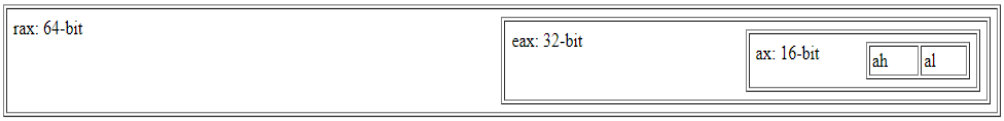
Assembly - 1

Assembly programlama dili günümüzde her ne kadar yazılım geliştirme sürecinde aktif olarak kullanılmasa da Binary Exploiting, Reverse Engineering, Code Optimization gibi birçok alanda aktif olarak kullanılmaktadır. Bu yazıda Assembly dilinin Syntax'ına aşina olmak ve bu dilde yazılan veya dönüştürülen kodları anlamlandırabilmek adına hazırlandı.

Başlangıç olarak sık karşılaşacağımız Register'larla başlayabiliriz. Register'lar Segment Registers ve General Purpose Registers olmak üzere ikiye ayrılır. Genel amaçlı kullanılan Register'ları ise kendi içerisinde Data Registers, Pointer Registers ve Index Registers olarak üçe ayırabiliriz. Bu Register'lara bakıldığında;

1- Data Registers

- **RAX**, Accumulator Register olarak adlandırılır. Aritmetik, mantık ve veri aktarımı gibi çeşitli işlemlerde kullanılır.
    - 64 bit mimaride -> RAX, 32 bit mimaride -> EAX, 16 bit mimaride -> AX (AH ve AL olmak üzere iki parçadan oluşur), 8 bir mimaride -> AL sembolüyle temsil edilmektedir.



- **RBX**, Base Register olarak adlandırılır. Genellikle dolaylı adresleme tabanlı, endeksli veya Register tabanlı bir veri işaretçisi içerir.
    - 64 bit mimaride -> RBX, 32 bit mimaride -> EBX, 16 bit mimaride -> BX (BH ve BL olmak üzere iki parçadan oluşur), 8 bir mimaride -> BL sembolüyle temsil edilmektedir.

- **RCX**, Counter Register olarak adlandırılır. Döngüler için bir sayaç görevi görür. Kaydırma/döndürme talimatları ve dize manipülasyonunun her ikisi de sayım yazmacının bir sayaç olarak kullanılmasına izin verir.
    - 64 bit mimaride -> RCX, 32 bit mimaride -> ECX, 16 bit mimaride -> CX (CH ve CL olmak üzere iki parçadan oluşur), 8 bir mimaride -> CL sembolüyle temsil edilmektedir.

- **RDX**, Data Register olarak adlandırılır. I/O işlemlerinde veri kaydı port numarası olarak kullanılabilir. Aynı zamanda bölme ve çarpma işlemlerine de kullanılır.
    - 64 bit mimaride -> RDX, 32 bit mimaride -> EDX, 16 bit mimaride -> DX (DH ve DL olmak üzere iki parçadan oluşur), 8 bir mimaride -> DL sembolüyle temsil edilmektedir.

2- Index Registers

- **RSI**, Source Index Register olarak adlandırılır. DS'nin adreslediği veri segmentindeki hafıza adreslerini tanımlamak için kullanılır. Bu nedenle SI içeriğini arttırdığımızda ardışık hafıza konumlarına erişmek kolaydır.

- 64 bit mimaride -> RSI, 32 bit mimaride -> ESI, 16 bit mimaride -> SI (SH ve SL olmak üzere iki parçadan oluşur), 8 bir mimaride -> SL sembolüyle temsil edilmektedir.

- **RDI**, Designated Register olarak adlandırılır. RSI Register'ı ile aynı amaçla kullanlır.
  - 64 bit mimaride -> RDI, 32 bit mimaride -> EDI, 16 bit mimaride -> DI (DH ve DL olmak üzere iki parçadan oluşur), 8 bir mimaride -> DL sembolüyle temsil edilmektedir.

