradaki değeri yüklelemiz gerekir:

```
LDR     R0, [R0, #0]
```

O halde genel olarak biz bir sembolün (etiketin) belirttiği adresteki bilgiyi aşağıdaki kalıpla bir yazmaca yükleyebiliriz:

```
LDR     R0, =symbol
LDR     R0, [R0]
```

2) Yazmaç + Yazmaç + Ötelemeli Sabit: Köşeli parantez içerisinde iki yazmacın toplamı offset belirtebilmektedir. Ancak ikinci yazmaç ayrıca ötelemeye de sokulabilmektedir. Bu kalıba uygun komutların birkaç örneği şöyle olabilir:

```
LDR     R0, [R1, R2]         ; öteleme miktarı kullanılmamış
LDR     R0, [R1, R2, LSL 2]  ; R2'nin değeri 2 kez sola ötelendikten sonra R1 ile toplanıyor
```

Bu kalıpta iki yazmacın toplamı bir adres belirtmektedir ve o adresteki bilgi yazmaca yüklenmektedir. Ancak ikinci yazmaç istenirse belli miktarda öteleme işlemine sokulabilir. İkinci yazmacın sokulabileceği işlemler şunlardır:

LSL (Sola işaretsiz öteleme, yani ikinin kuvvetiyle çarpma)
LSR (Sağa işaretsiz öteleme, yani, ikinin kuvvetiyle bölme)
ASR (Sağa işaretli öteleme)
ROR (Sağa döndürme, CF işleme karışmaz)
RRX (Sağa bir kez döndürme, CF işe karışır)

LSL, LSR, ASR ve ROR işleminin miktarı 0-31 olmaktadır. LSL işleminde miktar sıfır ise bu durum herhangi bir öteleme işleminin yapılmayacağı anlamına gelir. RRX'te miktar belirtilmez, bu seçenekte zaten sağa döndürme bir kez yapılır. Şimdi birkaç örnek verelirm:

```
LDR     R0, [R1, R2, LSL 4]
```

Burada R1 + R2 * 16 adresinden çekilen 4 byte R0 yazmacına yüklenmektedir. Örneğin:

```
LDR     R0, [R1, R2, LSR 2]
```

Burada R1 + R2 / 4 adresinden çekilen 4 byte R0 yazmacına yüklenmektedir. Örneğin:

```
LDR     R0, [R1, R2]
```

Burada R1 + R2 adresinden çekilen 4 byte R0 yazmacına yüklenmektedir. Görüldüğü gibi ikinci yazmaca herhangi bir öteleme ya da döndürme uygulanması zorunlu değildir.

Tabii yukarıdaki LDR komutlarının hepsi normal, ön indeksli ya da son indeksli biçimde kullanılabilir. Örneğin:

```
LDR     R0, [R1, R2, LSR 2]!
```

Burada önce R1 + R2 * 4 adresindeki 4 byte değer R0 yazmacına yerleştirilecek ve işlemden sonra R1 yazmacı (base register) R2 * 4 kadar artırılacaktır. Örneğin:

```
LDR     R0, [R1], R2, LSR 2
```

Burada R1 adresindkei 4 byte R0 yazmacına yerleştirilecek sonra da R2 * 4 değeri R1 yazmacına yerleştirilecektir.

STR komutlarının genel yapısı LDR komutlarıyla aynıdır. Komutun sembolik biçiminde yine yazmaç sol tarafta bellek operandı sağ tarafta tutulur (yani LDR'de olduğu gibi) yalnızca işlemin yönü terstir. Yani yazmaçtan belleğe atama yapılmaktadır. Örneğin:

```
STR     R0, [R1]
```

Burada R0 yazmacındaki bilgi R1 yazmacı ile belirtilen adrese aktarılmaktadır. Komutun eşdeğeri şöyledir:

```
STR     R0, [R1, #0]
```

Örneğin:

```
STR     R0, [R1, R2]
```

Burada R0'daki değer R1 + R2 adresine aktarılmaktadır. Komutlar yine normal, ön indeksli ya da son indeksli olabilmektedir.

LDR ve STR komutları 32 bitlik veriler üzerinde aktarım yapmaktadır. Fakat 32 bit ARM işlemcilerinde istenirse 8 bit (1 byte), ve 16 bit (2 byte) aktarımlar da yapılabilmektedir. 8 bit işlemler LDRB, LDRSB ve STRB komutlarıyla 16 bit işlemler de LDRH, LDRSH ve STRH komutlarıyla yapılmaktadır. Bu komutların genel biçimi LDR ve STR komutlarıyla aynıdır. Yalnızca işlem genişliği farklıdır. LDRB, LDRSB ve STRB komutlarındaki 'B' harfi "Byte" sözcüğünden, LDRH, LDRSH ve STRH komutlarındaki 'H' harfi ise "Half Word" sözcüklerinden kısaltılmıştır.

Anahtar Notlar: ARM mimarisinde "word" sözcüğü 4 byte'ı yani bir yazmaç genişliğini belirtmektedir. Böylece "half word" 16 bit anlamına gelir. "Double word" ise 64 bit anlamına gelmektedir.

LDRB ve LDRH komutları işaretsiz yükleme yapmak için, LDRSB ve LDRSH komutları da işaretli yükleme yapmak için kullanılmaktadır. İşaretsiz yüklemede değer yazmaca çekilirken yüksek anlamlı bitler '0' ile, işaretli yüklemede işaret biti ile doldurulmaktadır.

Örneğin:

```
LDRB    R0, [R1]
```

Burada komut işaretsiz olduğu için R1 yazmacı ile belirtilen adresteki 1 byte R0 yazmacının en düşük anlamlı byte'ına atanmaktadır. R0 yazmacının yüksek anlamlı 3 byte'ı sıfırlanacaktır. Böylece R0'ın içerisinde R1 adresindeki 1 byte değer 32 bit halinde gözükecektir. Örneğin:

```
STRH    R0, [R1, R2]
```

Burada R0 yazmacındaki düşük anlamlı 16 bitlik değer R1 + R2 adresine 16 bit olarak aktarılmaktadır. Yine komut işaretsiz olduğu için R0 yazmacının yüksek anlamlı 2 byte'ı sıfırlanacaktır.

LDRSB ve LDRSH komutlarında bellekten yüklenen değerler sanki işaretli tamsayı sistemindeymiş gibi değerlendirilmektedir. Böylece yüklenen yazmacın yüksek anlamlı bitleri işaret dbitleriyle (eğer değer negatifse '1' ler ile sayı pozitifse '0' bitleri ile) doldurulur. Örneğin:

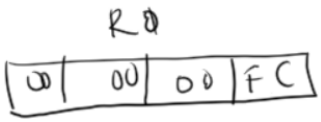
```
LDRSB   R0, [R1]
```

Bu komutta R1 adresindkei değerin 0xFC olduğunu varsaylım. Bu FC değeri işaretsiz sistemde +252, işaretli sistemde -4 olur. İşte komut LDRSB (yani işaretli byte yükleme) komutu olduğu için R0 yazmacı 32 bit olarak -4 biçiminde olacak halde aşağıdaki gibi yüklenecektir:

Bu komut işaretsiz biçimde uygulanmış olsun:

```
LDRB    R0, [R1]
```

Artık R0'ın yüksek anlamlı bitleri 0'larla beslenecektir:



Yukarıdkai anlatımdan da görüldüğü gibi STRB ve STRH komutlarının işaretli biçimleri yoktur. STRB komutu her zaman yazmacın düşük anlamlı byte'ını, STRH ise düşük anlamlı iki byte'ını bire bir belleğe aktarır.

ARM mimarisinde load/store komutlarının dışındaki diğer aritmetik ve mantıksal komutların "byte" ve "half word" biçimleri yoktur. Bu komutlar her zaman 4 byte (32 bit) işlem yaparlar.

Komutlardaki Koşullar

Giriş kısmında da belirtildiği gibi ARM mimarisinde neredeyse her komutun bir koşul kısmı vardır. Yani komutlar istenirse yalnızca bazı bayrak koşulları sağlandığında çalıştırılabilir. Bu özellik geenel olarak Intel gibi CISC tarzı işlemcilerde bulunmamaktadır. Örneğin bir ADD komutu bayraklar uygun konumda değilse çalıştırılmaz akış sonraki komuta geçer. Koşullar komutların yüksek anlamlı 4 bitinde kodlanmaktadır. Örneğin aşağıda ADD komutunun ikilik sistemdeki karşılığı görülmektedir:

31 30 29 28	27 26	25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0	0	0 1 0 0	S	Rn	Rd	imm5	type	0	Rm

Toplam 16 koşul aşağıda listelenmiştir:

Table A8-1 Condition codes

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) [a]	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS [b]	Carry set	Greater than, equal, or unordered	C == 1
0011	CC [c]	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) [d]	Always (unconditional)	Always (unconditional)	Any

a. Unordered means at least one NaN operand.
b. HS (unsigned higher or same) is a synonym for CS.
c. LO (unsigned lower) is a synonym for CC.
d. AL is an optional mnemonic extension for always, except in IT instructions. For details see *IT* on page A8-390.

Komutta hiçbir koşul belirtilmeye de bilir (Cond bitlerinin 1110 olduğu durum). Bu durumda komut her zaman çalıştırılır. Koşullar sembolik makine dilinde komutun yanına bitişik olarak yukarıda koşul belirten soneklerin getirilmesiyle oluşturulurlar. Örneğin ADDGE, ADDCS, ADDLE gibi. Gerçekten de ARM mimarisinde aslında jump komutları da koşul özelliğine sahip olduğu için tek bir jump komutu (B(ranch) komutu) koşullu dallanma için yeterli olmaktadır. Koşullu komutlar pek çok RISC mimarisinde (örneğin MIPS'te de) benzer biçimde bulunmaktadır. Bazı durumlarda kodun çok daha etkin yazılmasına olanak sağlamaktadır.

Komutların Bayrakları Etkileyip Etkilememesi Durumu

ARM mimarisinde bazı RISC mimarilerinde de olduğu gibi çeşitli komutların bayrak yazmacını (STATUS yazmacı) etkileyip etkilemeyeceği komutun içerisinde verilebilmektedir. Eğer sembolik makine dilinde komutun sonuna (fakat koşul kısmından önce 'S' karakteri getirilirse bu komutun bayrak yazmacını etkileyeceği anlamına gelir. Eğer 'S' karakteri getirilmezse komut bayrak yazmacını etkilemez. Örneğin:

```
ADD     R0, R1, R2      ; komut bayrakları etkilemeyecektir
ADDS    R0, R1, R2      ; komut bayrakları etkileyecektir.
ADDLT   R0, R1, R2      ; komut çalışırsa bayrakları etkilemeyecektir
ADDSLT  R0, R1, R2      ; komut çalışırsa bayrakları etkileyecektir.
```

Komutların bayrakları isteğe bağlı olarak etkileyip etkilememesi kodun daha etkin düzenlenmesine olanak sağlamaktadır. Böylece baayarı etkileyen bir işlemden sonra hemen bayrakların durumuna bakılması gerekmez. Bayrakları bozmayan çeşitli komutlardan sonra bayraklara bakılabilir. Anımsanacağı gibi Intel gibi CISC mimarilerinde komutlar her zaman bayrakları etkilemektedir.

Bazı komutlar diğer mimarilerde olduğu gibi zaten bayrakları doğal olarak etkilememektedir (örneğin LDR, STR komutları gibi). Bu tür komutların sonuna 'S' getirilmez zaten getirilmesinin anlamı da yoktur. Bazı

komutlar ise bayrakları her zaman etkilemektedir (CMP komutunda olduğu gibi). Bunların sonuna yine ‘S’ getirilmez. Ancak sanki S getirilmiş gibi düşünülebilir.

ARM Dokümanlarında Komutların Sembolik Makine Dilindeki Gösterimleri

ARM mimarisi hakkında pek çok kitap ve dokğman bulunmakla birlikte şüphesiz teknik bakımdan en öneml, ve kesin olan ARM firmasının kendi dokümanlarıdır. ARM işlemcilerinin A serisi ve R serisi için V7 komut kümesi ve mimarisi “ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition” dokğmanında açıklanmıştır. Bu dokümanda gerekli her bilgi biraz kompakt biçimde dokğmante edilmiştir. Doküman içerisinde tek tek makine komutları ele alınmış ve bunların sembolik makine dili temsilleri de belirtilmiştir. Bu dokğmanlarda komutların sembolik makine dili temsilleri aşağıdaki örnekte olduğu gibidir:

A8.8.7 ADD (register, ARM)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}
```

31 30 29 28	27 26 25	24 23 22 21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 0 0	0 1 0 0	S	Rn	Rd	imm5	type	0	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

```
if Rd == '1111' && S == '1' then SEE SUBS PC, LR and related instructions;
if Rn == '1101' then SEE ADD (SP plus register);
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = (S == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
```

Burada komut şöyle ifade edilmiştir:

```
ADD{S}<c> <Rd>, <Rn>, <Rm>{, <shift>}
```

Burada kğüme parantezi içerisindkei öğeler isteğe bağlı (optional) olanlardır. Yani yukarıdaki örnekte komutun ana kısmından sonra ‘S’ harfinin getirilip getirilmeyeceği isteğe bağlıdır. Benze rbiçimde ikinci yazmaçta bit operasyonları da isteğe bağlıdır. Açısal parantezler zorunlu kısımları belirtir. Örneğin komutun <c> kısmı koşul eklerini belirtir. Her ne kadar bu koşul kısmı zorunlu biçiminde belirtildiyse de koşulun kendi içerisinde “koşulun bulunmama” durumu da vardır. Rd sözcüğündeki ‘d’ harfi “destination”dan gelmektedir. Örneğin:

Encoding A1 ARMv4*, ARMv5T*, ARMv6*, ARMv7

```
LDR<c> <Rt>, [<Rn>,+/-<Rm>{, <shift>}]{!}
LDR<c> <Rt>, [<Rn>],+/-<Rm>{, <shift>}
```

31 30 29 28	27 26 25	24	23	22	21	20	19 18 17 16	15 14 13 12	11 10 9 8 7	6 5	4	3 2 1 0
cond	0 1 1	P	U	0	W	1	Rn	Rt	imm5	type	0	Rm

For the case when cond is 0b1111, see *Unconditional instructions* on page A5-216.

```
if P == '0' && W == '1' then SEE LDRT;
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = (P == '1');  add = (U == '1');  wback = (P == '0') || (W == '1');
(shift_t, shift_n) = DecodeImmShift(type, imm5);
if m == 15 then UNPREDICTABLE;
if wback && (n == 15 || n == t) then UNPREDICTABLE;
if ArchVersion() < 6 && wback && m == n then UNPREDICTABLE;
```

Burada LDR komutunun komutunun yazmaçlı biçiminin sembolik makine dilindeki genel biçimi şöyle ifade edilmiştir:

```
LDR<c> <Rt>, [<Rn>,+/-<Rm>{, <shift>}]{!}
LDR<c> <Rt>, [<Rn>],+/-<Rm>{, <shift>}
```

Temel Aritmetik Komutlar

Anımsanacağı gibi ARM mimarisinde (genel olarak RISC mimarilerinde) Load/Store komutlarının dışındaki komutlar bellek operandı almamaktadır. Bu komutlar genel olarak üç yazmaç operandına sahiptir. Burada temel komutlar ele alınacaktır. Burada ele alınacak tüm komutların bir 'S' kısmı ve koşul kısmı vardır. Ancak örneklerde bu kısımlar kullanılmayacaktır.

ADD Komutu

ADD komutu toplama işlemi yapar. Komutun genel biçimleri şöyledir:

```
ADD{S}{<c>}{<q>} {<Rd>,} <Rn>, #<const>
ADD{S}{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

Komuttaki sabit kısım 12 bit işaretsiz bir değer belirtir. [0, 4095]). Komutun ikinci biçimi iki yazmaç toplamına ilişkindir. Ancak yazmaçlardan biri işlemden önce bit işlemine sokulabilir.

ARM mimarisinde INC ve DEC gibi komutlar yoktur. Dolayısıyla örneğin bir yazmacın içerisindeki değeri 1 artırma işlemi şöyle yapılabilir:

```
ADD R0, R0, #1
```

Bir yazmaçtaki değere 4095'ten daha büyük bir değeri eklemeye çalışalım. Bunu nasıl yapabiliriz? Eğer değer 4095'e yakınsa birden fazla ADD komutu ile işlem etkin biçimde yapılabilir. Değilse bu sabit değerin .text bölümünde yakın bir yere yerleştirilmesi ve oradan LDR ile yükleme yapılması sonra ADD kullanılması uygun olur. Örneğin R0'daki değeri 500000 artırmak isteyelim:

```
LDR     R1, [pc, #göreli uzaklık]
ADD     R0, R0, R1
...
val:    .word 500000
```

Anımsanacağı gibi sembolik makine dili derleyicileri LDR komutunun sahte (pseudo) bir biçimini de kabul etmektedir. LDR komutunda ikinci operand bir etiket ise derleyici aslında PC yazmacına göreli bir biçimde oradaki değeri yazmaca yüklemektedir. Yani yukarıdaki işlem sembolik makine dilinde şöyle de yazılabilir:

```
LDR     R1, val
ADD     R0, R0, R1
...
val:    .word 500000
```

ADD komutunun ayrıca CF bayrağını işin içine katan ADC isimli bir biçimi de vardır.

SUB KOMUTU

SUB komutunun genel biçimi de ADD ile aynıdır:

```
SUB{S}{<c>}{<q>} {<Rd>,} <Rn>, #<const>
SUB{S}{<c>}{<q>} {<Rd>,} <Rn>, <Rm>, <type> <Rs>
```

Komut bir yazmacın içerisindkei değeri diğer yazmaçtan ya da sabitten çıkartarak hedef yazmaca yerleştirir. Örneğin:

```
SUB     R0, R1, R2
```

Bu komut R0 = R1 - R2 işlemini yapmaktadır.

Örneğin R0 yazmacındaki değeri şöyle 1 eksiltebiliriz:

```
SUB     R0, R0, #1
```

SUB komutunun ayrıca CF bayrağını da işin içine katan SBC biçimi vardır.

MUL Komutu

Bu komutun sabitli biçimi yoktur. Genel biçimi şöyledir:

```
MUL{S}{<c>}{<q>} <Rd>, <Rn>{, <Rm>}
```

Sembolik makine dilinde yazım kolaylığı sağlamak için komut sanki iki operand'lıymış gibi de ifade edilebilmektedir. Bu durumda üçüncü operand'ın hedef operand'la aynı olduğu kabul edilir. MUL komutunda yazmaçlardan biri üzerinde bit işlemi yapılamamaktadır. Örneğin:

```
MUL R0, R1, R2
```

Burada R0 = R1 * R2 işlemi yapılmaktadır. Örneğin:

```
MUL R0, R1, R0
```

Burada R0 = r1 * R0 işlemi yapılmaktadır. Bu komut daha sade olarak şöyle de yazılabilmektedir:

```
MUL R0, R1
```

MUL işlemi hem işaretli hem de işaretsiz değerlerde sorunsuz çalışabilmektedir. Yani biz bu komutun operand'larının her ikisi pozitif, her ikisi negatif ve biri pozitif biri negatif olabilir. Sonucun işareti uygun biçimde oluşmaktadır. Intel x(6 mimarisinde çarpma işleminin işaretli ve işaretsiz bçimlerinin ayrı makine komutları ile yapıldığını anımsayınız.