3- Pointer Registers

- **RBP**, Base Pointer olarak adlandırılır. Program çalıştırıldığında programın STACK alanındaki verilerinin nereden itibaren depolandığını işaretlemek için kullanılır. Yani bir anlamda Stack yapısında tabanının adresini tutmaktadır. Program akışında Stack yapısına veri depolandıkça RSP Register değeri değişikliğe uğradığı için genelde Stack üzerinde aralarda kalan değerlere ulaşabilmek için RBP Register referans alınarak erişim sağlanır.
  - 64 bit mimaride -> RBP, 32 bit mimaride -> EBP, 16 bit mimaride -> BP (SBA ve SBL olmak üzere iki parçadan oluşur), 8 bir mimaride -> SBL sembolüyle temsil edilmektedir.

- **RSP**, Stack Pointer olarak adlandırılır. Program yığınını işaretlemek için kullanılan Stack Pointer olarak bilinir. Stack yapısının tepesindeki ögeyi belirtir. Stack yapısına henüz veri depolanmamışsa SP (16 bit) değeri 0xFFFE adresini gösterecektir.
  - 64 bit mimaride -> RSP, 32 bit mimaride -> ESP, 16 bit mimaride -> SP (SPA ve SPL olmak üzere iki parçadan oluşur), 8 bir mimaride -> SPL sembolüyle temsil edilmektedir.

64 bit mimaride General Purpose Register ailesine ek olarak 8 Register daha eklenmiştir. Bunlar;

- R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D

Genel amaçlı Register'lar hakkında fikir sahibi olunduktan sonra program içerisinde komutların çalıştırılmasıyla (aritmetik ve mantıksal işlemler sonucunda) değişikliğe uğrayan **EFLAGS** Registers üzerindeki bitlerin anlamlarının bilinmesi gerekiyor. EFLAGS Registers'ı üzerindeki bitler kullanım amaçlarına göre Status Flag, Control Flag ve System Flag olmak üzere üç farklı kategori altında incelenebilir. Bu bitlere bakıldığında;

1- Status Flags

- **CF** (Carrier Flags), işlem sonucunda Register'lar üzerinde bir taşma meydana geldiğinde (işlem sonucu kullanılan mimarideki bit miktarını aştığında) set edilen bittir.
  - STC, CLC veya CMC komutlarıyla set edilebilir veya sıfırlanabilir.
- **PF** (Parity Flags), işlem sonucunun en anlamsız 8 bitinin (örneğin sonuç EAX üzerinde tutuluyorsa AL üzerinde) bir çift sayı içermesi durumunda set edilen bittir.
- **AF** (Auxiliart Carry Flags)
- **ZF** (Zero Flags)
- **SF** (Sign Flags)
- **OF** (Overflow Flag)

2- Control Flags

- **DF** (Direction Flag)

3- **System Flags**
    - **TF** (Trap Flag)
    - **IF** (Interrupt Enable Flag)
    - **IOPL** (I/O Priviledge Level)
    - **NT** (Nested Task Flag)
    - **RF** (Resume Flag)
    - **VM** (Virtual Memory)

Temelde sık kullanılan birkaç temel komutun anlamlarına ve kullanım şekillerine bakıldığında;

- **MOV**, hedef değişkende tutulan değeri kaynak değişkene taşımak için kullanılan komuttur.
    - Bu komut 1 Clock Cycles harcar.
    - MOV EAX, EBX //EBX Register içerisindeki değer EAX Register'ına taşınıyor.

- **ADD**, kaynak değişkende tutulan değerin üzerine hedef değişkende tutulan değeri eklemek için kullanılan komuttur.
    - ADD EAX, EBX // EAX = EAX + EBX ifadesine karşılık gelir.
    - Bu komut 1 ila 3 Clock Cycles arası harcar.
    - İşlem sonucuna göre CF, ZF, SF, OF, PF, AF bitleri set edilebilir.

- **SUB**, kaynak değişkende tutulan değerden hedef değişkende tutulan değeri çıkarmak için kullanılan komuttur.
    - SUB EAX, EBX // EAX = EAX - EBX ifadesine karşılık gelir.
    - Bu komut 1 ila 3 Clock Cycles arası harcar.
    - İşlem sonucuna göre "ADD" komutunda olduğu gibi CF, ZF, SF, OF, PF, AF bitleri set edilir.

- **MUL**, EAX Register'ı içerisinde bulunan değer ile "MUL" komutunda verilen Register içerisinde bulunan değeri çarpmak için kullanılan komuttur. Çarpma işleminin sonucu EAX Register'ında depolanır. Bu nedenle çarpma işlemine başlamadan önce çarpılmak istenen değer EAX Register'ına taşınır. Çarpan değerini taşıyan değişken ise "MUL" komutunda kullanılır.
    - MOV EAX, 12345h
      MOV EBX, 100h
      MUL EBX // EAX = 1234500h, CF = 0
    - İşlem sonucuna göre CF ve OF bitleri set edilir.
    - Çarpma işleminde kullanılan değerler kullanılan Register boyutlarında büyük çıkması durumunda (CF biti 1 set ediliyor) çarpma işleminin sonucu EDX ve EAX üzerine iki parça halinde depolanır. Örnek olarak;
      MOV EAX, 123675h
      MOV ECX, 1000h
      MUL ECX //EDX = 00000012h, EAX = 87650000h, CF = 1
    - Bu komut kullanılan bit mimariye göre 8 ve 16 bit mimaride -> 11, 32 bit mimaride -> 10 Clock Cycles harcar.

- “MUL” komutu işaretsiz Integer değerler üzerinde işlemler yaparken kullanılmaktadır. İşaretli Integer değerler üzerinde işlemler yapılırken “IMUL” komutu kullanılıyor. Örnek olarak;

```
MOV EAX, 4823424
MOV EBX, -423
IMUL EBX        //EAX = FFFFFFFF86635D80h
```

- **DIV**, EAX Register’ı içerisinde bulunan değeri “DIV” komutunda verilen Register içerisindeki değere bölmek için kullanılan komuttur. İşlem sonucunda bölüm değeri EAX Register’ında depolanır. Kalan değer ise EDX Register’ı içerisinde depolanır. Bu nedenle bölme işlemine başlamadan önce EDX Register’ını sıfırlamakta fayda vardır.
  -
```
MOV EDX, 0
MOV EAX, 8003h
MOV CX, 100h
DIV CX                  //EAX = 00000080h, EDX = 3
```
  - İşlem sonucuna göre herhangi bir bayrak biti set edilmez.
  - Bu komut kullanılan bit mimariye göre 8 bit mimaride -> 17, 16 bit mimaride -> 25, 32 bir mimaride -> 41 Clock Cycles harcar.
  - “MUL” komutunda olduğu gibi “DIV” koutu da işaretsiz Integer sayılar üzerinde işle yapmak için kullanılıyor. İşaretli sayılar üzerinde işlem yapabilmek için “IDIV” komutu kullanılıyor.
    - Bu komut kullanılan bit mimariye göre 8 bit mimaride -> 22, 16 bit mimaride -> 30, 32 bir mimaride -> 46 Clock Cycles harcar.

- **INC**, verilen değişkenin içeriğindeki değeri bir arttırmak için kullanılıyor.
  -
```
INC EAX         //EAX ++; ifadesine karşılık geliyor.
```
  - Bu komut 1 Clock Cycles harcar.
  - İşlem sonucuna göre CF, ZF, SF, OF, PF, AF bitleri set edilebilir.

- **DEC**, verilen değişkenin içeriğindeki değeri bir azaltmak için kullanılıyor.
  -
```
DEC EAX         //EAX –; ifadesine karşılık geliyor.
```
  - Bu komut 1 Clock Cycles harcar.
  - İşlem sonucuna göre CF, ZF, SF, OF, PF, AF bitleri set edilebilir.

- **CMP**, kendisine verilen iki değeri birbirinden çıkararak karşılaştırma işlemi yapan komuttur. Karşılaştırma sonucuna göre ZF (Zero Flag) ve CF (Carrier Flag) bitleri set edilir. Bu bitler kullanılarak karar karar mekanizmaları oluşturulur (iki değer de eşit ise ZF = 1 set edilir, kaynak değişkenin değeri hedef değişkenden büyükse sonuç negatif çıkacağı için CF biti 1 set edilir. Aksi şekilde kaynak değişkenin değeri hedef değişkenin değerinden büyükse sonu. Pozitif çıkacağı için CF biti 0 set edilecektir).
  -
```
CMP EAX, 1      //EAX ==1 ise ZF biti 1 set edilir.
```
  - Bu komut 1 Clock Cycles arası harcar.
  - İşlem sonucuna göre CF, ZF, SF, OF, PF, AF bitleri set edilir.