SDIV ve UDIV Komutları

Bölme işleminin işaretli ve işaretsiz biçimleri ayrı komutlar olarak bulunmaktadır. Bu komujtların genel biçimi şöyledir:

```
SDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
UDIV{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

Örneğin:

```
UDIV    R0, R1, R2
```

Burada R0 = R1/R2 işlemi yapılmaktadır. Yine ikinci operand üzerinde bit işlemi yapılamamaktadır.

Bit İşlemleri Yapan Komutlar

ARM'da bit işlemleri yapan komutların genel biçimleri ayınıdır:

```
<Bit Komutu>{S}{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
<Bit Komutu>{S}{<c>}{<q>} {<Rd>,} <Rn>, <Rm>
```

ARM temel bit komutları şunlardır:

Komut	Anlamı
AND	Bit AND işlemi yapar
ORR	Bit OR işlemi yapar
EOR	Bir EXOR işlemi yapar
BIC	İkinci operandın tersi ile AND işlemi yapılır. Örneğin

	BIC R0, R1, R2 komutunda R2'nin bitsel tersi alınrak R1 ile AND işlemine sokulur ve sonuç R0 yazmacına atanır.
ORN	İkinci operandın tersi ile OR işlemi yapılır. Örneğin ORN R0, R1, R2 komutunda R2'nin bitsel tersi alınrak R1 ile OR işlemine sokulur ve sonuç R0 yazmacına atanır.
RBIT	Bitleri ters yüz eder. Yani 0'ıncı bit 31'inci , 1'inci bit 30;'uncu bit vs. haline getirilir (strrev işlemi gibi)
BFC	Belli bit bit pozisyonundan başlayarak yan yana n tane biti 0'lamak için kullanılır.
BFI	Bir yazmaçtaki 0'ıncı bitten başlayarak n tane biti diğer yazmacın herhangi bir pozisyonuna yerleştirmek amacıyla kullanılır.
CLZ	Bir yazmaçtaki düşükten yükseğe doğru ilk 1 olan bitin numarasını verir.

ROR komutu sağa n kez döndürme işlemini yapar. ARM mimarisinde sola döndürme yoktur. Zaten sola N defa döndürme sağa 32 - N defa döndürmeyle aynı anlama gelir. RXX işin içine CF bayrağını katarak sağa 1 kez ötelme yapar (Intel'deki RCR gibi). Intel'de olduğu gibi ARM mimarisinde de sola ötelemenin işaretli ve işaretsiz biçimleri yoktur. Fakat sağa ötelemenin vardır. LSL komutu (Logical Shift Left) sola ötelem için, LSR (Logical Sghift Right) komutu sağa işaretsiz öteleme için ve ASR ise sağa işaretli öteleme yapmaktadır.

GNU Sembolik Makine Dili Derleyicisinde Özel Yazmaç İsimleri

GNU'nun sembolik makine dili derleyicilerinde dört yazmaca okunabilirliği iyileştirmek için özel isimler de verilmiştir:

R15 = PC (Program Counter)
R14 = LR (Link Register)
R13 = SP (Stack Pointer)
R11 = FP (Frame Pointer)

R15 (PC) işlemci tarafından komutlar işletildikçe değer almaktadır. R14 (LR) yazmacı yalnızca BL komutunda (yani CALL işleminde) etkili olur. Bunun dışında R14 yazmacının diğer yazmaçlardan bir farkı yoktur. R13 (SP) genel olarak stack göstericisi amacıyla kullanılmaktadır. Ancak stack göstericisi için bu yazmacın kullanılması zorunlu değildir. Fakat ARM çağırma biçimine (calling convention) stack göstericisi için önerilen yazmaç budur. R11 (FP) stack'ten bilgi çekmek için yani yerel değişkenlere erişmek için kullanılmaktadır. Bu yazmaç Intel'deki EBP yazmacına benzetilebilir. Ancak ARM mimarisinde yerel değişkenlere erişmek için bu yazmacın kullanılması zorunlu değildir. Bu nedenle bu yazmacn da diğer yazmaçlardan aslında bir farkı yoktur. Ancak ARM çağırma biçiminde (ARM calling convention) yerel değişkenlere erişmek için R11 (FP) yazmacının kullanılması gerekmektedir.

Dallanma (Branch) İşlemleri

ARM mimarisinde akışın başka bir yere aktarılması "jump" terimiyle değil "branch" terimiyle ifade edilmektedir. Bu mimaride dallanma işlemini yapan temelde B isimli tek bir makine komutu vardır. Zaten ARM mimarisinde yalnızca dallanma komutlarının değil neredeyse tüm komutların bir koşul kısmı bulunmaktadır. Dolayısıyla B komutunun yanına daha önceden de ele aldığımız aşağıdaki eklerden biri getirilebilir:

Code	Meaning (for `cmp` or `subs`)	Flags Tested
`eq`	Equal.	`Z==1`
`ne`	Not equal.	`Z==0`
`cs` or `hs`	Unsigned higher or same (or carry set).	`C==1`
`cc` or `lo`	Unsigned lower (or carry clear).	`C==0`
`mi`	Negative. The mnemonic stands for "minus".	`N==1`
`pl`	Positive or zero. The mnemonic stands for "plus".	`N==0`
`vs`	Signed overflow. The mnemonic stands for "V set".	`V==1`
`vc`	No signed overflow. The mnemonic stands for "V clear".	`V==0`
`hi`	Unsigned higher.	`(C==1) && (Z==0)`
`ls`	Unsigned lower or same.	`(C==0) || (Z==1)`
`ge`	Signed greater than or equal.	`N==V`
`lt`	Signed less than.	`N!=V`
`gt`	Signed greater than.	`(Z==0) && (N==V)`
`le`	Signed less than or equal.	`(Z==1) || (N!=V)`
`al` (or omitted)	Always executed.	None tested.

Örneğin biz “işaretli olarak küçükse dallan” işlemi için BLT komutunu, “eşitse dallan” işlemi için BEQ komutunu kullanabiliriz. Bu koşulların aslında her komuta uygulanabileceğine dikkat ediniz. Örneğin:

```
BEQ         label
ADDEQ       R0, R1, R2
```

BEQ ZF bayrağı 1 ise ilgili adrese dallanır değilse zaten işleme girmez akış sonraki komutla devam eder. Aslında ADDEQ komutu da benzer biçimde çalışmaktadır. Yani ZF bayrağı 1 ise bu toplama işlemi yapılır, 0 ise akış sonraki komuta geçer.

Anımsanacağı gibi koşullu komutlarda koşul hiç olmayabilir. Bu durumda sembolik makine dili komutunda koşul hiç belirtilmez ya da AL (Always) soneki ile belirtilir. Yani örneğin ADD komutu ile ADDAL komutu eş anlamlıdır. Benzer biçimde B komutu ile BAL komutu da eşdeğerdir. B komutu koşulsuz bir biçimde dallanma yapmaktadır.

B dallanma komutu 24 bit işaretli yer değiştirme (displacement) miktarı almaktadır. Yani tıpkı Intel’de olduğu gibi dallanılacak yer her zaman o anda işletilen komuta (PC yazmacına) göreli uzaklıktır. (ARM’da PC (Program Counter) yazmacının çalışmakta olan komutun iki sonrasındaki komutu gösterdiğini anımsayınız.) Budurumda yer değiştirme miktarı [-8388608, 8388607] aralığındadır. Ancak bu yer değiştime miktarı byte cinsinden değil 4 byte cinsindendir. Başka bir deyişle gerçek yer değiştirme miktarı komutta belirtilen sayının 4 ile çarpılmasıyla elde edilir. Bu durumda byte cinsinden yer değiştirme miktarı [-33554432, +33554431] aralığında olur. Örneğin aşaüıdaki gibi bir sembolik makine kodu bukunuyor olsun:

```
.L1:
    nop
    nop
    mov     r1, r0
    cmp     r0, r1
    bgt     .L1
    mov     r0, r1
    bx      lr
```

Burada bgt komutu çalışırken PC yazmacı iki ileriyi (yani bx lr komutunu) gösteriyor durumdadır. O halde bu nokta sıfır olmak üzere dallanılan yer bundan 6 geridir. Komutların makine kodu karşılığını inceleyiniz:

```
 0:   e3a00001        mov     r0, #1
 4:   e3a01002        mov     r1, #2
 8:   e1a00000        nop                     ; (mov r0, r0)
 c:   e1a00000        nop                     ; (mov r0, r0)
10:   e1a01000        mov     r1, r0
14:   e1500001        cmp     r0, r1
18:   cafffffa        bgt     8 <main+0x8>
1c:   e1a00001        mov     r0, r1
20:   e12fff1e        bx      lr
```

bgt komutunun 24 bitindeki değer FFFFFA biçimindedir. Bu da işaretli sistemde -6 değerini belirtir. Şimdi bunun tersini yapalım:

```
    mov     r1, r0
    cmp     r0, r1
    bgt     .L1
    mov     r0, r1
    mov     r1, r0
    nop
    nop
.L1:
    bx      lr
```

Burada bgt komutu çalışırken PC yazmacı onun iki ilerisini (yani mov r1, r0 komutunu göstermektedir. Burası o kabul edildiğinde dallanılacak yer buradan 3 ileridedir. Böylece komuttaki yer değiştirme miktarı 3 olmalıdır. Aşağıda kodun makine kodlarını görüyorsunuz:

```
 0:   e1a01000        mov     r1, r0
 4:   e1500001        cmp     r0, r1
 8:   ca000003        bgt     1c <main+0x1c>
 c:   e1a00001        mov     r0, r1
10:   e1a01000        mov     r1, r0
14:   e1a00000        nop                     ; (mov r0, r0)
18:   e1a00000        nop                     ; (mov r0, r0)
1c:   e12fff1e        bx      lr
```

bgt komutunun makine kodu karşılığının düşük anlamlı 24 bitinin 000003 olduğuna dikkat ediniz.

ARM mimarisinde de tıpkı Intel mimarisinde olduğu gibi önce bayrakları konumlandırıp sonra dallanma yapmak gerekir. Anımsanacağı bunun için SUB komutu ya da bunun bayrakları etkilemeyen CMP versiyonu kullanılıyordu. ARM mimarisinde de aynı durum söz konusudur. Böylece ARM mimarisinde de önce bir CMP komutu uygulayıp sonra B komutu ile dallanma yapılabilir. Örneğin:

```
CMP     r0, r1
bgt     .L1     ; r0 işaretli olarak r1'den büyükse dallan
...
.L1:
...
```

Pekiyi ARM'da dolaylı dallanma komutu var mıdır? Yanıt evet. ARM'daki dolaylı dallanma yazmaç içerisindeki dallanma işlrmini yapan BX komutuyla gerçekleştirilmektedir. BX komutu operand olarak aldığı yazmacın içerisindeki adrese dallanır. Bu adres doğrudan dallanılmak istenen adres olmalıdır. (Yani onun bir ilerisi ya da iki ilerisi olmamalıdır.) Böylece tipik olarak bellekte bir adrese dolaylı dallanma işlemi şöyle yapılabilir:

```
LDR     R0, label
BX      R0
```

BX komutu R14 yazmacıyla (ki buna LR (Link Register) de denilmektedir. Kullanıldığında adeta RET makine komutu etkisi yaratılmaktadır. Bu konu sonraki başlıkta ele alınmaktadır.

ARM Mimarisinde Fonksiyonların Çağrılması (CALL İşlemi)

ARM mimarisinde fonksiyon çağırma işlemleri için CALL makine komutu değil BL ya da BLX makine komutları kullanılmaktadır. Fonkisyona dallanma sırasında sonraki komutun adresi Intel mimarisinde olduğu gibi stack'te saklanmaz. R14 yazmacında saklanır. R14 yazmacına "Link Yazmacı (Link Register)" da denilmektedir. Link yazmacı bu işlemin dışında normal diğer yazmaçlar gibi her işleeme sokulabilir. BL (Branch with Link) komutu operand olarak tıpkı B komutunda olduğu gibi göreli yer değiştirme miktarını (displacement) alır. Sembolik makine dilinde bu da tipik olarak bir etiket olur. Yer değiştirme miktarı yine işaretli 24 bit ile ifade edilmektedir. Sınırlar B komutuyla aynıdır. Bu komutta da yer değiştirme miktarı byte cinsinden değildir. Byte'ın 4 katı cinsindendir (yani komut sayısı miktarı cinsinden). BL komutu dallanmayı yapmadan önce sonraki komutun adresini LR yazmacına yerleştirmektedir. Örneğin:

```
foo :
    ...
    bx  lr
...
bl  foo
mov r0, r1
```

Burada foo isimli fonksiyon BL komutuyla çağrılmıştır. Bu çağırma sırasında sonraki komut olan mov r0, r1 komutunun adresi LR yazmacına (R14 yazmacı) yerleştirilir. Sonra dallanma yapılır. BX LR komutu adeta Intel'dkei RET makine komutu gibi etki göstermektedir. Bu komut "LR yazmacındaki adrese koşulsuz dallan" anlamına gelir ki bu da geri dönüş işlemini yapacaktır. O halde ARM mimarisinde fonksiyonu çağırmak için BL, oradan geri dönmek için BX LR komutu kullanılmaktadır. Tabii burada ilk akla gelecek soru "iç içe çağrımlarda LR'nin ne olacağıdır". Yani örneğin yukarıda foo fonksiyonu da bar fonksiyonunu çağırsaydı LR yazmacının değeri bozulmayacak mıydı?

```
bar:
    ...
    bx  lr
foo :
    bl  bar
    bx  lr
...
bl  foo
mov r0, r1
```

Evet burada LR yazmacındaki değer bozulacağı için kod düzgün çalışmayacaktır. İşte iç içe çağırmalarda artık programcının kendisinin LR yazmacını stack'e atıp BX LR yapmadan önce stack'ten çekmesi gerekir. Stack kullanımı sonrakş başlıkta ele alınmaktadır.

ARM mimarisinde dolaylı CALL (indirect CALL) işlemi için BLX komutu kullanılmaktadır. Bu komut bir yazmacı operand olarak alır. LR yazmacına sonraki komutun adresini aktardıktan sonra operandı olan yazmacın gösterdiği adrese dallanma yapar. Oradan geri dönüş yine BX LR ile yapılmalıdır.

Yazmaçların Bellekte Saklanıp Geri Alınması ve Stack Kullanımı

ARM mimarisinde stack kullanımı Intel'den biraz farklıdır. Anımsanacağı gibi Intel'de stack ESP (ya da SP ya da RSP) yazmacının gösteridği yerdedir. PUSH ve POP isimli iki makine komutu stack'e değer yerleştirip geri almak için kullanılır. PUSH ve POP işlemleri sırasında ESP (ya da SP ya da RSP) otomatik olarak artırılıp azaltılmaktadır. Halbuki ARM mimarisinde stack aslında LDM ve STM denilen yazmaçları bellekte saklayıp alma işleminni yapan komutlarla idare edilmektedir. Bu biçimde stack organizasyonu daha genel bir kullanım sunmaktadır.

LDM ve STM komutlarının genel biçimi şöyledir:

```
LDM{addr_mode} {cond} Rn{!}, {reglist}
STM{addr_mode} {cond} Rn{!}, {reglist}
```

Buradan da görüldüğü gibi LDM ve STM komut isimlerini bunlarla bitişik yazılan "adres modları" izlemektedir. Dört adres modu vardır:

```
IA (Increment After)
IB (Increment Before)
DA (Decrement After)
DB (Decrement Before)
```

Böylece komutlar şu varyasyonlara sahip olurlar: LDMIA, LDMIB, LDMDA, LDMDB, STMIA, STMIB, STMDA ve STMDB. Bu komut varyasyonları tipik olarak stack gerçekleştirimi için de kullanıldığından bunlara alternatif isimler de verilmiştir. Şöyle ki:

```
STMDB = STMFD
STMIB = STMFA
STMDA = STMED
STMIA = STM = STMEA

LDMIA = LDM = LDMFD
LDMDA = LDMFA
LDMIB = LDMEB
LDMDB = LDMEA
```

Komuttaki yazmaç listesi herhangi bir ya da birden fazla yazmacı içerebilir. Ancak ARM mimarisi R13 (SP) ve R15 (PC) yazmaçlarının listede belirtilmesini “deprecated” ilan etmiştir. Yani bu yazmaçların listede bulunması yasak değildir ancak tavsiye edilmemektedir. İleriki versiyonlarda tümden yasaklanma olasılığı da vardır. Yazmaç listesini küme parantezleri içerisinde verirken RX-RY biçiminde aralık belirtilebilir. Örneğin:

{R0-R7, R10}

Burada listede R0, R1, R2, R3, R4, R5, R6, R7 ve R10 yazmaçları bulunmaktadır. GNU’nun sembolik makine dili derleyicisinde küme parantezi içerisindeki yazmaçların numara sırasına göree dizilmiş olması gerekmektedir. Yani örneğin {FP, R0-R4, LR} dizilimi geçerli değildir. Ancak {R0-R4, FP, LR} dizilimi geçerlidir.

STM ve LDM komutlarının birinci operandları bir yazmaç olmak zorundadır, saklamanın ya da yüklemenin yapılacağı bellek adresini belirtir. Örneğin:

```
STM R13, {R0-R5}
```

Bu komut “R0, R1, R2, R3, R4, R5 yazmaçlarını R13 yazmacının içerisindeki adresten itibaren belleğe aktar” anlamına gelir. R13 yazmacının genellikle “Stack Göstericisi” olarak kullanıldığını anımsayınız. O halde kabaca (ayrıntılar biraz ileride ele alınacaktır) bu komut bu yazmaçların stack’e atılacağı anlamına gelir. Birinci operandın sonuna ! karakteri getirilebilir. Bu durum “load” ya “store” işleminden sonra bu yazmaçta adres moduna göre güncelleme yapılacağı anlamına gelir. Örneğin:

```
STM R13!, {R0-R5}
```

Burada R13 yazmacı biraz ileride ele alınacağı gibi değer değiştirecektir. Şüphesiz bu komutların stack amaçlı kullanımlarında ! bulunmalıdır. (Aslında komutun !’li olup olmadığı makine kodundaki bir bir ile belirlenmektedir.)

Şimdi STM ve LDM komutlarının adres modlarını açıklayalım. Yukarıda da belirtildiği gibi bu komutlar dört adres modu almaktadır:

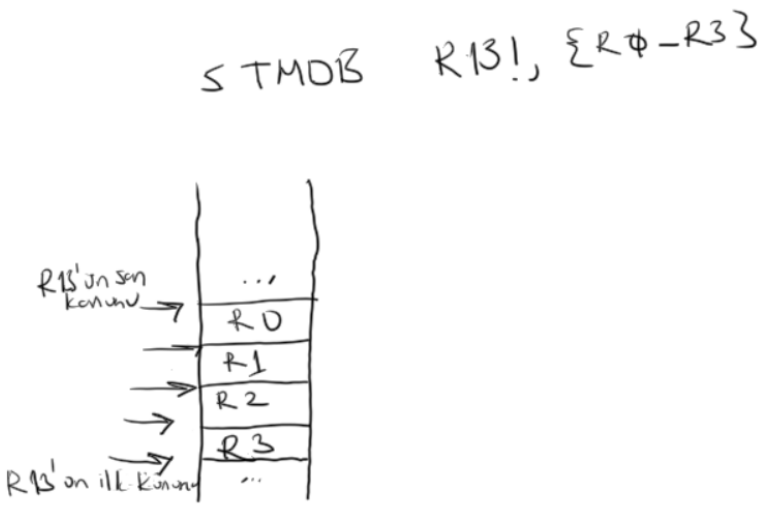
```
IA (Increment After)
IB (Increment Before)
DA (Decrement After)
DB (Decrement Before)
```

Ayrıca bu adres modlarının bazı eşdeğer isimleri de yukarıda belirtilmiştir. Burada bu komutları tek tek ele alalım.

STMDB/STMFD Komutu

Adres modundaki DB (decrement before) yazmaç aktarımından önce adresin eksiltileceğini belirtir. Buna göre STMDB komutu şöyle çalışır: Komut listede belirtilen yazmaçları ters sırada (yüksek numaradan düşük numaaraya doğru) birinci operand ile belirtilen adresten itibaren adresi azalta azalta belleğe aktarır. Ancak adresin azaltılma işlemi belleğe aktarım işleminden önce yapılmaktadır. Örneğin:

```
STMDB   R13!, {R0-R3}
```



Komutta birinci operand'ta '!' olduğu için R13'ün değeri değişecektir. Eğer komutta '!' karakteri bulunmasaydı işlemler aynı biçimde yapılacaktı ancak R13 konum değiştirmeyecekti. R13 yazmacının genellikle stack göstericisi olarak kullanıldığını anımsayınız. O zaman bu komutu şöyle de yazabilirdik:

```
STMDB   SP!, {R0-R3}
```

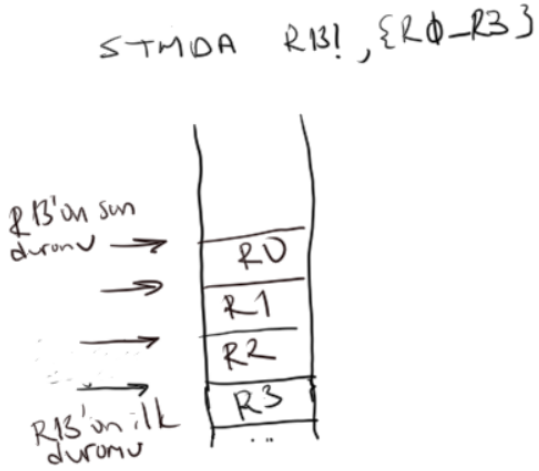
Bu işlemin Intel mimarisindeki PUSH işlemine benzediğine dikkat ediniz. Gerçekten de ARM mimarisinde stack'e push işlemi tipik olarak STMDB/STMFD komutuyla yapılmaktadır. Örneğin biz FP(R11) yazmacını stack'e atmak isteyelim:

```
STMFD   SP!, {fp}
```

STMDA/STMED Komutu

Adres modundaki DA (Decrement After) aktarım adresinin aktarımdan sonra eksiltileceği anlamına gelir. Yani yazmaçlar ters sırada (yüksekten düşüğe doğru) yine birinci operand ile belirtilen adresten itibaren belleğe aktarılır. Ancak adresin eksiltilmesi aktarımdan sonra yapılır. Örneğin:

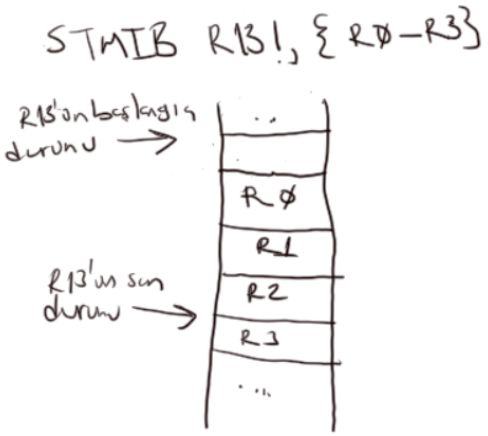
```
STMDA   R13!, {R0-R3}
```



STMIB/STMFA Komutu

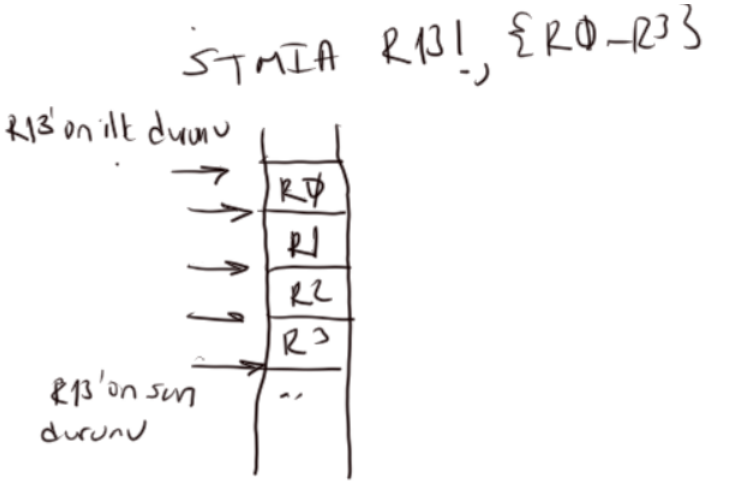
Komuttaki IB (Increment Before) önce adres artırılıp sonra adrese yerleştirme yapılacağı anlamına gelir. Yani yazmaçlar ters sırada adres artırılarak belleğe yerleştirilmektedir. Örneğin:

```
STMIB   R13!, {R0-R3}
```



STMIA/STM/STMEA Komutu

Adres modundaki IA (Increment After) önce yerleşrime yapılıp sonra adresin artırılacağı anlamına gelir. STM komutunun default durumu (yani adres modunu yazmadığımız durum) bu biçimdedir.

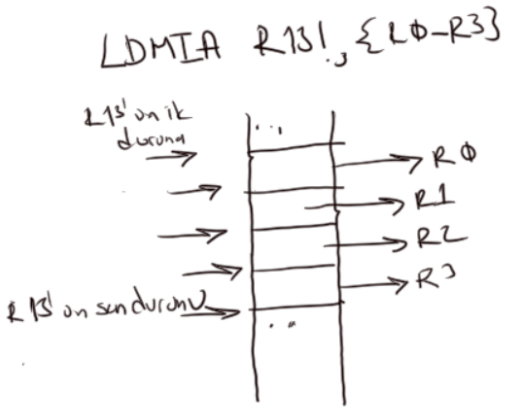


,

LDMIA/LDM/LDMFD Komutu

Bu komutta önce birinci operand ile belirtilen adresten yazmaç değeri çekilerek yazmaca yerleştirilir sonra artırım uygulanır. Bu aslında stack bağlamında Intel'deki POP komutuna benzemektedir. O halde biz stack'e STMDB/STMFD ile push işlemi yapmışsak LDMIA/LDM/LDMFD ile POP işlemi yapmalıyız. Örneğin:

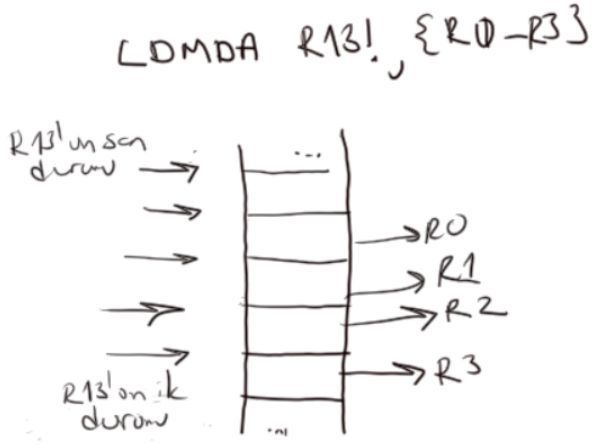
```
LDMIA R13!, {R0-R3}
```



LDMDA/LDMFA Komutu

Burada önce birinci operand ile belirtilen adresteki değer sıradaki yazmaca çekilir, sonra adres azaltılır. Eğer stack'e STMIB/STMFA komutuyla PUSH işlemi yapılmışsa LDMDA/LDMFA ile POP işlemi yapılmalıdır. Örneğin:

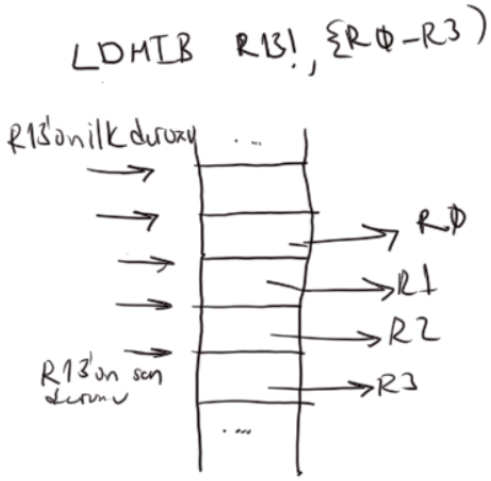
```
LDMDA R13!, {R0-R3}
```



LDMIB/LDMEB Komutu

Burada önce artırım yapılıp sonra yerleştirme yapılmaktadır. eğer stack'e PUSH işlemi STMDA/STMED komutuyla yapılmışsa POP işlemi de LDMIB/LDMEB komutuyla yapılmalıdır. Örneğin:

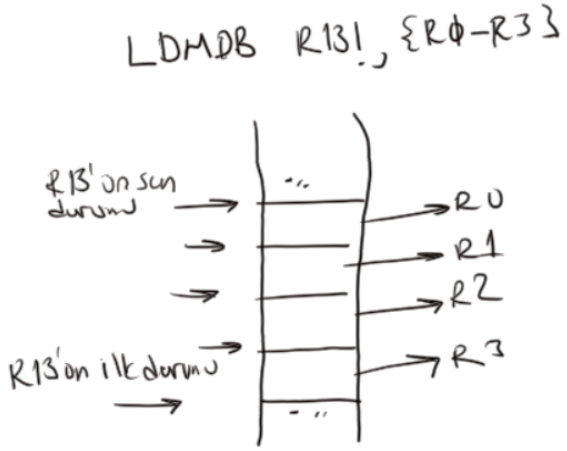
```
LDMIB R13!, {R0-R3}
```



LDMDB/LDMEA Komutu

Burada önce adres azaltılır sonra yazmaca yerleştirme yapılır. Eğer stack'e PUSH işlemi STMIA/STM/ STMEA komutuyla yapılmışsa POP işlemi de LDMDB/LDMEA komutuyla yapılmalıdır. Örneğin:

```
LDMDB R13!, {R0-R3}
```



Yukarıda da görüldüğü gibi STM ve LDM komutlarının dört farklı biçimi vardır. Bu komutlar aslında genel load/store komutlarıdır. Ancak ağırlıklı olarak stack gerçekleştiriminde kullanılmaktadır. ARM mimarisi için çağırma biçiminde stack tamamen Intel mimarisindeki gibi organize edilmektedir. O halde biz stack gerçekleştiriminde PUSH işlemini STMDB/STMFD ile POP işlemini de LDMIA/LDMFD/LDM komutuyla yaparız. Stack amaçlı olarak bu komutların kullanılması durumunda okunabilirlik için genellikle STMFD ve LDMFD isimleri tercih edilmektedir. Örneğin stack'e üç yazmacı PUSH edip onları sonra POP edelim:

```
STMFD SP!, {R0, R1, R2}
...
LDMFD SP!, {R0, R1, R2}
```

ARM sembolik mekine dili derleyicilerinde diğer mimarilerde çalışanlara kolaylık sağlamak için STMFD ve LDMFD komutlarının stack amaçlı kullanımı için PUSH ve POP isimleri de kullanılabilmektedir. Şöyle ki:

```
STMFD SP!, {yazmaç listesi}
```

komutunun eşdeğeri:

```
PUSH {yazmaç listesi}
```

biçiminde de yazılabilmektedir. Benzer biçimde:

```
LDMFD SP!, {yazmaç listesi}
```

komutunun da eşdeğeri:

```
POP {yazmaç listesi}
```

biçiminde yazılabilmektedir.

32 Bit ARM Mimarisinde C ve C++ İçin Fonksiyon Çağırma Biçimi (32 Bit ARM Calling Convention)

32 Bit ARM mimarisinde farklı fonksiyon çağırma biçimleri yoktur. Tek bir standart biçim kullanılmaktadır. Bu biçimin kuralları şöyledir:

- Stack "Full Descending" olarak kullanılır. Yani bu anlamda stack2in kullanımı tamamen Intel'deki gibidir. Bu nedenle fonksiyon içerisinde STMFD ve LDMFD gibi load/store komutları sık kullanılmaktadır.

- R0, R1, R2 ve R3 yazmaçları fonksiyon çağrılarında soldan sağa ilk dört parametreyi aktarmak için kullanılır. Eğer fonksiyon dörtten fazla parametreye sahipse geri kalan paramtreler stack yoluyla aktarılır. Geri kalan parametreler stack'e sağdan sola push edilir. (Yani düşük adreste beşinci parametre olacak biçimde)

- R0, R1, R2, R3 yazmaçları çağrılan fonksiyon tarafından (callee) bozulabilir. Çağrılan fonksiyon bunları saklamak zorunda değildir. Ancak R4-R11 ve R14 yazmaçları çağrılan fonksiyon tarafından korunmak zorundadır. Yani çağrılan fonksiyon bu yazmaçları kullanacaksa önce onları stack'e push edip fonksiyondan çıkarken sonra pop etmelidir.

- Fonksiyonların geri dönüş değerleri 4 byte ise R0 ile daha uzunsa R0, R1, R2, R3 yazmaçları ile iletilir. R0 burada düşük anlamlı "word"ü (ARM'a göre 4 byte'ı) belirtir. Eğer fonksiyonun geri dönüş değeri daha büyükse bu durumda fonksiyon kendi içerisinde bir (stack alanını) alanı tahsis eder, onun adresini R0 yazmacı ile verir.

- Çağırma işlemi BL komutuyla yapıldığı için fonksiyondan geri dönüş de BX LR ile yapılır. Tabii bu durumda fonksiyon kendi içerisinde de başka fonksiyonu çağıracaksa LR (R14) yazmacının da stack'e push edilmesi sonra da pop edilmesi gerekir.

- Çok kullanılan bir yöntem parametrelerin öncelikle stack'te ya da R0-R11 yazmaçlarında saklanmasıdır. Benzer biçimde bazı derleyiciler yerel değişkenleri de mümkün olduğunca R4-R11 yazmaçlarında saklamaktadır.

- Fonksiyon genel olarak stack'teki parametre değişkenlerini ve yerel değişkenleri R11 (Frame Pointer) yazmacını kullanarak çeker. Hatta GASM'de R11 yazmacı için FP ismi de kullanılmaktadır.

Şimdi bazı denemeler yapalım:

```
/* sample.c */

#include <stdio.h>

int add(int a, int b);

int main(void)
{
    int result;

    result = add(10, 20);
    printf("%d\n", result);

    return 0;
```

```
}
```

Burada iki parametreli bir add fonksiyonu çağrılmıştır. Derleyici a parametresini R0 ile b parametresini de R1 ile aktaracaktır. Sonra geri dönüş değerini R0'dan alacaktır. Buradaki add fnksiyonunu sembolik makine dilinde yazarken aslında herhenagi bir push ya pop işlemine gereksinim duymayız. Çünkü zaten R0-R3 çağrılan fonksiyon tarafından bozulabilen yazmaçlardır. O halde add fonksiyonu şöyle yazılabilir:

```
/* util.s */

.arm
.global add

add:
    add r0, r0, r1

    bx  lr
```

Derleme işlemi şöyle yapılabilir:

```
as -o util.o util.s
gcc -o sample sample.c util.o
```

Aslında bu iki işlem aşağıdaki gibi tek bir işlem olarak da yapılabilir:

```
gcc -o sample sample.c util.s
```

Sonraki örneklerde hep C kaynak dosyası "sample.c", sembolik makine dili dosyası ise "util.s" olarak alınacaktır. Dolayısıyla denemeleri yukarıdaki gibi yapabilrsizniz.

Şimdi parametre sayısını fazlalaştıralım:

```
/* sample.c */

#include <stdio.h>

int add(int a, int b, int c, int d);

int main(void)
{
    int result;

    result = add(10, 20, 30, 40);
    printf("%d\n", result);

    return 0;
}
```

Fonksiyon sembolik makine dilinde şöyle yazılabilir:

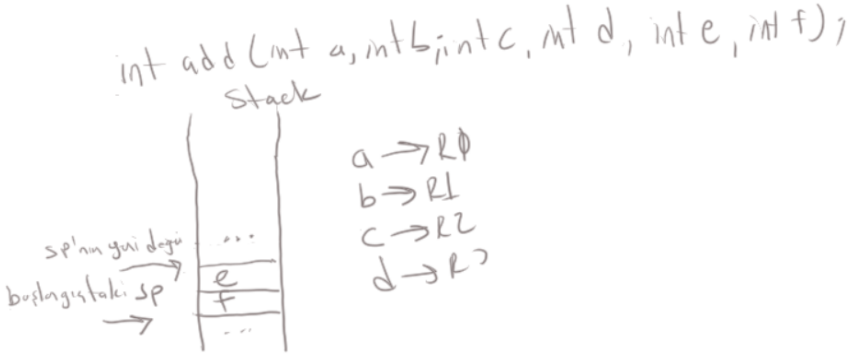
```
/* util.s */

.arm
.global add

add:
    add     r3, r2, r3
    add     r0, r0, r1
    add     r0, r0, r3
```

```
    bx  lr
```

Şimdi parametreleri biraz daha fazlalaştıralım:

```
/* sample.c */

#include <stdio.h>

int add(int a, int b, int c, int d, int e, int f);

int main(void)
{
    int result;

    result = add(10, 20, 30, 40, 50, 60);
    printf("%d\n", result);

    return 0;
}
```

add fonksiyonunu yazarken son iki parametrenin stack yoluyla aktarılacağına dikkat ediniz. Akış fonksiyona geldiğinde stack'in durumu şöyle olacaktır:

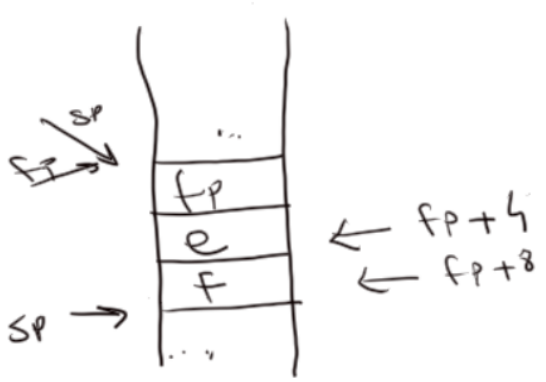


O halde biz fonksiyonda [sp, #0] bellek operandı ile e parametresine, [sp, #4] bellek operandıyla da f parametresine erişiriz.

```
/* util.s */

.arm
.global add

add:
    add     r3, r2, r3
    add     r0, r0, r1
    add     r0, r0, r3
    ldr     r1, [sp, #0]
    ldr     r2, [sp, #4]
    add     r1, r1, r2
    add     r0, r0, r1

    bx  lr
```

Buraada da yine bir push işlemine gerek olmadığına dikkat ediniz. Şimdi add fonksiyonunu stack'teki belgiyi R11 (FP) yazmacı ile çekecek biçimde yeniden yazalım. Buradaki R11 (FP) yazmacı Intel'deki EBP yazmacına karşılık gelmektedir. (Tabii ARM mimarisinde bu amaçla R11 yazmacının kullanılması zorunlu değildir. Genel bir eğilimdir.)