- **TEST**, verilen değişkenleri bit bit (Bitwise) “AND” işlemine tabi tutar (Yani AND operatörü ile aynı görevi görüyor). AND operatöründen tek farkı işlem sonucunu kaynak değişkene kaydetmek yerine işlem sonucuna göre ZF (Zero Flag), SF (Sign Flag) ve PF (Parity Flag)

bitlerini set etmesidir. İşlem sonucuna göre set edilen bayrak bitlerine göre kontrol mekanizmaları oluşturulur. TEST işlemi sonucuna göre;

- Src & Dst > 0 ise SF = 0, ZF = 0
  Src & Dst == 0 ise SF = 0, ZF = 1
  Src & Dst < 0 ise SF = 1, ZF = 0
- MOV EAX, 00000101b
  TEST EAX, 1 //AND işleminin sonucu > 0 olacağı için ZF = 0
  TEST EAX, 10b // (10b = 2) AND işleminin sonucu = 0 olacağı için ZF = 1
- Bu komut 1 Clock Cycles harcar (hafızada tutulan bir değişken üzerinde işlem yapılıyorsa 2 Clock Cycle harcar).
- İşlem sonucuna göre ZF, AF ve PF bitleri set edilir.

- **JMP**, verilen etiket adresine koşulsuz dallanmak için kullanılan komuttur.
  - START:
    JMP START //START etiketinin bulunduğu adrese dallanır.
  - Bu komut doğrudan dallanmalarda 1, dolaylı dallanmalarda 2 Clock Cycles harcar. Çeşitlerine göre harcanan Clock Cycle değeri de değişiklik gösterir.
  - Karşılaştırılma **komutlarıyla** birlikte kullanıldığında koşullu dallanma komutu olarak da kullanılabilmektedir. Karşılaştırma komutlarıyla kullanılan versiyonlarının birkaçına bakıldığında;
    - JE (Jump Equal), öncesinde kullanılan herhangi bir karşılaştırma işlemi sonucunda iki değişkenin değeri de birbirine eşitse (**ZF = 1 ise**) komutla verilen etiket adresine dallanılır. Aksi taktirde program bir alt satırdan itibaren çalıştırılmaya devam edilir.
    - JNE (Jump Not Equal), öncesinde kullanılan herhangi bir karşılaştırma sonucunda iki değişkenin değeri de birbirine eşit değilse (**ZF = 0 ise**) komutla verilen etiket adresine dallanılır. Aksi taktirde program bir alt satırdan itibaren çalıştırılmaya devam edilir.
    - JZ (Jump Zero) ZF bitini set edebilen herhangi bir işlem sonrasında **ZF biti 1 set edilmişse (yani sonucu 0 çıkmışsa**) komutla verilen etiket adresine dallanılır. Aksi takdirde bir alt satırdan itibaren program çalıştırılmaya devam eder.
    - JNZ (Jump Not Zero) ZF bitini set edebilen herhangi bir işlem sonrasında **ZF biti 0 set edilmişse** komutla verilen etiket adresine dallanılır. Aksi takdirde bir alt satırdan itibaren program çalıştırılmaya devam eder.
    - JS (Jump Sign), Sign bayrak biti 1 (negatif) set edilmişse (**SF = 1 ise**) komutla verilen etiket adresine dallanılır. Aksi takdirde program bir alt satırdan itibaren çalışmaya devam eder.
    - JNS (Jump Not Sign), Sign bayrak biti 0 (pozitif) set edilmişse (**SF = 0 ise**) komutla verilen etiket adresine dallanılır. Aksi takdirde program bir alt satırdan itibaren çalışmaya devam eder.
    - JG (Jump Greater), öncesinde kullanılan herhangi bir karşılaştırma işlemi sonucunda birinci değişken değeri ikinci değişkenden büyük olması durumunda (**ZF = 0 ve SF = 0 ise**) komutla verilen etiket adresine dallanılır. Aksi takdirde program bir alt satırdan itibaren çalıştırılmaya devam edilir.
    - JGE (Jump Greater or Equal), JG komutu gibi çalışır. Farklı olarak karşılaştırılan iki değişken değerinin eşit olması durumunda (**OF = SF ise**) da komutla verilen etiket adresine dallanılır.