```
/* util.s */

.arm
.global add

add:
    stmfd   sp!, {fp}
    mov     fp, sp
    add     r3, r2, r3
    add     r0, r0, r1
    add     r0, r0, r3
    ldr     r1, [fp, #4]
    ldr     r2, [fp, #8]
    add     r1, r1, r2
    add     r0, r0, r1

    ldmfd   sp!, {fp}
    bx  lr
```



fp'nin stack'e atılıp geri alınması str ve ldm komutlarıyla şöyle yapılabilirdi:

```
/* util.s */

.arm
.global add

add:
    str     fp, [sp, #-4]!

    mov     fp, sp
    add     r3, r2, r3
    add     r0, r0, r1
    add     r0, r0, r3
    ldr     r1, [fp, #4]
    ldr     r2, [fp, #8]
    add     r1, r1, r2
    add     r0, r0, r1

    ldr     fp, [sp], #4
    bx  lr
```

Burada:

```
str     fp, [sp, #-4]!
```

komutuyla önce fp sp -4 adresine aktarılmış sonra da sp'den 4 çıkartılmıştır. Bu tamamen Intel sentaksıyla "push fp" gibi etki yaratmaktadır. Benzer biçimde:

```
ldr     fp, [sp], #4
```

Komutunda da önce sp'nin gösterdiği yerdeki değer fp'ye atanmış daha sonra sp 4 artırılmıştır. Bu işlem de Intel'deki "pop fp"ye benzemektedir.

Şimdi iki int sayının büyüğünü bulan bir fonksiyonu yazmaya çalışalım:

```
/* sample.c */

#include <stdio.h>

int getmax(int a, int b);

int main(void)
{
    int result;

    result = getmax(20, 10);
    printf("%d\n", result);

    return 0;
}
```

getmax sembolik makine dilinde şöyle yazılabilir:

```
/* util.s */

.arm
.global getmax

getmax:
    cmp     r0, r1
    bgt     .L1
    mov     r0, r1
.L1:
    bx  lr
```

Şimdi bir döngü kurmaya çalışalım. Örneğin bir int dizinin tüm elemanlarının toplamına geri dönen bir fonksiyon yazmaya çalışalım:

```
/* sample.c */

#include <stdio.h>

int get_total(const int *pi, int size);

int main(void)
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int result;


    result = get_total(a, 10);
    printf("%d\n", result);

    return 0;
}
```

get_total Fonksiyonu şöyle yazılabilir:

```
/* util.s */

.arm
.global get_total

/*
    ro -> dizinin başlangış adresi
    r1 -> dizinin uzunluğu
    r2 -> sayaç
    r3 -> kümülatif toplam
    r4 -> dizinin o anki elemanı
*/

get_total:
    stmfd   sp!, {r4}

    mov     r2, #0
    mov     r3, #0
    b       .L1
.L2:
    ldr     r4, [r0, #0]
    add     r3, r3, r4

    add     r2, r2, #1
    add     r0, r0, #4
.L1:
    cmp     r2, r1
    blt     .L2
    mov     r0, r3

    ldmfd   sp!, {r4}

    bx  lr
```

Burada R4 yazmacının stack'e push edildiğine dikkat ediniz. Çünkü çağırma biçimi kuralına göre R4-R11 ve R14 yazmaçları çağrılan fonksiyon (callee) tarafından korunmalıdır. Döngü sayacı için R2 yazmacı kullanılmıştır. Toplam değer de R3 yazmacında tutulmuştur. Dizi elemanlarını geçici süre tutmak için R4 yazmacından faydalanılmıştır.

Şimdi de int bir dizinin en büyük elemanıyla geri dönen get_max fonksiyonunu yazmaya çalışalım:

```
/* sample.c */

#include <stdio.h>

int get_max(const int *pi, int size);

int main(void)
{
    int a[10] = {40, 23, 24, 112, 0, -4, 23, 67, 32, 10};
    int result;


    result = get_max(a, 10);
    printf("%d\n", result);

    return 0;
}
```

Fonksiyon sembolik makine dilinde şöyle yazılabilir:

```
/* util.s */
```

```
.arm
.global get_max


/*
    ro -> dizinin başlangış adresi
    r1 -> dizinin uzunluğu
    r2 -> sayaç
    r3 -> en büyük elemanın saklandığı yer
    r4 -> dizinin o anki elemanı
*/

get_max:
    stmfd   sp!, {r4}


    ldr     r3, [r0], #4
    mov     r2, #1
    b       .L1
.L2:
    ldr     r4, [r0], #4
    cmp     r3, r4
    bge     .L3
    mov     r3, r4
.L3:
    add     r2, r2, #1
.L1:
    cmp     r2, r1
    blt     .L2

    mov     r0, r3

    ldmfd   sp!, {r4}

    bx  lr
```

Burada dizinin en büyük elemanının R3'te tutulduğuna dikkat ediniz. İşin başında dizinin ilk elemanı en

Şimdi bir adresten başka bir adrese n byte kopyalayan memcpy fonksiyonunu my_memcpy ismiyle yazmaya çalışalım:

```c
/* sample.c */

#include <stdio.h>

void *my_memcpy(void *dest, const void *source, size_t size);

int main(void)
{
    int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int b[10];
    int *pi;
    int i;

    pi = (int *)my_memcpy(b, a, sizeof(int) * 10);
    for (i = 0; i < 10; ++i)
        printf("%d ", pi[i]);

    printf("\n");

    return 0;
}
```

Fonksiyon şöyle yazılabilir:

```
/* util.s */

.arm
.global my_memcpy

/*
    ro -> dest dizisin başlangış adresi
    r1 -> source dizisinin başlangıç adresi
    r2 -> size
    r3 -> sayaç
    r4 -> dizinin o anki elemanı
*/

my_memcpy:
    stmfd   sp!, {r0, r4}

    mov     r3, #0
    b       .L1
.L2:
    ldrb    r4, [r1, #0]
    strb    r4, [r0, #0]

    add     r0, r0, #1
    add     r1, r1, #1
    add     r3, r3, #1
.L1:
    cmp     r3, r2
    blo     .L2

    ldmfd   sp!, {r0, r4}

    bx  lr
```

Buradaki kod ARM mimarisi dikkate alındığında biraz kısaltılabilir. Şöyle ki: Anımsanacağı gibi LDR komutlarında postfix artırım uygulanabiliyordu. Yani:

```
ldrb    r4, [r0, #0]
add     r0, r0, #1
```

İşlemi yerine tek hamlede:

```
ldrb    r4, [r0], #1
```

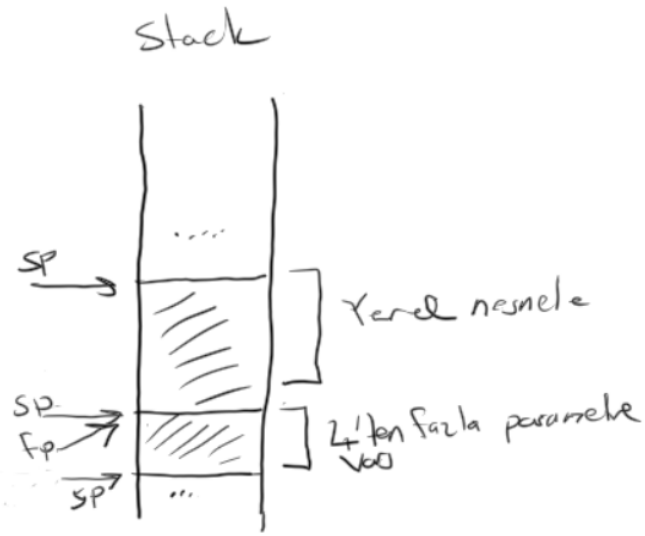
biçiminde yazılabilir:

```
/* util.s */

.arm
.global my_memcpy

/*
    ro -> dest dizisin başlangış adresi
    r1 -> source dizisinin başlangıç adresi
    r2 -> size
    r3 -> sayaç
    r4 -> dizinin o anki elemanı
*/

my_memcpy:
    stmfd   sp!, {r0, r4}

    mov     r3, #0
```

```
    b       .L1
.L2:
    ldrb r4, [r1], #1
    strb r4, [r0], #1

    add     r3, r3, #1
.L1:
    cmp     r3, r2
    blo     .L2

    ldmfd   sp!, {r0, r4}

    bx  lr
```

Yerel Değişkenlerin Kullanılması

ARM mimarisinde de yerel değişkenler yine Intel mimarisinde olduğu gibi kullanılırlar. Yani yerel değişkenlere erişmek için FP (R11) sabitlenir. Sonra yerel değişkenlere yer açmak için stack göstericisi yukarı çekilir. Hangi yerel değişkenin stackte yaratılacağı konusunda bir belirleme yoktur. Bazı derleyiciler ilk bildirilen nesneyi düşük adreste tutarken çalışırken bazıları yüksek adreste tutabilmektedir.



Tabii fp yazmacının değerini koruması için başlangıçta stack'e push edilmesi gerekmektedir. Tıpkı Intel'de olduğu gibi burada da yerel değişkenlere artık [FP, # -N] bellek operandıyla erişilir.

Örneğin aşağıdaki mul fonksiyonunu sembolik makine dilinde yazmaya çalışalım:

```
int mul(int a, int b)
{
    int total;

    total = a * b;

    return total;
}

/* util.s */

.arm
.global mul

/*
    ro -> dizinin başlangış adresi
    r1 -> dizinin uzunluğu
    r2 -> sayaç
    r3 -> en büyük elemanın saklandığı yer
    r4 -> dizinin o anki elemanı
```

```
*/

mul:
    stmfd   sp!, {fp}
    mov     fp, sp

    mul     r2, r0, r1
    str     r2, [fp, #-4]    /* total = a * b; */

    ldr     r0, [fp, #-4]/* return total; */


    ldmfd   sp!, {fp}

    bx  lr
```

Şimdi de bazı fonksiyonlar için gcc'nin nasıl kod ürettiğine bakalım. Örneğin:

```
get_total:
    @ args = 0, pretend = 0, frame = 16
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    str fp, [sp, #-4]!
    add fp, sp, #0
    sub sp, sp, #20
    str r0, [fp, #-16]
    str r1, [fp, #-20]
    mov r3, #0
    str r3, [fp, #-8]
    mov r3, #0
    str r3, [fp, #-12]
    b   .L2
.L3:
    ldr r3, [fp, #-12]
    mov r3, r3, asl #2
    ldr r2, [fp, #-16]
    add r3, r2, r3
    ldr r3, [r3, #0]
    ldr r2, [fp, #-8]
    add r3, r2, r3
    str r3, [fp, #-8]
    ldr r3, [fp, #-12]
    add r3, r3, #1
    str r3, [fp, #-12]
.L2:
    ldr r2, [fp, #-12]
    ldr r3, [fp, #-20]
    cmp r2, r3
    blt .L3
    ldr r3, [fp, #-8]
    mov r0, r3
    add sp, fp, #0
    ldmfd   sp!, {fp}
    bx  lr
```

Burada yerel değişkenler için biraz fazlaca yer ayrıldığını görüyorsunuz. Burada önce parametrelerin stack'te oluşturulan bazı yerlerde geçici olarak saklandığını görüyorsunuz. Birinci parametre olan dizi adresi R0 yazmacından yazmacından alınarak FP - 16 adresine yerleştirilmiştir. İkinci parametre olan size ise FP - 20 adresin yerleştirilmiştir. Dizi indeksi olarak kullanılan i değişkeni FP - 8'dedir. total yerel değişkeni için FP - 12 adresi kullanılmıştır. Tabii buradaki kodun oldukça verimsiz bir biçimde üretilmiş olduğu gözükmektedir. Derleyicinin optimizasyon seçeneğini -O2 ve -O3 yaptığımızda çok daha sade kod üretildiğini görebilmekteyiz.

Çağrılan fonksiyonun başka bir fonksiyonu çağırdığı durumda LR (R14) yazmacının da stack'e push

edilmesi gerektiğini unutmayınız. Örneğin:

```
/* sample.c */

#include <stdio.h>

int mul(int a, int b)
{
    return a * b;
}

int add(int a, int b)
{
    return mul(a, b);
}

int main(void)
{
    int result;

    result = add(10, 20);
    printf("%d\n", result);

    return 0;
}
```

add fonksiyonu içerisinde LR yazmacının saklanması gerekir. gcc optimizasyon seçenekleri kapalıyken bu kod için aşağıdaki sembolik makine kodarını üretmektedir:

```
mul:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    @ link register save eliminated.
    str fp, [sp, #-4]!
    add fp, sp, #0
    sub sp, sp, #12
    str r0, [fp, #-8]
    str r1, [fp, #-12]
    ldr r3, [fp, #-8]
    ldr r2, [fp, #-12]
    mul r3, r2, r3
    mov r0, r3
    add sp, fp, #0
    ldmfd   sp!, {fp}
    bx  lr
    .size   mul, .-mul
    .align  2
    .global add
    .type   add, %function
add:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
    add fp, sp, #4
    sub sp, sp, #8
    str r0, [fp, #-8]
    str r1, [fp, #-12]
    ldr r0, [fp, #-8]
    ldr r1, [fp, #-12]
    bl  mul
    mov r3, r0
    mov r0, r3
    sub sp, fp, #4
    ldmfd   sp!, {fp, pc}
```

Üretilen kodda yazmaçların gereksiz bir biçimde stack'te saklandığı görüyorsunuz. Ancak mul fonksiyonunda LR yazmacının stack'e push edilmediğine ancak add fonksiyonunda edildiğine dikkat ediniz. push ve pop işlemlerinde ilginç bir optimizasyonun yapıldığını da görüyorsunuz:

```
stmfd   sp!, {fp, lr}
...
ldmfd   sp!, {fp, pc}
```

ARM mimarisinde PC yazmacı normal bir yazmaç gibi MOV, LDR, STR, LDM ve STM işlemlerine sokulabilmektedir. Burada stack'ten çekilen LR değeri doğrudan PC yazmacına aktarılmıştır.

ARM Tabanlı Linux Sistemlerinde Merhaba Dünya Programı

Linux ortamında ARM'da ekrana Merhaba Dünya yazısı çıkartan bir program şöyle yazılabilir:

```
    /* sample.s */

    .arm

    .section .data
    .LC1:
    .ascii  "Merhaba Dunya\n"

.text
        .global _start

_start:
    mov r0, #1
    ldr r1, =.LC1
    mov r2, #14
    mov r7, #4
    swi #0

    mov r0, #0
    mov r7, #1
    swi #0
```

Linux'un X86 ve ARM versiyonlarında sistem fonksiyonlarının numaraları aynıdır. Yukarıdaki programda C'ce şunlar yapılmıştır:

```
write(1, "Merhaba Dunya\n", 14);
exit(0);
```

Sistem fonksiyonları yine normal fonksiyonlarla aynı çağırma biçimine sahiptir. Sistem fonksiyonları çağrılırken sistem fonksiyonun numarası R7 yazmacına yüklenir. Sonra parametreler sırasıyla R0, R1, R2, R3, R4 yazmaçları ile aktarılır. write sistem fonksiyonun numarasının 4, exit sistem fonksiyonun ise 1 olduğunu anımsayınız. Derleme ve link işlemi şöyle yapılabilir:

```
as -o sample.o sample.s
ld -o sample sample.o
```

Thumb Komut Kümesi

ARM işlemcileri belirli modelerinden sonra Thumb denilen komut kümesini desteklemeye başlamıştır. Thumb her komutun 4 byte yerine 2 byte'tan oluştuğu daha kompakt kod yerleşiminin mümkün olduğu bir komut kümesi sunmaktadır. Komutların 4 byte yerine 2 byte içerisinde kodlanması bazı durumlarda bazı komutlar için daha hızlı bir çalışma da sunar. Thumb komut kümesinin işlemcinin word uzunluğu ile bir ilgisi yoktur. Bu komut kümesinin en önemli tasarım nedeni kodun kapladağı alanı azaltmaktır. Şüphesiz her komutun 4 byte ile kodlandığı normak komut kümesindeki her komut thumb komutu olarak ifade

edilememektedir. Dolayısıyla thumb komut kümesinin normal komut kümesinin bir alt kümesi olduğu söylenebilir. Thumb komut kümesi kullanıldığında normak komut kümesine göre şu farklılıklar oluşmaktadır:

- Komutların çoğu 2 byte uzunluktadır ancak 4 byte uzunlukta komutlar da vardır.
- Thumb komut kümesinde pek çok komutta koşul kısmı yoktur
- Komutların çoğu üç yerine iki operand'lıdır.
- Komutlarda açıkça belirtilen yazmaçlar 16 yerine 8 tanedir. Pek az komut geri kalan 8 yazmacı açıkça kullanabilmektedir.

Thumb komut kümesinin iki önemli versiyonu vardır: Thumb1 (T1) ve Thumb2 (T2). Bu iki komut kümesinin encoding'leri farklıdır. Ancak nispeten yeni pek cortex Thumb2 komut kümesini kullanmaktadır.

GASM'de sembolik makine dili derleyicisinin 2 byte'lık Thumb komutları üretmesi için ".code 16" direktifinin kullanılması gerekir. Benzer biçimde ".code 32" direktifi de derleyicinin 4 byte'lık normak ARM kodu üretmesini sağlar. Bu direktifler kodun birden fazla yerinde kullanılabilir. Bir direktif diğeri gelene kadar o bölgede etkili olur.

Pekiyi ARM işlemcileri o anki komutun Thumb komutu mu yoksa normal ARM komutu mu olduğunu nasıl anlamaktadır? İşte işlemcinin bayrak yazmacındaki (Application Program Status Register) T biti işlemcinin o anda hangi komut kümesini çalıştırdığını belirtmektedir. Bu bit set edilmişse işlemci thumb kümesini, set edilmemişse normal ARM kümesini çalıştırır. Ancak bu bitin set edilmesi MOV işlemleriyle yapılmaz. Branch işlemleriyle yapılmaktadır.

ARM modundan Thumb moduna geçmek için BX, BL ya da BLX makine komutları kullanılır. Anımsanacağı gibi BX komutu yazmaç yoluyla dallanmak için, BL komutu fonksiyon çağırmak için kullanılan komutlardır. Dallanma sırasında hedef adres (yer değiştirme miktarı) tekse hem dallanma gerçekleşir hem de işlemci thumb moduna geçer. BLX komutu ise thumb komut kümesi için fonksiyon çağırma işlemi yapan özel bir komuttur. Bu komutta hedef adresin tek olması gerekmez. ARM modunda fonksiyon adreslerinin 4'ün katlarına hizalanmak zorunda olduğunu anımsayınız. Thumb modunda ise hizalama ikinin katlarına yapılmaktadır.

BX ve BL komutlarında biz hedef adresi tek adres olarak verirsek işlemci hedef adresi ikinin katlarına aşağıya yuvarlayarak thumb moduyla dallanmaktadır. Ancak BLX komutunda biz hedef adresi dördün katlarında tutarak da dallanmayı yapabiliriz. BLX komutu aslında mod değiştirmektedir. Yani eğer ARM modundaysak BLX ile thumb moduna, thumb modundaysak da ARM moduna geçeriz. BX komutu da benzer biçimde tek adres verildiğinde mod değiştirme işlemini yapmaktadır.

Thumb modunda LDM ve STR komutları kullanılabilir ancak bellek adresi belirten yazmaç opereandı ve küme parantezleri içerisindeki yazmaç listesi ancak R0-R7 arasındaki yazmaçlar olabilir. Anımsanacağı gibi PUSH ve POP komutları aslında LDMFD ve STMFD komutlarının SP'li eşdeğerleridir. Yani:

```
LDMFD   SP!, {yazmaç listesi}
```

komu ile:

```
PUSH    {yazmaç listesi komutu}
```

ve,

```
STMFD   SP!, {yazmaç listesi}
```

komu ile:

```
POP {yazmaç listesi komutu}
```

komutu eşdeğerdir. Ancak 16 bit modda yukarıda da belirtildiği gibi LDM ve STM komutlarında R0-R7 yazmacının kullanılması zorunlu turulmuştur. Thumb komut kümesinde PUSH işleminde istisna olarak LR yazmacı (numarası 14'tür ve yüksektir), POP komutunda da PC yazmacı (numarsı 15'tir ve yüksektir) kullanlabilmektedir. Yani thumb modunda biz:

```
PUSH    {R0, R1, LR}
```

gibi bir komut verebiliriz. Benzer biçimde:

```
POP {R0, R1, PC}
```

komutunu da uygulayabiliriz. Ancak GNU Sembolik makine dili derleyicileri her ne kadar aşağıdaki iki komutun opcode'ları yukarıdakilerle aynı olsa da bunları kabul etmemktedir:

```
LDMFD   SP!, {R0, R1, LR}
STMFD SP!, {R0, R1, PC}
```

Şimdi ARM modundan thumb moduna ve thumb modundan yeniden ARM moduna geçiş yapan bir program örneği verelim:

```
/* util.s */

.arm
.global _start


.code 16
.align 2     // thumb modu için zaten default durum

_bar:
    mov  r1, r2
    bx   lr

_foo:
    push{lr}
    mov      r0, r1
    mov      r1, r2
    add      r1, r2
    bl       _bar
    pop      {pc}