        - JL (Jump Less), öncesinde kullanılan herhangi bir karşılaştırma işlemi sonucunda birinci değişken değeri ikinci değişkenden küçük olması durumunda (**SF != OF ise**) komutla verilen etiket adresine dallanılır. Aksi takdirde program bir alt satırdan itibaren çalıştırılmaya devam edilir.
        - JLE (Jump Less or Equal), JL komutu gibi çalışır. Farklı olarak karşılaştırılan iki değişken değerinin eşit olması durumunda (**SF != OF veya ZF = 1 ise**) da komutla verilen etiket adresine dallanılır.
        - JA (Jump Above), öncesinde kullanılan herhangi bir karşılaştırma işlemi sonucunda (**işaretsiz**) birinci değişken değeri ikinci değişken değerinden büyükse (**CF = 0 ve ZF = 0 ise**) komutla verilen etiket adresine dallanılır.
        - JB (Jump Below), öncesinde kullanılan herhangi bir karşılaştırma işlemi sonucunda (**işaretsiz**) birinci değişken değeri ikinci değişken değerinden küçükse (**CF = 1 ise**) komutla verilen etiket adresine dallanılır.
    - Yukarıdakilere benzer daha birçok Jumo dallanma komutu versiyonu bulunmaktadır. Dallanma komutlarının tamamında dallanma işlemi sonrasında herhangi bir bayrak biti set edilmiyor.

- **CALL**, bit prosedür çağırır/bir etiket adresine dallanılır. Dallanma öncesinde dönüş adresini (dallanılan adresten bir sonraki çalışacak satırın adresi) Stack'e kaydedilir (PUSH). Dallanma adreste prosedür sonlandığında ("RET" komutu kullanılır) Stack'e kaydedilen adres çıkarılarak (POP) dallanmadan önce kaydedilen adrese geri dönülür. Program dallanmadan önce çalıştırılacak adresten itibaren kaldığı yerden çalışmaya devam eder.
    - LABEL1:
      RET
    - CALL LABEL1 //LABEL1 etiketinin bulunduğu adrese dallanılır.
    - Dallanılan etiket adresinden/prosedürden dönüşte bir değer döndürülmek isteniyorsa EAX Register'a depolanarak değer ana akışa taşınabilir.
    - Dallanılan prosedürün/etiket adresinin sonunda dallanmadan önceki adrese geri dönebilmek için "RET" komutu kullanılmak zorundadır.
    - İşlem sonucunda herhangi bir bayrak biti set edilmez.

- **LOOP**, ECX Register'ı üzerinden kontrol sağlayarak döngü yapısı oluşturmak için kullanılan komuttur (Kullanılmadan önce ECX Register'ını set etmek gerektiğini gösterir). Her çalışmasında ECX Register'ını bir azaltıp sıfıra eşit olup olmadığını kontrol eder. Eğer ki sıfır olmamışsa komutla verilen etiket adresine dallanılır.
    - MOV ECX, 5
      LABEL1:
      PRINT "loop"
      LOOP LABEL1 //5 defa "loop" çıktısı elde edilir.
    - Bu komut 5 veya 6 Clock Cycles harcar.
    - İşlem sonucunda herhangi bir bayrak biti set edilmez.

- **RET**, EIP'i (Extended Instruction Pointer) Stack üzerinde bulunan dönüş adresine set ederek programın dallanmadan önce kaydedilen adresten itibaren çalışmasını sağlayan komuttur.
    - RET
    - RET <Address>

- **AND**, verilen değişkenler arasında karşılıklı tüm bitleri mantıksal VE işlemine tabi tutarak sonucu kaynak değişkende depolayan komuttur. Bitler karşılıklı olarak 1 olmadığı sürece sonuç her zaman 0'dır.
    - 1 AND 1 = 1
      1 AND 0 = 0
      0 AND 1 = 0
      0 AND 0 = 0
    - MOV EAX, 11010111b
      AND EAX , 00011011b //EAX = 00010011
    - Bu komut 1 Clock Cycles arası harcar.
    - İşlem sonucuna göre ZF, SF ve PF bitleri set edilir.

- **OR**, verilen değişkenler arasında karşılıklı tüm bitleri mantıksal VEYA işlemine tabi tutarak sonucu kaynak değişkende depolayan komuttur. Bitler karşılıklı olarak 0 olmadığı sürece sonuç her zaman 1'dir.
    - 1 OR 1 = 1
      1 OR 0 = 1
      0 OR 1 = 1
      0 OR 0 = 0
    - MOV EAX, 11010111b
      OR EAX, 00011011b //EAX = 11011111b
    - Bu komut 1 Clock Cycles arası harcar (değişken üzerinde/bellek üzerinden okunması gerekiyorsa 3 Clock Cycle harcanır).
    - İşlem sonucuna göre ZF, SF ve PF bitleri set edilir.