.code 32
.align 4          // ARM modu için zaten default durum

_start:

    blx      _foo

    mov      r0, #0
    mov      r7, #1
    swi      #0
```

Burada objdump -d ile aşağıdaki gibi bir çıktı elde edilir:

thumbswitch.o: elf32-littlearm

Disassembly of section .text:

```
00000000 <_bar>:
   0:   1c11        addsr1, r2, #0
   2:   4770        bx  lr

00000004 <_foo>:
   4:   b500        push{lr}
   6:   1c08        addsr0, r1, #0
   8:   1c11        addsr1, r2, #0
   a:   1889        addsr1, r1, r2
```

```
   c:   f7ff fff8    bl  0 <_bar>
  10:   bd00         pop {pc}
  12:   0000         movsr0, r0
  14:   e1a00000     nop          ; (mov r0, r0)
  18:   e1a00000     nop          ; (mov r0, r0)
  1c:   e1a00000     nop          ; (mov r0, r0)

00000020 <_start>:
  20:   fafffff7     blx 4 <_foo>
  24:   e3a00000     mov r0, #0
  28:   e3a07001     mov r7, #1
  2c:   ef000000     svc 0x00000000
```

Burada geri dönüş işleminin POP {PC} ile yapıldığına dikkat ediniz. Geri dönüş işlemi _foo içerisinde BX LR ile de yapılabilirdi. Ancak _foo fonksiyonu _bar fonksiyonunu çağırdığı için LR yazmacı stack'e push edilmiştir. LR yazmacı thumb modunda POP edilemediği için bunun yerine POP {PC} işlemi uygulanmıştır.

GCC derleyicisinde thumb koduyla derleme işlemi de yapılabilir. Bunun için -mthumb seçenceği kullanılır. Ancak bu seçenek kullanılırken bizim hiçbir gerçek sayı işlemi yapmamamız ya da gerçek sayı işlemlerini emülasyon yoluyla yapmamız gerekir. Çünkü thumb komut kümesinde gerçek sayı işlemleri bulunmamaktadır. Ayrıca biz gerçek sayı işlemleri yapmamış olsak bile bağlama aşamasında gerçek sayı işlemlerinin yapıldığı amaç dosyaları ya da kütüphane dosyalarını link işlemine sokamayız. Maalesef ARM'ın thumb komut kümesi için GCC'nin C kütüphanesi de yok gibi gözükmektedir. Sonuç olarak biz GCC'de -c seçeneği ile içerisinde gerçek işlemleri geçmeyen kodları ya da gerçek sayı emülasyonu yapılan kodları thumb olarak derleyebiliriz. Örneğin:

```
/* sample.c */

#include <stdio.h>

double add(double a, double b)
{
    return a + b;
}


int mul(int a, int b)
{
    return a * b;
}
```

Derleme işlemi şöyle yapılabilir:

```
gcc -c -mthumb -mfloat-abi=soft sample.c
```

sample.o dosyasının objdump çıktısı da şöyledir:

sample.o: elf32-littlearm dosya biÃ§emi

Disassembly of section .text:

```
00000000 <add>:
   0:   b590         push{r4, r7, lr}
   2:   b085         sub sp, #20
   4:   af00         add r7, sp, #0
   6:   60b8         str r0, [r7, #8]
   8:   60f9         str r1, [r7, #12]
   a:   603a         str r2, [r7, #0]
   c:   607b         str r3, [r7, #4]
   e:   68b8         ldr r0, [r7, #8]
  10:   68f9         ldr r1, [r7, #12]
  12:   683a         ldr r2, [r7, #0]
```

```
  14:   687b        ldr r3, [r7, #4]
  16:   f7ff fffe   bl  0 <__aeabi_dadd>
  1a:   1c03        addsr3, r0, #0
  1c:   1c0c        addsr4, r1, #0
  1e:   1c18        addsr0, r3, #0
  20:   1c21        addsr1, r4, #0
  22:   46bd        mov sp, r7
  24:   b005        add sp, #20
  26:   bd90        pop {r4, r7, pc}

00000028 <mul>:
  28:   b580        push{r7, lr}
  2a:   b082        sub sp, #8
  2c:   af00        add r7, sp, #0
  2e:   6078        str r0, [r7, #4]
  30:   6039        str r1, [r7, #0]
  32:   687b        ldr r3, [r7, #4]
  34:   683a        ldr r2, [r7, #0]
  36:   4353        mulsr3, r2
  38:   1c18        addsr0, r3, #0
  3a:   46bd        mov sp, r7
  3c:   b002        add sp, #8
  3e:   bd80        pop {r7, pc}
```

ARM İşlemcilerinde Gerçek Sayılarla İşlemler

ARM cortex'lerinin belirli modelleri gerçek sayı birimi içermektedir. Örneğin Cortex-A6'ların bir bölümü, Cortex A-7'lerin hepsi, ve sonraki sonraki cortex'ler genel olarak gerçek sayı birimine sahiptir. Yine M serisi Cortex'lerin (mikrodenetleyici biçimindeki Cortex-M serisi) bazılarında gerçek sayı birimi bulunur.

ARM işlemcilerinde gerçek birimleri tıpkı Intel'de olduğu gibi ek bir işlemciymiş gibi (coprocessor) biçiminde bulunur. Böylece bazı cortex'lere bu birim eklenip bazılarından çıkartılabilmektedir. Kullanılan gerçek sayı biriminin (fpu coprocessor) çeşitleri vardır. Bunların son modellerine NEON denilmektedir. Ancak gerçek sayı birimi ne olursa olsun komutlarda belli bir geriye doğru uyum vardır. Gerçek sayı komut kümesi çeşitli versiyonlara sahiptir. Gerçek sayı komut kümelerinin tarihsel gelişimi şöyledir:

VFPv1
VFPv2
VFPv3
VFPv3-D32
VFPv3-D16
VFPv3-F16
VFPv4
VFPv4-D32
VFPv4-D16
VFPv5-D16-M

Çalışma sırasında hangi cortex'lerin hangi komut kümesini desteklediğinin bilinmesi gerekebilir. Tabii yukarıda da belirtildiği gibi bu komut kümeleri arasında geriye doğru belli bir uyum vardır. Örneğin Raspberry Pi 2'de ARM Cortex-A7 kullanılmıştır. Bu işlemcinin içerisinde NEON gerçek sayı işlemcisi bulunur. Bunun desteklediği gerçek komut kümesi VFPv4'tür. (VFP sözcüğü "Vector Floating Point" sözcüğünden kısaltmadır.) Benzer biçimde Raspberry Pi 3'te Cortex-A53 işlemcisi (64 bit) kullanılmaktadır. Bu işlemci de NEON gerçek sayı birimine sahiptir ve VFPv4 komut kümesini kullanır. ARM'ın VFP komut kümesi "ARM Developer Suite Assembler Guide)" isimli ARM dokümanında bulunmaktadır.

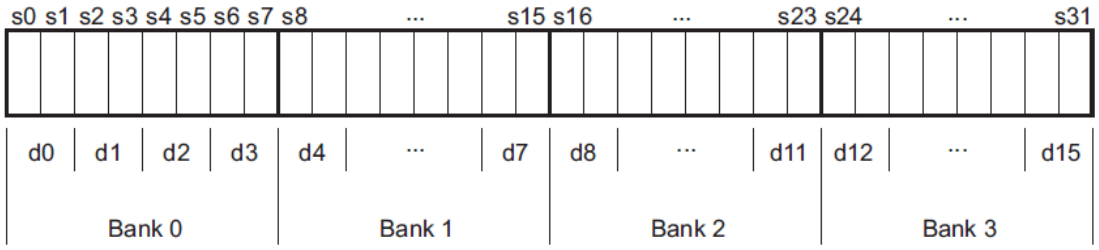
ARM VFP komut kümesinde gerçek sayı komutlarının başı F harfiyle başlamaktadır. Ancak ARM'ın akternatif "UAL (Uniform Assembler Language)" isimli yeni sentaksında bu komutlar V harfiyle başlar. Eski sentaksla yeni sentaks bunun dışındaki birkaç farklılık dışında çok benzerdir.

GNU'nun sembolik makine dili derleyicisinde gerçek sayı işlemleri için ".fpu vfp" direktifinin (bunun da VFP versiyonlarına göre seçenekleri vardır) bulunması gerekir.

ARM'ın gerçek birimleri SIMD biçiminde tasarlanmıştır. Yani istenirse tek bir değer üzerinde gerçek sayı işlemi yapılabileceği gibi istenirse tek bir komutla birden fazla gerçek sayı üzerinde işlemler yapılabilir.

VFP komut kümesinde 32 float genişliğinde (single), 16 double genişliğinde yazmaç vardır. Bu yazmaçlar gerçek sayı biriminin içerisindedir. Float yazmaçlar s0-s31 biçiminde, double yazmaçlar da do-d15 biçiminde numaralandırılmıştır. Ancak aslında double yazmaçlar iki float yazmacın bileşiminden oluşur. Yani örneğin s0 ve s1 yazmacı aslında d0 yazmacı, s2 ve s3 yazmacı d1 yazmacı biçimindedir.

Gerçek sayı yazmaçları 4 bank'e ayrılmaktadır. SIMD işlemleri ancak farklı bank'lerdeki yazmaçlar arasında yapılabilmektedir. s0-s7 ve d0-d3 sıfırıncı bank'i, s8-s15 ve d4-d7 birinci bank'i, s16-s23 ve d8-d11 ikinci bank'i ve s24-s31 ve d12-d15 de üçüncü bank'i oluşturmaktadır.



ARM gerçek sayı terminolojisinde paralel işleme sokulacak yazmaçlara vektör denilmektedir. Vektörel işlemlerde (SIMD işlemlerinde) farklı bank'lerde farklı yazmaçlardan başlanarak işlemler yapılabilir. Ancak bank'in sonuna gelindiğinde yeniden başa sarma yapılmaktadır. Örneğin biz s5'ten başlayarak 6 tane float sayı üzerinde işlem yapacak olsak bu float yazmaçlar s5, s6, s7, s0, s1, s2 olacaktır. Bnezer biçimde bank 1'de s15'ten başlayarak 3 tane float üzerinde işlem yapacak olsak bu yazmaçlar s15, s8, s9 olacaktır. Double değerlerde de yine sarmalama yapılmaktadır. Örneğin bank 2'de d10'dan itibaren 3 double sayı üzerinde işlem yapacak olsak bu yazmaçlar d10, d11, d8 olacaktır. Ancak bir vektör aynı yazmacı birden fazla kez içeremez. Örneğin bir bank'te 4'ten fazla double sayı üzerinde işlem yapılamaz. Çünkü bu durumda o bank'te aynı yazmaç birden fazla kez belirtilmiş gibi olmaktadır.

Gerçek sayı işlemcisinde vektör belirtirken "stride" kavramı da kullanılmaktadır. "Stride" atlama değerini belirtir. Örneğin stride = 1 ise yazmaçlar peşi sıra gider. stride = 2 ise yazmaçlar birer atlayarak gitmektedir. Örneğin bank 0'da stride = 2 durumunda s6'dan başlayan 4 uzunluklu vektördeki yazmaçlar s6, s0, s2 ve s4 olacaktır. Benzer biçimde 2 uzuluklu stride = 2 durumunda d1'den başlayan vektör d1 vde 3'ten oluşur.

VFP Komut Kümesinde Gerçek Sayı Komutlarının Genel Yapısı

VFP'de gerçek sayı komutları iki operand'lı ya da üç operand'lı olabilmektedir. Bunların genel biçimi aşağıdaki gibidir:

```
Op  Fd, Fn, Fm
Op  Fd, Fm
```

Bu komutlarda ilemlerin skaler mi (yani tek operand üzerinde işlem), vektörel mi (yani birden fazla operand'la paralel işlem) yoksa karışık mı (mixed) yapılacağı operand'lardaki yazmaçların bank'lerine bağlı olmaktadır. Üç durum söz konusu olabilir:

1) Eğer hedef operand (yani Fd) sıfırıncı bank'teyse diğer operand'lar herhangi bir bak'te olabilir ve bu durumda her zaman skaler işlem yapılır. FPSCR yazmacındaki vektör uzunluğunun ve stride değerinin bu durumda bir önemi kalmaz. Örneğin:

```
FADDS   S0, s9, s20
FMULD   D2, D7, D12
```

2) Birinci ve üçüncü operanda (iki openrad'lı da ikinci operand) yani Fd ve Fm ilk bank'te değilse işlem her zaman vektörel yapılır. Bu durumda vektör uzunluğu ve stride değeri FPSCR yazmacından elde edilmektedir. Örneğin:

```
FADDS   S8, S8, S16
FMULD   D4, D8, D12
```

3) Eğer üçüncü operand (iki operand'lı da ikinci operand) 0'ıncı bank'teyse ancak hedef operand 0'ıncı bank'te değilse bu durumda işlem karışık (mix) yapılmaktadır. Karışık işlem demek skalerle vektörün her elemanının işleme sokulması demektir. Örneğin uzunluk 8 olsun ve stride değeri de 1 olsun:

```
FMULS   S8, S8, S0
```

Burada aslında şu işlem yapılmaktadır:

```
{S8, S9, S10, S11, S12, S13, S14, S15} = S0 * {S8, S9, S10, S11, S12, S13, S14, S15}
```

Bemzer biçimde örneğin uzunluk 4 ve stride değeri 1 olmak üzere aşağıdaki komut uygulanmış olsun:

```
FADDD   D4, D8, D0
```

Burada D0'daki değer tek D8-D11 yazmaçlarıyla toplanarak karşıklı biçimde D4'ten itibaren yerleştirilir. Yani işlemin eşdeğeri şöyledir:

```
{D4, D5, D6, D7} = D0 + { D8, D9, D10, D11}
```

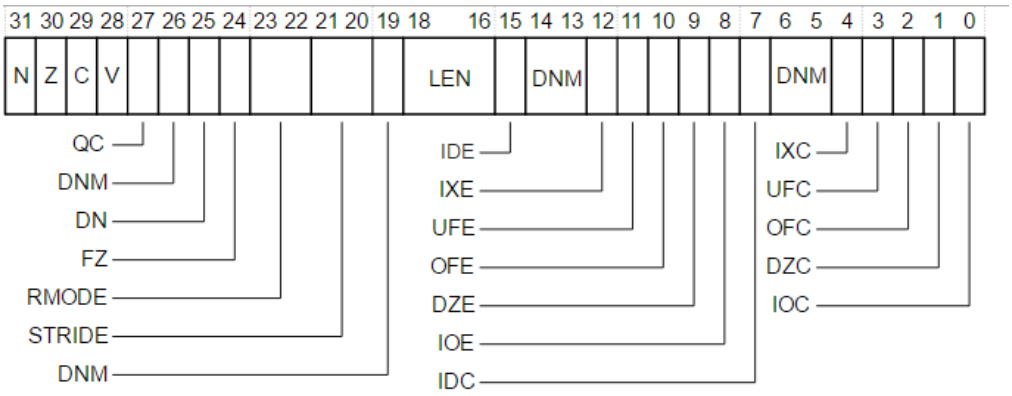
Gerçek sayı komutlarının çoğu yine koşul kısmına sahiptir. Komutların isimlerinin hemen sonuna ayrıca S ya da D ekleri getirilmektedir. Bu ekler işlemin float (single) düzeyinde mi yoksa double düzeyinde mi yapılacağını belirtir. u durumda komut isimlerinin genel oluşturulma biçimi şöyledir:

```
FLD<S ya da D>{koşul}
```

Koşul kısmına getirilecek ekler önceki bölğmlerde ele alınmıştı.

FPSCR Yazmacı

Gerçek sayı işlemcisinin ayrı bir durum yazmacı vardır. Buna FPSCR (Floating Point Status Control Register) denilmektedir. Bu yazmaç bitlerden oluşur. Yazmacın bitleri şöyledir:



Yazmacın yüksek anlamlı 4 biti (N, Z, C, V) karşılaştırma işlemi sonucunda dallanmalar için kullanılıri Aslında gerçek sayı komutları da durum koduna sahip olabilmektedir. Bu komutlar da yine bu bitleri dikkate alır. LEN bitleri vektörel işlemde işlemin uzunluğunu belirtir. Ancak uzunluk buradaki değerden bir fazladır. STRIDE için de iki bit ayrıldığına dikkat ediniz. Stride değeri buradaki değerden yine 1 fazladır. Şüphesiz vektörel işlem yaparken en azından programcının işlem skaler değilse bu yazmaçtaki LEN ve STRIDE bitlerini set etmesi gerekir. FPSCR yazmacının LEN ve STRIDE bitlerini uygun biçimde set etmek maalesef en az 4 makine komutuyla yapılabilmektedir. Önce FPSCR yazmacı FMRX komutuyla ana işlemcinin bir yazmacına çekilir. Sonra LEN ve STRIDE bitleri diğer bitlere dokunulmadan AND ya da BIC komutlarıyla 0'lanır. Sonra LEN ve STRIDE bitleri ORR komutuyla uygun biçimde set edilir. Nihayet sonuçta elde edilenm değer FMXR komutuyla yeniden gerçek sayı işlemcisinin FPSCR yazmacına atanır. Örneğin biz FPSCR yazmacını diğer bitlerine dokunmadan STRIDE = 1 ve LEN = 4 konumuna şöyle getirebiliriz:

```
FMRX    R10,FPSCR             ; copy FPSCR into r10
BIC     R10, R10, #0x00370000 ; clears STRIDE and LEN
ORR     R10, R10, #0x00030000 ; sets STRIDE = 1, LEN = 4
FMXR    FPSCR, R10            ; copy r10 back into FPSCR
```

BIC R0, R1, R2 komutunun R0 = R1 & ~R2 işlemini yaptığını anımsayınız.

ARM Mimarisinde Gerçek Sayı İçeren Kodlarda Gerçek Sayıların Fonksiyonlara Aktarılması ve Geri Döndürülmesi İle İlgili Çağırma Biçimleri

ARM mimarisinde bazı cortex'ler gerçek sayı birimi içermediği için GCC gibi derleyicilerde gerçek sayı işlemlerinin nasıl yapılacağı bazı seçeneklerle ayralanmaktadır. Bunlardan birincisi -mfpu=XXX seçeneğidir. Bu seçenek gerçek sayı işlemcisinin ne olduğunu başka bir deyişle hangi komut kümesini desteklediğini belirtmek için kullanılmaktadır. -mfpu için tipik seçenkler şunlardır.

vfpv2
vfpv3
vfpv4
neon-vfpv3
neon-vfpv4

Eğer seçenek olarak yalnızca -mfpu=vfp kullanılırsa bu -mfpu=vfpv2 ile aynı anlama gelmektedir. Eğer seçenek olarak yalnızca -mfpu=neon kullanılırsa bu da -mfpu=neon-vfpv3 anlamına gelmektedir. Eğer -mfpu hiç kullanılmazsa bu durumda GCC default duruma bakar. Default durum GCC'nin portlarına göre değişebilmektedir. Örneğin Raspberry Pi'da default durum -mfpu=vfpv2'dir. Ancak Windows çapraz derleyicisinde default durumda hiç gerçek sayı komutları kullanılmaz her şey emülasyonla yapılır.

Gerçek sayı içeren kodlarda gerçek sayı türlerinin fonksiyonlara aktarımı için GCC'de -mfloat-abi=xxx seçeneği kullanılır. Bu seçenekte üç farklı giriş yapılabilir:

```
soft
softfp
hard
```

soft seçeneğinde gerçek sayı aktarımı ana işlemcinin yazmaçlarıyla yapılır. Bu seçenekte zaten derleyici gerçek sayı komutlarını kullanmamaktadır. Gerçek sayı işlemleri için emülasyon yapar. softfp seçeneğinde derleyici gerçek sayı komutlarını -mfpu seçeneğine uygun olarak kullanır. Ancak aktarımı ana işlemcinin yazmaçlarıyla yapar. Aktarınm kuralı şöyledir:

- Aktarım için eğer R0-R3 yazmaçlarında yer kaldıysa bu yazmaçlar yoluyla yapılır. float değerler R0, R1, R2, R3 yazmaçlarıyla, double değerler R0-R1 ve R2-R3yazmaçlarıyla aktarıolır. Eğer R0-R3 yazmaçlarının kullanımı kalmadıysa geri kalan aktarım stack yoluyla yapılacaktır. Fonksiyoınun geri dönüş değeri de float ise R0 ile double ise R0-R1 ile aktarılır. Örneğin:

```
double foo(double a, double b, double c);
```

Burada a parametresi R0-R1 ile, b parametresi R2-R3 ile aktarılır. c parametresi de stack'push edilir. Geri dönüş değeri R0-R1'den alınacaktır. Örneğin:

```
float bar(int a, int b, float c, double d);
```

Burada a parametresi R0 ile, b parametresi R1 ile, c parametresi R2 ile aktarılır. d ise stack'e push edilir. Fonksiyonun geri dönüş değeri R0'dan alınacaktır.

hard seçeneğinde aktarımda ve geri dönüş değerinde matematik işlemcinin yazmaçları kullanılır. Şöyle ki:

- float aktarımlar s0, s1, s2.. yazmaçlarıyla
- double aktarımlar d0, d1, d2, ... yazmaçlarıyla yapılır.

- Geri dönüş değeri float ise s0 ile double ise d0 ile aktarılır.

Örneğin:

```
double foo(double a, double b, double c);
```

Burada a parametresi d0 ile, b parametresi d1 ile ve c parametresi d2 ile aktarılır. Geri dönüş değeri de d0'dan alınacaktır. Örneğin:

```
float bar(float a, double b, double c);
```

Burada a parametresi s0 ile, b parametresi d1 ile (çünkü d0'da yer kalmadı) ve c parametresi de yine d2 ile aktarılır. Geri dönüş değeri s0'dan alınır.

GCC ile derleme yaparken -mfloat-abi seçeneği girilmezse default durum GCC'nin portlarına göre değişebilmektedir. Örneğin Raspberry Pi'da default durum "hard" biçimindedir. Yani aktarım matematik işlemcinin yazmaçlarıyla yapılmaktadır. Halbuki Windows'taki çapraz derleyicide default durum softfp biçimindedir.

Burada Raspberry Pi için şöyle bir notu düşelim: Raspberry Pi'da maalesef standart C kütüphaneleri "hard" olarak derlenmiştir. Bu nedenle biz -mfloat-abi=soft ya da -mfloat-abi=softfp ile derleme yaptığımızda bağlama aşamasında kütüphanenin kullandığı çağırma biçimiyle uyuşmazlık ortaya çıkltığından error oluşmaktadır.

Önemli Gerçek Sayı Komutları

Yukarıda da belirtildiği gibi ARM sembolik makine dili için kullanılan iki sentaks vardır. Bir tanesi eskiden beri kullanılan sentakstır. İkincisi de UAL (Uniform Assembly Language) denilen daha yeni sentakstır. Eski ve yeni sentaks birbirlerine çok benzemektedir. Gerçek sayı işlemleri söz konusu olduğunda eski ve yeni sentaks arasındaki en önemli farklılık eski sentaksta komutların F harfiyle başlaması yeni sentaksta ise V harfiyle başlamasıdır. Ayrıca yeni sentaksta işlem genişliği S ya da D soneki ile değil, .F32 ve .F64 sonekiyle belirtilmektedir. GNU'nun ARM derleyicisinde hangi sentaksın kullanılacağı .syntad diektifiyle belirlenir:

```
.syntax [unified | divided]
```

Default durum yeni GNU sembolik makine dili derleyicilerinde UAL sentaksıdır. Ancak bu sentaksta gerçek sayı komutları eski ya da yeni sentaksla (yani F'li ya da V'li sentaksta) verilebilmektedir.

Ayrıca GNU sembolik makine dili derleyicisinde gerçek sayı komutlarını kullanabilmek için ".fpu vfp" direktifinin (vfp yerine vfpv2, vfpv3, vfpv4 gibi version numaraları da getirilebilir. vfp demekle vfpv2 anlaşılmaktadır) verilmesi gerekir.

Şüphesiz öncelikle gerçek sayı birimindeki yazmaçlara bellekten değer yükleyen ve oradaki değerleri de belleğe yükleyen komutların öncelikle ele alnması gerekir.

FMOV (VMOV) Komutları

Bu komut gerçek sayı biriminin yazmaçları arasında atama yapmak için ya da gerçek sayı yazmaçlarıyla asıl yazmaçlar arasında atama yapmak için kullanılır. Örneğin:

```
VMOV.F64    D0, D1
```

Burada D1'deki değer V0'a atanmıştır. Benzer biçimde:

```
VMOV.F32    S0, S1
```

Örneğin:

```
VMOV.F32        S0, R0
VMOV.F32        R0, S0
```

Burada asıl işlemcinin yazmaçlarıyla matematik işlemcinin yazmaçları arasında float ataması yapılmıştır. 64 bit atama için iki asıl yazmacın verilmesi gerekir. Örneğin:

```
VMOV.F64    D0, R0, R1
```

Burada R0 ve R1 yazmaçlarının içerisindeki 64 bitlik double değer matematik işlemcinin D0 yazmacına atanmıştır. Atama yönü ters de olabilirdi:

```
VMOV.F64    R0, R1, D0
```

Aslında asıl iki yazmaç matematik işlemcinin iki float yazmacına da atanabilir:

```
VMOV.F32    S0, S1, R0, R1
VMOV.F32    R0, R1, S0, S1
```

Örneğin:

```
/* util.s */

.arm
.fpu vfp
.global foo

.text
foo:
    ldr      r0, .L1
    ldr      r1, .L1 + 4

    ldr      r2, .L1 + 8
    ldr      r3, .L1 + 12

    vmov.f64 d0, r0, r1
    vmov.f64 d1, r2, r3
    vadd.f64 d0, d0, d1

    bx           lr

.L1:
    .word    0
    .word    1076101120        // 10
    .word    0
    .word    1077149696        // 20

/* sample.c */

#include <stdio.h>

double foo(void);

int main()
{
    printf("%f\n", foo());

    return 0;
}
```

Derleme şöyle yapılabilir:

```
gcc -o sample sample.c util.s
```

VLDR ve VSTR Komutları

Bu komutlar adeta LDR ve STR komutlarının matematik işlemci versiyonudur. Yani bellekteki belli bir bilgiyi matematik işlemcinin yazmaçlarına yüklemekte kullanılır. Komut eski versiyonda FLDS ya da FLDD biçimindedir. Bellek operandı yalnızca [RN, #sayı] biçiminde oluşturulur. Yine yazmaç olarak PC kullanılabilir. Örneğin .data bölümündeki 64 bitlik bir double değer matematik işlemcinin yazmacına şöyle yüklenebilir:

```
/* util.s */

.arm
.fpu vfp

.data

.L1:
    .word 0x12345678
    .word 0x12345678
.text
_start:
    ldr          r0, =.L1
    vldr.f64     d0, [r0]
    //...
    mov      r0, #0
    mov      r7, #1
    swi      #0
```

Diğer Komutlar

Gerçek sayı işlemlerini yapan pek çok komut vardır. Bunların tam listesi ARM dokğmanlarından elde edilebilir. Aşağıdaki komutlar tipik olanlarıdır:

VADD
VMUL
VDIV
VSUB
VSIN
VCOS
VABS

ARM İşlemcilerinde Çalışma Modları

ARM işlemcilerinde “çalışma modu” kavramı kod önceliği (privilege) ve “yazmaç kullanma yeteneği”nin birleşiminden oluşmaktadır. Her çalışma modunun bir erişim hakkı (privilege) ve kullanabildiği yazmaç kümesi vardır. Intel mimarisindeki gibi komut koruması (intruction protection) yerine ARM işlemcilerinde yazmaç koruma (register protection) yöntemi benimsenmiştir. Zaten sistem derecesinde öneli işlemler bazı kontrol yazmaçlarının bitlerini değiştirerek yapılmaktadır. Dolayısıyla yazmaç koruması bir anlamda komut koruması gibidir.

ARM mimarisinde 9 çalışma modu vardır:

Table B1-1 ARM processor modes

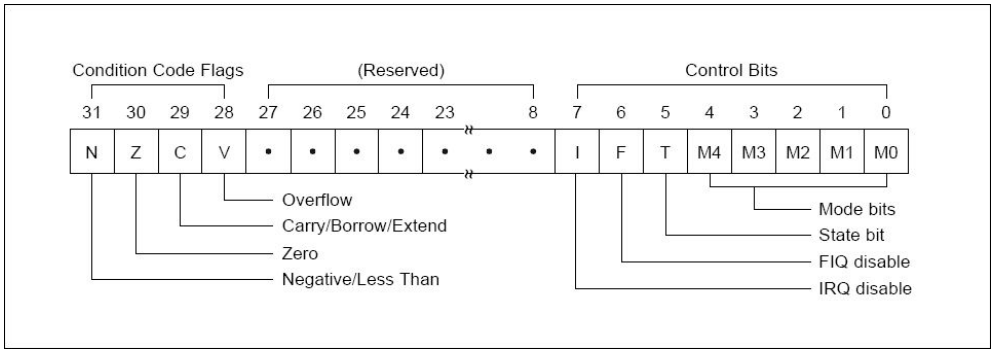
Processor mode		Encoding	Privilege level	Implemented	Security state
User	usr	10000	PL0	Always	Both
FIQ	fiq	10001	PL1	Always	Both
IRQ	irq	10010	PL1	Always	Both
Supervisor	svc	10011	PL1	Always	Both
Monitor	mon	10110	PL1	With Security Extensions	Secure only
Abort	abt	10111	PL1	Always	Both
Hyp	hyp	11010	PL2	With Virtualization Extensions	Non-secure only
Undefined	und	11011	PL1	Always	Both
System	sys	11111	PL1	Always	Both

ARM dokümanlarında her mod üç harfli bir kısaltmayla ifade edilmektedir (tablonun ikinci sütunu).

ARM mimarisinde toplam üç öncelik derecesi vardır. Bunlar PL0, PL1 ve PL2 öncelikeridir. PL2 sanallaştırma moduyla ilgilidir ve burada ihmal edilecektir. O halde önemli iki öncelik PL0 ve PL1 önceliklerdir. PL0 önceliğine kullanıcı önceliği (user privilege), PL1'de sistem önceliği (system privilege) de denilmektedir. Genel olarak kullanıcı modunda çalışan prosesler PL0 önceliğine ilişkindir. Bu Intel'in CPL = 3'üne karşılık getirilebilir. Çekirdek modunda çalışan kodlar ise PL1 önceliğine sahiptir. Bu da Intel'deki CPL = 0'a karşı gelmektedir. PL0 önceliğinde çalışan kodlar bellek korumasına tabi tutulurlar. PL1 kodları ise genel olarak koruma engeline takılmazlar.

Yukarıdaki çalışma modlarından en önemlileri User, System ve Supervisor modlarıdır. User modunda proses kurs içerisinde sözü edilen temel yazmaçları kullanabilmektedir. Bu mod programlar PL0 önceliğine sahiptir. Linux ve türevleri gibi işletim sistemlerinde sıradan uygulama prosesleri bu moda sahiptir. System Modunda proses PL1 önceliğine sahip olur. Koruma engeline takılmaz. Ancak özel kontrol yazmaçlarını kullanamaz. System modunda kullanılabilen yazmaçlar User modundakilerle aynıdır. Suğervisor modunda ise proses PL1 önceliğine sahiptir. Yani koruma engeline takılmaz. Aynı zamanda tüm kontrol yazmaçlarını da kullanabilir. Dolayısıyla mod değiştirme işlemi User modda ya da System modunda yapılamaz. Ancak Supervisor modunda yapılabilir.

Bayrak yazmacı olan CPRS'nin düşük anlamlı 5 biti modu belirtmektedir. Mod bu beş bit set edilerek değiştirilir. Ancak User ve System modlarında (ve diğer bazı modlarda) bu yazmacın bu bitlerine erişilememektedir.



İşlemci reset edildiğinde çalışma Supervisor modda başlar. Daha sonra CPSR yazmacı kullanılarak çalışma modları dinamik olarak değiştirilebilmektedir. ARM mimarisinde üç tür kesme oluşturulabilmektedir. Birincisi yaılım kesmesidir. Bu kesme SWI komutuyla oluşturulur. Donanım kesmeleri Hızlı (Fast) ve Normal olmak üzere ikiye ayrılmaktadır. Bunlar için CPU'da iki ayrı uç bulunmaktadır. Hızlı kesme oluştuğunda çalışma otomatik olarak FIQ moduna, normal kesme oluştuğunda da IRQ moduna geçmektedir.

ARM Mimarisinde Kesme İşlemleri

Yukarıda da belirtildiği gibi ARM mimarisinde yazılım ve donanım kesmeleri oluşturulabilmektedir. Bir kesme oluştuğunda bayrak yazmacı (CPSR) ve Link yazmacı (R14) otomatik olarak bunların gölge yazmaçlarında saklanır. Akışın kesme koduna gitmesi yine Intel'deki gibi kesme vektörü yoluyla yapılmaktadır. Ancak kesme vektörü 8 elemanlıdır ve her kesme türü için bir tane vektör girişi vardır. Bu anlamda kesmelerin numaraları yoktur. Kesme vektörünün yeri ya 0X00000000 adresinde (yani belleğin tepesinde) ya da 0xFFFF0000 adresinde bulunmaktadır. Vektörün nerede olduğu SCTLR (System Control Register) yazmacının V bitine bağlıdır. Bu V biti 0 ise kesme vektörü 0x00000000 adresinde 1 ise 0xFFFF0000 adresindedir. Vektörün 0 adresinde olması en doğal durumdur. Çünkü işlemci reset edildiğinde zaten V biti de 0 olmaktadır.

Interrupt/Exception/Reset	Mode	Address
Reset	SVC	0x00000000
Undefined instruction	UND	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort	ABT	0x0000000c
Data abort	ABT	0x00000010
Reserved	*N/A*	*0x00000014*
IRQ	IRQ	0x00000018
FIQ	FIQ	0x0000001c

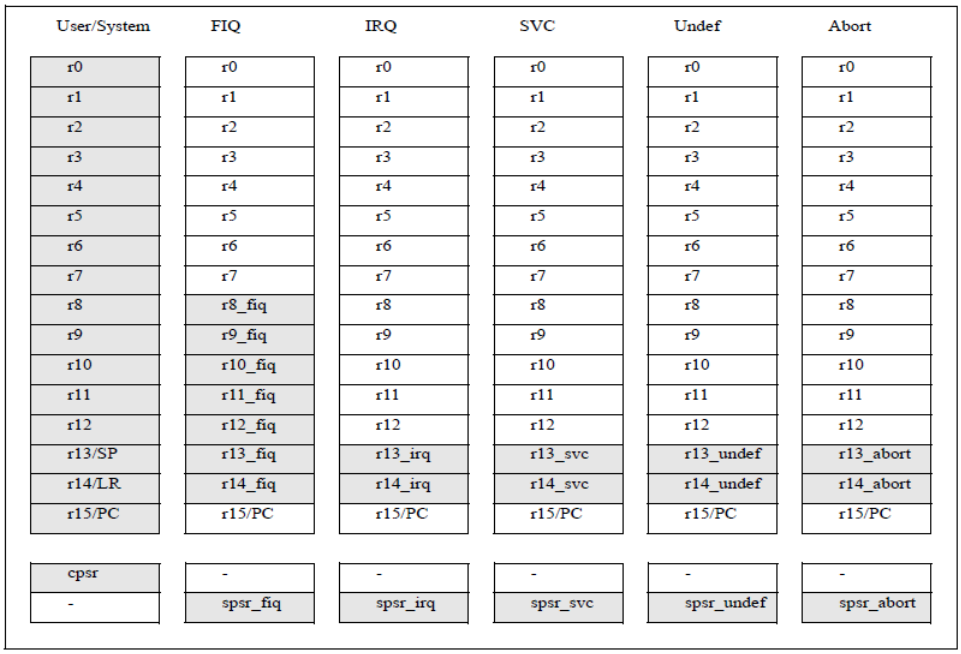
Vektörün elemanlarının dörder byte olduğuna dikkat ediniz. Yukarıdakş tabloda kesmenin oluşma biçimine göre hangi vektör elemanın kullanılacağı belirtilmiştir. Kesme oluştuğunda ARM işlemcileri kesmenin türüne göre akışı ilgili vektör elemanına aktarır. Örneğin SWI komutunda akış 0x00000008 numaralı adrese, normal donanım kesmelerinde ise 0x00000018 adresine aktarılmaktadır. Tabii buradaki dört byte'a kesme kodu yerleştirilemeyeceğine göre bu slotlarda ancak bir Branch komutu bulundurulabilir. Bu durumda gerçek kesme komutları dallanılan yerde tutulacaktır. Intel mimari ile kıyaslandığında iki farklılık dikkati çekmektedir:

1) Kesme vektöründe dallanılacak adres yoktur. Kesme oluştuğunda işlemci bizzat kesme vektörüne dallanmaktadır.
2) Kesme numarası kavramı yoktur. Kesme vektörü kesmenin türlerine göre elemanlara sahiptir.

Kesme vektörüne yerleştirilecek iki branck komutu tipik olarak şunlar olabilir:

```
B       etiket
LDR     PC, [PC, #displacement]   // Eşdeğeri LDR, PC, etiket
```

Oluşan kesmenin türüne göre kullanılabilen gölge yazmaçlar (banked registers) şöyledir:



User/System	FIQ	IRQ	SVC	Undef	Abort
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	r8_fiq	r8	r8	r8	r8
r9	r9_fiq	r9	r9	r9	r9
r10	r10_fiq	r10	r10	r10	r10
r11	r11_fiq	r11	r11	r11	r11
r12	r12_fiq	r12	r12	r12	r12
r13/SP	r13_fiq	r13_irq	r13_svc	r13_undef	r13_abort
r14/LR	r14_fiq	r14_irq	r14_svc	r14_undef	r14_abort
r15/PC	r15/PC	r15/PC	r15/PC	r15/PC	r15/PC
cpsr	-	-	-	-	-
-	spsr_fiq	spsr_irq	spsr_svc	spsr_undef	spsr_abort

Figure 1.7 Register organization

Yukarıdaki şekilde görüldüğü gibi her modun ayrı gölge (banked) yazmaçları vardır ve bu yazmaçların miktarları kesmenin türüne göre değişebilmektedir. Örneğin Tipik yazılım ve donanım kesmelerinde(IRQ) r13_irq (SP), R14_irq (LR) ve SPSR_irq yazmaçları gölge yazmaçlar olarak kullanılabilmektedir. Kesme oluştuğunda işlemci bu asıl yazmaçların değerlerini gölge yazmaçlara kopyalamaktadır. Böylece kesme oluştuğunda bayrak yazmacı, stack yazmacı ve link yazmacı saklanmış olmaktadır. Kesme oluştuğunda LR'nin değeri geri dönüş için önemlidir. Kesme oluştuğunda link yazmacının (LR) konumu oluşan kesmeye göre değişebilmektedir. Kesme oluştuğunda LR yazmacının değeri kesmenin türüne göre o anda çalıştırılan komutun adresi, o anda çalıştırılan komutun adresinden 4 ya da 8 ileri olabilmektedir. Bu ARM işlemcilerinin tasarımıyla ilgilidir. Aşağıdaki tabloda hangi türden kesmeler oluştuğunda LR yazmacının sonraki komuttan ne kadar ileri olduğu ve kesmeden geri dönüş için hangi komutun uygulanabileceği verilmiştir:

Event	**Offset**	**Return from handler**
Reset	n/a	n/a
Data Abort	-8	`SUBS pc,lr,#8`
FIQ	-4	`SUBS pc,lr,#4`
IRQ	-4	`SUBS pc,lr,#4`
Pre-fetch Abort	-4	`SUBS pc,lr,#4`
SWI	0	`MOVS pc,lr`
Undefined Instruction	0	`MOVS pc,lr`

Figure 1.13 Pointer counter offset

Bu tabşloda örneğin bir bir yazılım kesmesinden dönüyorsak kesme kodunun sonunda MOVS PC, LR işlemini yapmamız gerekmektedir. (Komuttaki S sonekinin gerekliliğine dikkat ediniz. Fakat biz örneğin donanım kesmesinde dönceksek bu durumda SUBS PC, LR, #4 işlemini uygulamalıyız.

ARM mimarisinde bir kesme oluştuğunda tıpkı Intel'de olduğu gibi işlemci normal ve hızlı kesmelere otomatik olarak kapatılmaktadır. ARM işlemcilerinde Intel'deki bayrak yazmacının IF bitine karşılık gelecek biçimde CPSR yazmacında I ve F bitleri vardır. I biti (Interrupt Flag) biti 1 ise işlemci donanım kesmelerine kapalıdır, 0 ise açıktır. Benzer biçimde F biti (Fast Interrupt Flag) 0 ise işlemci hızlı kesmelere açık 1 ise kapalıdır. Bu durumda default olarak kesmeler iç içe geçmemektedir. Yani kesme kodu çalışırken başka bir kesme gelememektedir. Ancak sistem programcısı tıpkı Intel'de olduğu gibi kesme kodunun hemen başında işlemciyi U ve F bayraklarını reset ederek yeniden kesmeye açabilir. Bu durumda iç içe (nested) kesme oluşturabilir. Şüphesiz iç içe kesmeleri organize ederken sistem programcısının SP, LR ve CPSR

gibi yazmaçları artık koruması gerekmektedir. Bu koruma işlemi kesme kodunun başında bu yazmaçların stack'e atılmasıyla yapılabilir.

ARM mimarisinde her ne kadar kesmelerin bir numarası yoksa da bu numaralandırma yapay olarak kullanılan kesme denetleyicisi yoluyla yapılabilir. Örneğin işlemciye bir kesme denetleyicisi bağlanabilir. Bu kesme denetleyicisinin giriş uçları çeşitli donanım kaynaklarına verilir. Böylece bu kaynaklar kesme denetleyicisi yoluyla kesme oluşturabilirler. Sistem programcısı da kesme denetleyicisne sorarak olulan kesmenin hangi birim tarafından oluşturulduğunu anlayabilir. Benzer biçimde yazılım kesmelerinde SWI (ya da SVC) komutlarında kesmenin numarası belli bir yazmaca yüklenebilir. Böylece kesme kodu o yazmaca bakarak kendi içerisinde dallanma yapabilir. ARM mimarisinde çalışan Linux sistemlerinde sistem fonksiyon numaralarının SWI komutu öncesinde R7 yazmacına yüklendiğini diğer parametrlerin de sırasıyla R0, R1, R2, R3 yazmaçlarına yüklendiğini anımsayınız.

32 Bit ARM İşlemcilerinde Sayfalama Mekanizması

32 Bit ARM işlemcilerinde sayfalama mekanizması Intel işlemcilerine benzer biçimlerde işletilmektedir. ARM mimarisinde de Intel'de olduğu gibi sayfalamanın çeşitli modları vardır. Genel sistem Intel'e oldukça benzemektedir. Yine bir yazmaç birinci düzey dönüştürme tablosunun (Intel'deki sayfa dizini tablosu) fiziksel adresini tutar. Sonra oradan çekilen eleman sonraki tablolara indeks yapılarak nihai fiziksel sayfanın yeri bulunur. ARM'da kısa ve uzun sayfalar kullanılabilmektedir. Sayfa tablolarının girişleri 4 byte ya da 8 byte olabilmektedir. 32 bit ARM işlemcilerinin pek çoğu fiziksel ve sanal 40 bit adresleme (1 TB) yapabilmektedir.

32 bit ARM işlemcileri reset edildiğinde sayfalama mekanizması aktif durumda değildir. Sayfalama mekanizmasını aktif duruma getirmek için SCTLR yazmacının M biti 1'lenir. Bnezer biçimde sayfalama modundan çıkmak için de bu bitin 0'lanması gerekmektedir.

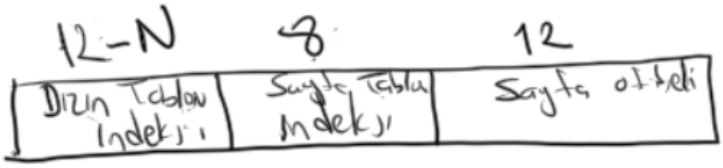
32 Bit ARM işlemcilerinde sayfa tablolarının girişleri (descriptor) 32 bit ve 64 bit olabilmektedir. Burada önce 32 bitlik sonra 64 bitlik girişler ele alınacaktır.

32 Bit Girişleri (Descriptors) Kullanan Sayfalama Mekanizması

32 bitlik sayfa girişleri (descriptor) kullanıldığında işlemci 32 bitlik sanal adresleri 32 bitlik fiziksel adreslere dönüştürebilmektedir. (32 bitlik ARM işlemcilerinin bir bölümü 40 bitlik fiziksel adresler kullanabilmektedir.) 32 bitlik sayfa girişleri için dört seçenek söz konusudur: 4KB'lik küçük sayfalar, 64KB'lik büyük sayfalar, 1MB'lik bölümler (section), 64MB'lik Bölümler.

4K'lık Sayfalardaki Organizsyon

4KB'lik sayfalar en çok kullanılan sayfalardır. İşletim sistemleri genel olarak Intel mimarisinde olduğu gibi 4KB'lik sayfaları kullanmaktadır. 4 KB'lik sayfalarda sanal adresin bileşenleri şöyledir:



Buradaki N değeri 0 olabilir ya da en fazla 7 olabilir. Bu değerin ne olacağı eğer dizin tablosunun adresi TTBR1 yazmacı tarafından gösteriliyorsa TTBCR yazmacının N bitlerinden elde edilmektedir. Eğer dizin tablosunun adresi TTBR0 yazmacı tarafından gösteriliyorsa N değeri 0'dır. N değerinin yüksek olması sanal adres kapasitesinin düşmesi anlamına gelir. İşletim sistemleri sanal adres kapasitesine düşürerek çeşitli metadata tablolarının daha az yer kaplaması sağlayabilir. Ancak Raspbian gibi pek çok işletim sisteminde N = 0'dır.

Buradan N = 0 için şu sonuçlar çıkartılabilir:

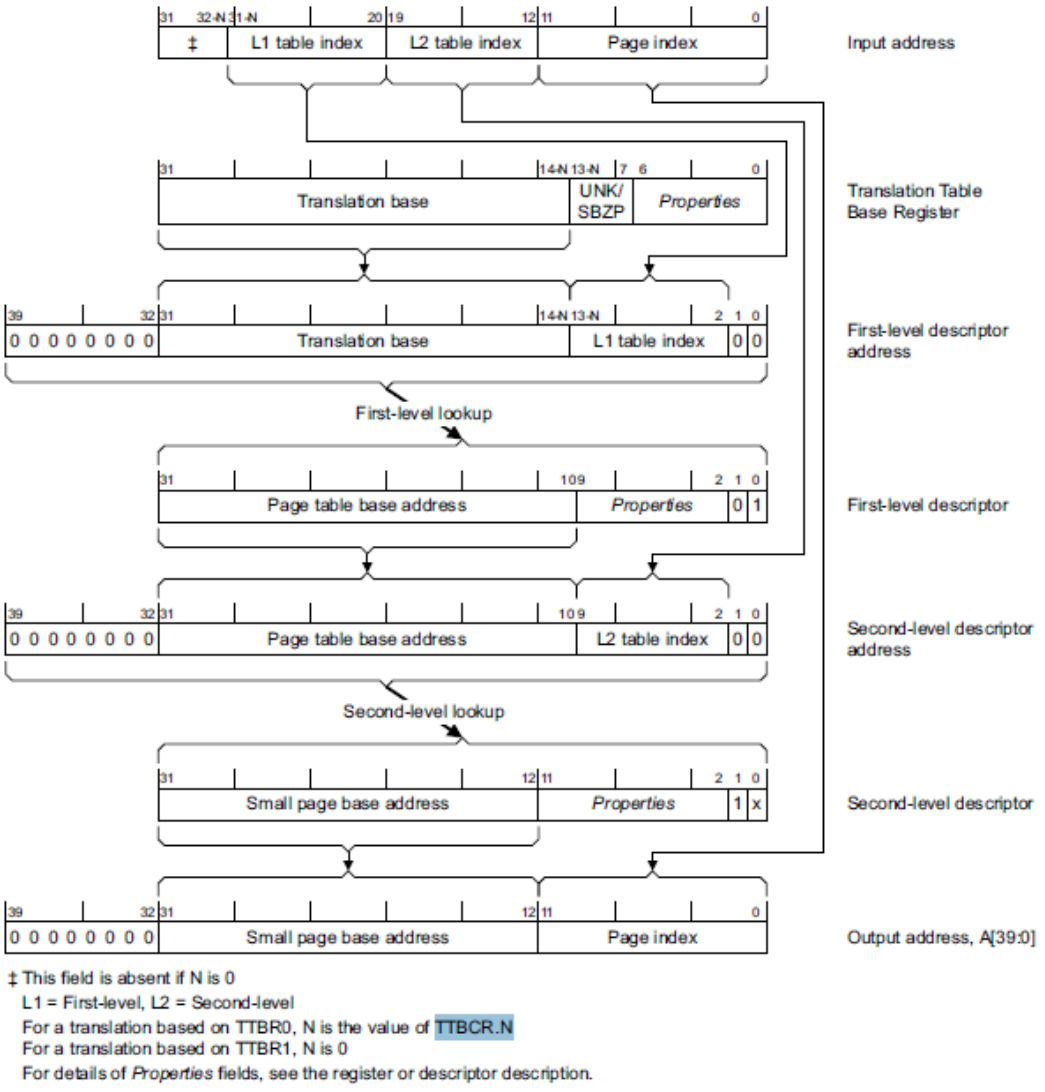
1) Dizin tablosu (birinci kademe dönüştrme tablosu) 16KB'ye hizalanmak zorundadır. TTBR0 ya da TTBR1 yazmaçlarında dizin tablosunun yeri için 18 bit yer ayrılmıştır. Yani dizin tablosunun fiziksel adresi bu 18 bitlik değerin 16K ile (2^{14}) ile çarpılmasıyla elde edilmektedir. Dizin tablosu toplam 4096 elemanlıdır. Bunun kapladığı yer de 16KB'dir.

2) Sanal adresin yüksek anlamlı 12 biti dizin tablosunda (birinci kademe dönüştürme tablosunda) bir indeks belirtir.

3) Dizin tablosunun (birinci kademe dönüştürme tablosunun) elemanları dörder byte'tır. Ve bu dörder byte'ların 22 bitleri sayfa tablolarının fiziksel bellekteki yerini gösterir. Sayfa tabloları 1KB'ye hizalanmak zorundadır. Bu durumda sayfa tablosunun fiziksel adresi dizin tablosu girişindeki 22 bitlik değerin 1KB ile (2^{10}) çarpılmasıyla elde edilir. ($2^{22} * 2^{10} = 2^{32}$ olduğuna dikkat ediniz.) Sayfa tabloları toplam 256 elemanlıdır ve bellekte 1KB yer kaplamaktadır.

4) Sanal adresin ortadaki 8 biti sayfa tablosunda bir indeks belirtmektedir. Sayfa tablosu girişleri de dörder byte'tır. Bu dört byte'ların 20 bitleri yine ilgili sayfanın fiziksel bellekteki başlangıç adresi belirtir. Bu 20 bit ile belirtilen indeksler 4KB ile (2^{12}) ile çarpılarak fiziksel adres elde edilmektedir.

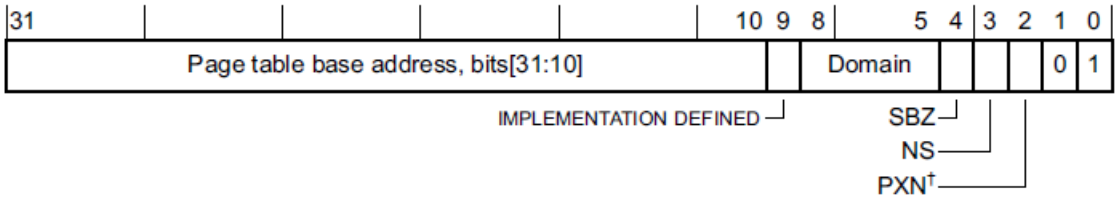
5) Sanal adresin düşük anlamlı 12 biti 4KB'lik sayfalarda offset belirtmektedir.



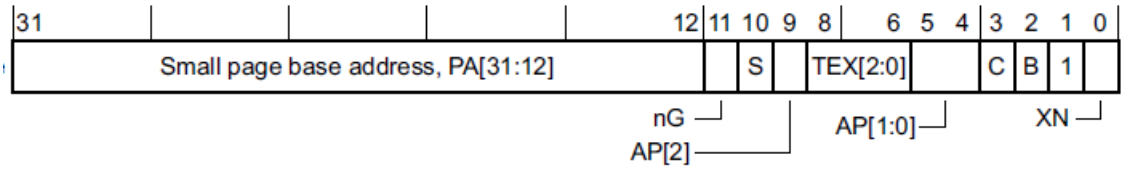
‡ This field is absent if N is 0
L1 = First-level, L2 = Second-level
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description.

Figure B3-11 Small page address translation

Dizin tablosu elemanları aşağıdaki gibi oluşturulmuştur:

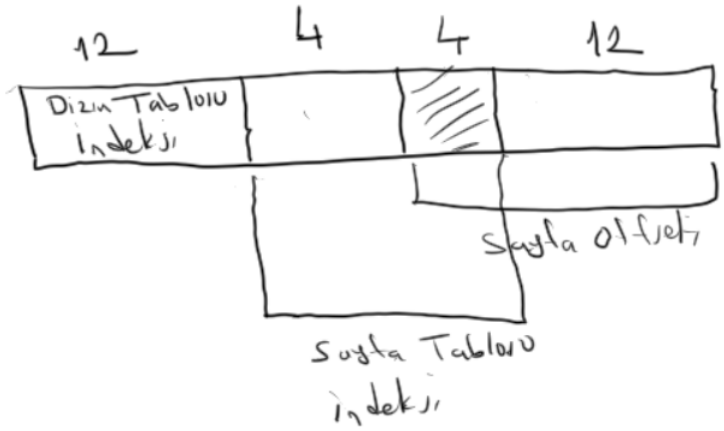


Sayfa tablosu elemanları da şöyle oluşturulmuştur:



64K'lık Sayfalardaki Organizasyon

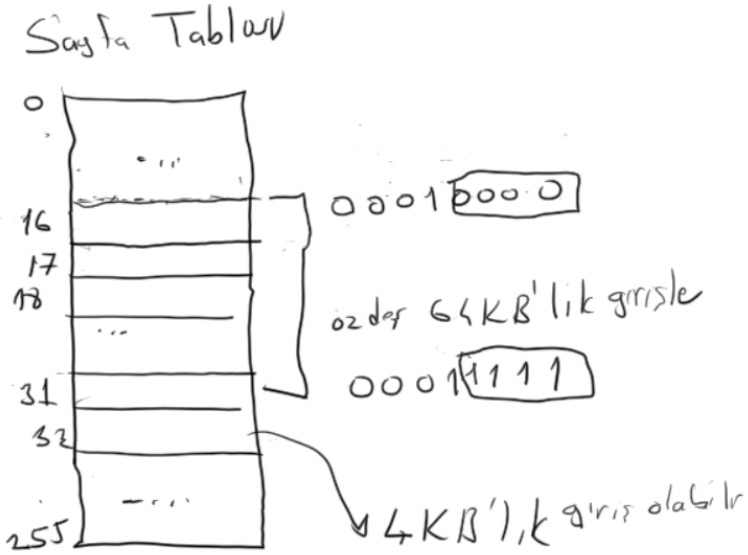
ARM mimarisinde 64KB'lik büyük sayfalar da kullanılmaktadır. Büyük sayfalar kullanılırken sanal adres şöyle bölümlere ayrılmaktadır:



Burada "Sayfa Tablosu Indeksi" ile "Sayfa Offseti" alanlarındaki 4 bitin çakışık olmasıtuhaf gözükebilir. Sanal adresin fiziksel adrese dönüştürülmesi şöyle yapılmaktadır:

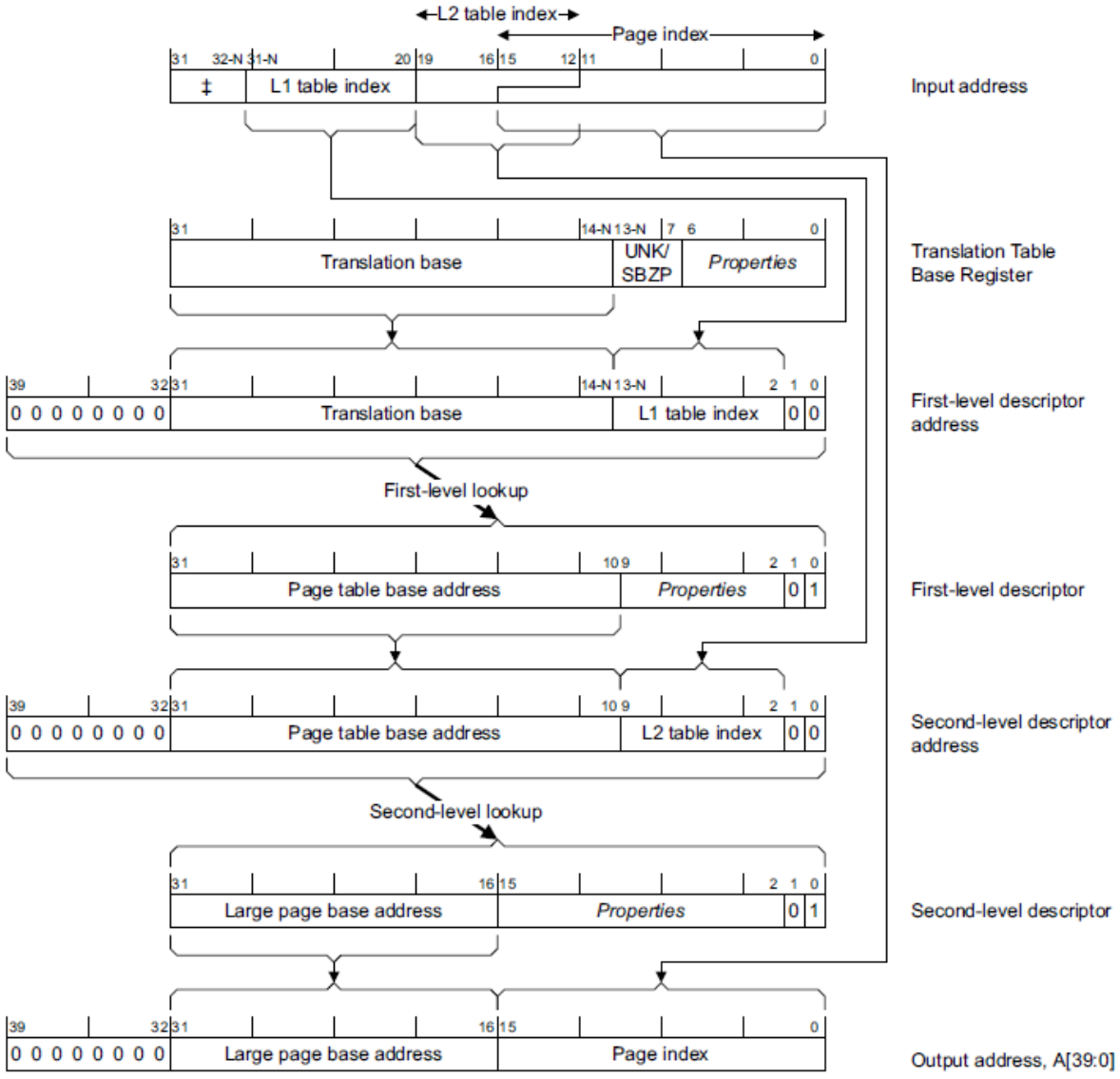
1) Yine işlemci TTBR0 ya da TTBR1 yazmacının gösterdiği yerde dizin tablosuna erişir. Sanal adresin yüksek anlamlı 12 biti yine bu tabloda bir indeks belirtmektedir. Bu aşamanın 4KB'lik sayfaların kullanımındaki aşamadan bir farkı yoktur.

2) Dizin tablosundan elde edilen giriş sayfa tablosunun fiziksel bellekteki yerini gösterir. Sanal adresin ortadaki 8 biti bu tabloya indeks yapılarak sayfanın fiziksel adresi elde edilecektir. ARM mimarisinde aslında sayfa tablosunda 4KB'lik sayfa girişleri ile 64 KB'lik sayfa girişleri beraber bulunabilmektedir. 64KB'lik sayfa girişleri sayfa tablosunda 16 elemanlık yer kaplar. Şöyle ki: Eğer 64KB'lik sayfa kullanılıyorsa sayfa tablosunun 16 girişi tamamen aynı değerleri tutacak biçimde yinelenmektedir. Böylece aslında sanal adresteki sayfa tablosunda indeks belirten 8 bitin düşük anlamlı 4 biti ne olursa olsun aynı giriş elde edilmektedir.



Sayfa tablosundaki 64KB2lik girişlerin 16'nın katlarına hizalanması zorunludur. Görüldüğü gibi işlemci aslında dönüştürme işlemine sanki 4KB'lik sayfalar kullanılacakmış gibi başlar, sayfa tablosundan çektiği sayfa girişlerinin düşük anlamlı 2 biti sayfanın hangi uzunlukta olduğunu vermektedir.

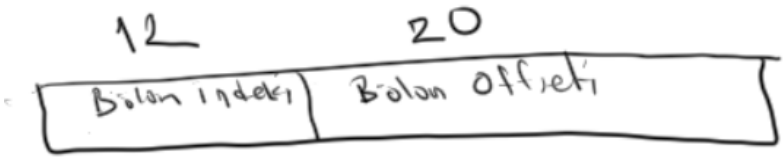
3) Artık işlemci sayfanın türünü belirledikten sonra sanal adresin düşük anlamlı kaç bitini sayfa offstei offset olarak kullanacağını anlar. Eğer 64KB'lik sayfalar söz konusuysa sayfa offset'i için 16 bit kullanılacaktır.



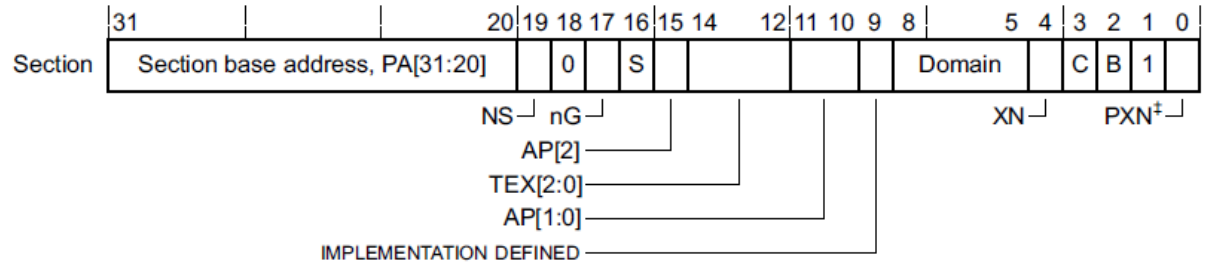
‡ This field is absent if N is 0
L1 = First-level, L2 = Second-level
For a translation based on TTBR0, N is the value of TTBCR.N
For a translation based on TTBR1, N is 0
For details of *Properties* fields, see the register or descriptor description

1MB'lik Bölüm (Section) Kullanımı

ARM'daki bölüm tabanlı sanal bellek kullanımı Intel'in segment tabanlı sanal bellek kullanımına benzemektedir. Bu kullanımda yalnızca dizin tablosu kullanılır. 1MB bölümleme sisteminde sanal adres iki kısma ayrılmaktadır.



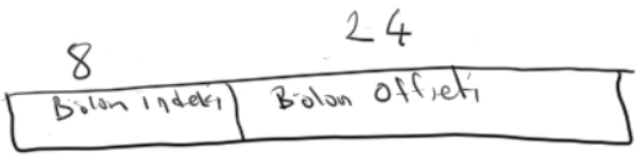
Dönüştürme tek aşamalıdır. TTBR0 ya da TTBR1 yazmacının gösterdiği yerden dizin tablosuna erişilir. Dizin tablosu elemanları dörder byte'tır. Bu elemanların 12 biti doğrudan bölümün fiziksel bellekteki bölüm indeksini belirtir. Bölüm belirten dizin elemanlarının formatı şöyledir:



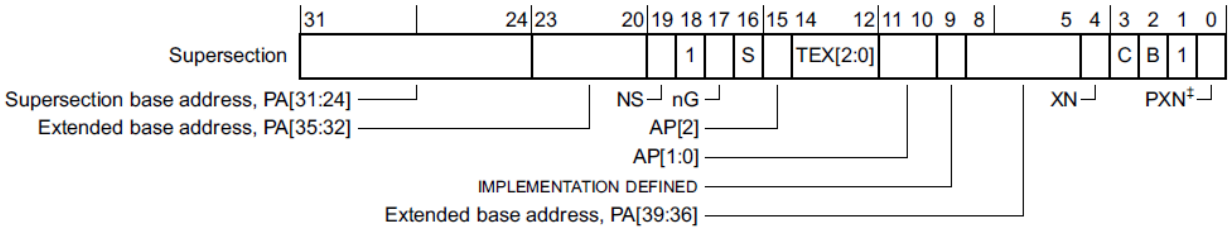
Dizin tablosundan çekilen dizin elemanı ile bölümün fiziksel adresi belirlendikten sonra sanal adresin düşük anlamlı 20 biti ile bu adres toplanarak gerçek fiziksel adrese erişilir. Dizin tablosunda bölüm girişleriyle normal sayfa tablosu girişlerinin beraber bulunabildiğine dikkat ediniz.

16 MB'lik Bölüm (Supeer Section) Kullanımı

Bu kullanımda sanal adres yine iki bölümden oluşmaktadır.



Yine dönüştürme yalnızca dizin tablosuna bakılarak tek aşamada yapılır. Dizin tablosu girişlerinde 16 MB'lik bölümün fiziksel bellekteki indeksini 8 bit vardır. Dizin tablosu girşinin genel formatı şöyledir:



Burada belirtilen 8 bitlik indeks değeri 16 MB ile çarpılarak fiziksel bellekte bölümün yeri belirlenir. Sonra bu değere böküm offseti toplanarak gerçek fiziksel adres elde edilir.

64 Bit Girişleri (Descriptors) Kullanan Sayfalama Mekanizması

64 bit girişleri (descriptors) kullanan sayfalama mekanizmasında dönüştürme üç aşamada yapılmaktadır. Bu girişler yalnızca 4KB'lik sayfaların kullanımına olanak sağlamaktadır.

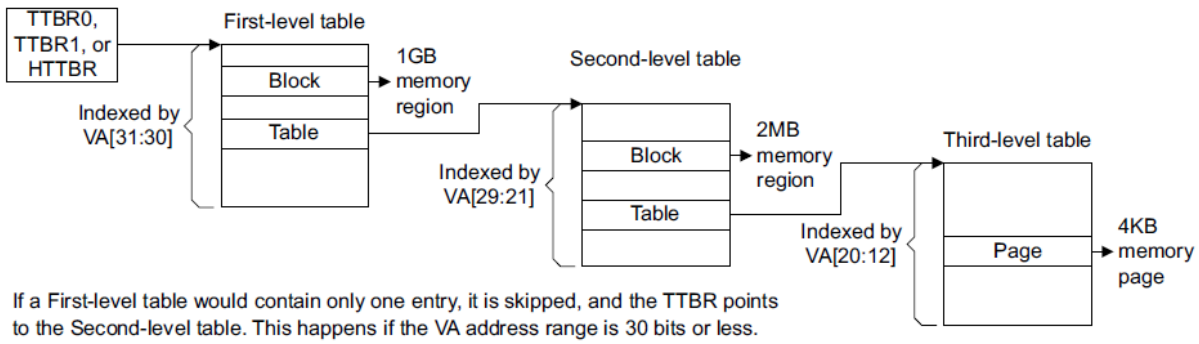
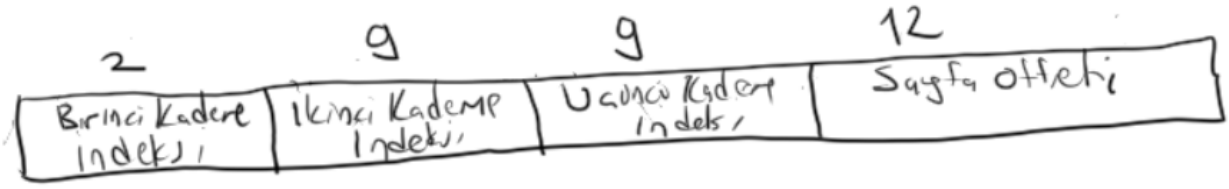


Figure B3-12 General view of stage 1 address translation using Long-descriptor format

Bu kullanımda sanal adres dört kısımdan oluşmaktadır:



Birinci Kademe Dönüştürme Tablosu (First Level Translation Table) toplam 4 girişe sahiptir. Birinci Kademe Dönüştürme Tablosu yine işlemci tarafından TTBR0 ya da TTBR1 yazmacının gösterdiği yerde aranmaktadır. Buradan çekilen 8 byte'lık girişte İkinci Kademe Dönüştürme Tablosunun Fiziksel Bellekteki indeksi bulunur. Bu girişin formatı şöyledir:

64 Bit ARM İşlemcileri

64 Bit ARM mimarisi ARM64 ya da AArch64 biçiminde isimlendirilmektedir. Bu mimarinin kullandığı komut kümesine ve yazmaç yapısına ARMV8 denilmektedir. ARM'ın 64 bite geçme öyküsü çok yenidir. Bu mimarinin ilk sürümü A53 cortex'i ile 2014'te yapılmıştır. Raspberry Pi 3 bu Cortex'i kullanmaktadır. ARM64 mimarisinin tipik özellikleri şunlardır:

- Daha geniş fiziksel adres alanı: ARM32 mimarisinde fiziksel adres özel olarak 40 bite çıkabiliyordu. ARM64 cortex'lerinde default olarak fiziksel adres 40 bit (1 TB) biçimdedir. Tabii ileride bu daha da artırılabilir.

- 64 Bit sanal adres alanı: ARM64 mimarisinde sanal adresler 64 bittir. Yani bir program teorik olarak 2^{64} = 16 EB uzunlukta olabilmektedir. Dolayısıyla ARM64 mimarisinde göstericiler de 8 byte uzunluğundadır.

- Artırılmış yazmaç sayısı: ARM64 mimarisinde toplam 32 genel amaçlı yazmaç bulunmaktadır. Yani ARM32 mimarisine göre yazmaç sayısı iki kat artmıştır. Tabii yazmaçlar da 64 bite yükseltilmiştir.

- ARM64 mimarisinde kodlar için 4 öncelik derecesi bulunmaktadır (EL0, EL1, EL2, EL3).

- ARM64 mimarisinde sanallaştırma için özel komutlar ve mekanizmalar bulundrulmuştur.

- ARM64 mimarisi geriye doğur ARM32 mimarisi (ARMV7) ile uyumludur. Yani ARM64 işlemcileri iki çalışma moduna sahiptir. 32 bit modda bunlar çok küçük farklılıklarla ARMV7 komut kümesini kullnarak normal 32 bit ARM işlemcileri gibi çalışmaktadır (Örneğin Raspberry Pi 3'teki Raspbian işlemciyi 32 bit modda çalıştırmaktadır). Ancak 32 bit modda 64 bit yazmaçlar ve 64 bit özellikler kullanılamamaktadır. Yine tıpkı Intel mimarisinde olduğu gibi 64 bit proseslerle 32 bit prosesler birarada ARM64 mimarisinde çalışabilmektedir. Şöyle ki: İşletim sistemi 64 bit prosesten 32 bit prosese geçiş yaparken işlemcinin modunu da değiştebilmekte böylece her iki uygulama birlikte çalışabilmektedir.

- ARM64 mimarisinde makine komutları yine 32 bittir (64 bit değildir.) Ancak ARMV7'den farklı bir encoding kullanılmıştır. Komutların genel formatında da önemli değişiklikler vardır. Genel olarak ARMV8 komutlarının daha tutarlı ve etkin kodlandığı düşünülmektedir.

- ARM64 mimarisinde genel olarak Thumb-2 komut kümesi kullanılabilmektedir.

ARM64 Yazmaçları

ARM64 mimarisinde yukarıda da belirtildiği gibi toplam 32 genel amaçlı yazmaç vardır ve bunların her biri 64 bittir. Yazmaçlar X önekiyle X0, X1, ..., X31 biçiminde isimlendirilmiştir. Bu genel amaçlık yazmaçların düşük anlamlı 32 bitlik kısmı W öneki ile W0, W1, ..., W31 biçiminde bağımsız olarak kullanılabilmektedir.

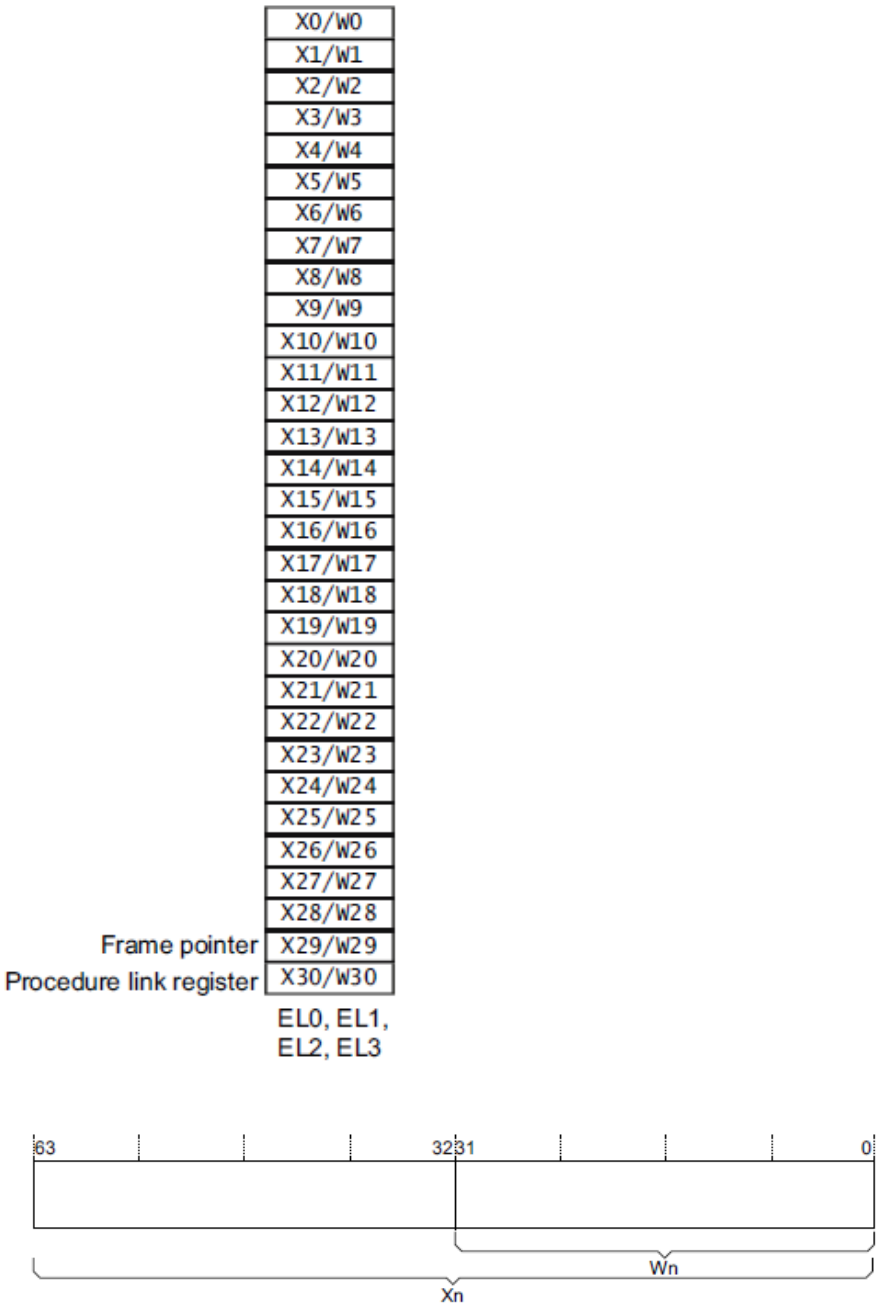


Figure 4-2 64-bit register with W and X access.

X yazmaçlarının düşük anlamlı W kısımları okunurken bunların yüksek anlamlı kısımları değişmez. Ancak bu W yazmaçlarına atama yapıldığında bunların içinde bulunduğu X yazmaçlarının yüksek anlamlı 32 bitlik kısmı sıfırlanmaktadır. X yazmaçarının yüksek anlamlı 32 bitlik kısımlarına ayrı bir biçimde erişme olanağı yoktur.

Anımsanacağı gibi 32 bit mimaride (ARMV7'de) load/store işlemleri dışındaki işlemler her zaman 32 bit genişliğinde yapılmaktadır. Ancak 64 bit mimaride 32 bit ya da 64 bit genişlikte işlem yapan komutlar bulunmaktadır. Yani örneğin biz iki X yazmacını ya da iki W yazmacını toplayabiliriz. Yukarıda da belirtildiği gibi komutların kodlanması büyük ölçüde değiştirilmiştir.

ARM64 mimarisinde PC yazmacına artık programlama yoluyla erişilememektedir. Gerçekten de X31 yazmacına erişilmek istendiğinde pek çok komutta artık bu yazmaç PC'yi değil Zero Yazmacı denilen bir yazmacı belirtmektedir. Zero yazmacı kavramı 32 bit mimaride yoktur. 64 bitlik zero yazmacı XZR ile, 32 bitlik zero yazmacı ise WZR ile gösterilmektedir. Bu yazmaçtan okuma yapıldığında her zaman sıfır okunur. Bu yazmaca yazma yapıldığında hiçbir şey olmaz. Yani yazılan değer ihmal edilir. Stack yazmacı SP ile gösterilir ve bazı komutlarda X31 biçiminde kodlanmaktadır.

ARM64 Mimarisinde C'de Fonksiyon Çağırma Biçimi (ARM64 Calling Convetion)

ARM64 için fonksiyon çağırma biçimi ARM'ın "Programmer's Guide for ARMv8-A dokümanın 9. Bölümünde ayrıntılı biçimde açıklanmıştır. Çağırma biçiminin özeti şöyledir:

- İlk 8 parametre X0-X7 yazmaçları yoluyla fonksiyona aktarılmaktadır.

- X9-X15 arası yazmaçları tamamen çağrılan fonksiyonun saklamak zorunda olmadığı yazmaçlardır. Bunlar çağrılan fonksiyon tarafından çeşitli işlemler için kullanılabilirler.

- X19-X29 arası yazmaçlar çağrılan fonksiyonun değerini saklamak zorunda olduğu yazmaçlardır. Yani başka bir deyişle çağıran fonksiyon bunların içerisindeki değerleri çağırma işleminden sonra aynı biçimde bulmayı ummaktadır.