- **XOR**, verilen değişkenler arasında karşılıklı tüm bitleri mantıksal XOR işlemine tabi tutarak sonucu kaynak değişkende depolayan komuttur.
    - 1 XOR 1 = 1
      1 XOR 0 = 0
      0 XOR 1 = 0
      0 XOR 0 = 1
    - MOV EAX, 11010111b
      XOR EAX, 00011011b //EAX = 00110011b
    - Bu komut 1 ila 3 Clock Cycles arasında harcar.
    - İşlem sonucuna göre ZF, SF ve PF bitleri set edilir.

- **NOP**, bekleme komutudur. Herhangi bir işlem yapmaz.
    - NOP //Bir kez bekler.
    - Bu komut 1 Clock Cycle harcar. Gecikmeler oluşturulurken kullanılabiliyor.
    - İşlem sonucunda herhangi bir bayrak biti set edilmez.

- **SHL**, kaynak değişken içerisindeki değeri ikinci kısımda verilen miktar kadar sola kaydırmak için kullanılan komuttur.
    - MOV EAX, 01110000b
      SHL EAX, 1 //EAX = 11100000b
    - Bir değer 2'nin katıyla çarpılmak istendiğinde de SHL komutu kullanılabiliyor.
    - Bu komut 1 Clock Cycle harcar (Hafıza alanındaki bir değişken üzerinde işlem yapılıyorsa 3 Clock Cycle harcar).

    - İşlem sonucuna göre CF ve OF bitleri set edilebilir.

- **SHR**, kaynak değişken içerisindeki değeri ikinci kısımda verilen miktar kadar sağa kaydırmak için kullanılan komuttur.
    - MOV EAX, 01110000b
      SHR EAX, 1 //EAX = 00111000b
    - Bir değer 2'nin katına bölünmek istendiğinde de SHR komutu kullanılabiliyor.
    - Bu komut 1 Clock Cycle harcar (Hafıza alanındaki bir değişken üzerinde işlem yapılıyorsa 3 Clock Cycle harcar).
    - İşlem sonucuna göre CF ve OF bitleri set edilebilir.
-
- **PUSH**, değişken içerisindeki değerin Stack alanına depolanması için kullanılan komuttur.
    - MOV EAX, 1234h
      PUSH EAX
    - Bu komut 1 Clock Cycle harcar (Bellekte bulunan bir değişken üzerinde işlem yapılıyorsa 2 Clock Cycle harcar).
    - İşlem sonucunda herhangi bir bayrak biti set edilmez.

- **POP**, Stack alanının tepesinde bulunan değeri komutla kullanılan değişkene taşımak için kullanılan komuttur.
    - MOV EAX, 1234h
      PUSH EAX
      POP EDX //EDX = 1234h
    - Bu komut genel amaçlı bir Register üzerine yapılıyorsa 1 Clock Cycle harcar (Bellekte bulunan bir değişken veya Segment Register üzerinde işlem yapılıyorsa 3 Clock Cycle harcar).
    - İşlem sonucunda herhangi bir bayrak biti set edilmez.

- **ENTER**,
- **LEA**,
- **INT**,
- IN,
- **OUT**,
- **LES**,

Kaynaklar

- https[:]//teknoltan.com/assembly-komutlari-ve-anlamlari/
- https[:]//www.tortall.net/projects/yasm/manual/htmlwww.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html/arch-x86-registers.html
- https[:]//www.csie.ntu.edu.tw/~acpang/course/asm_2004/slides/chapt_07_PartIISolve.pdf
- https[:]//senakaraduman.github.io/2019-04-11-x86-Assembly-Komutlar%C4%B1/
- https[:]//www.youtube.com/watch?v=FGlSBeW52HA
- https[:]//www.allaboutcircuits.com/technical-articles/how-to-write-assembly-basic-assembly-instructions-ARM-instruction-set/
- https[:]//courses.cs.washington.edu/courses/cse410/17wi/lectures/CSE410-L10-asm-III_17wi_ink.pdf
- https[:]//aakgul.sakarya.edu.tr/sites/aakgul.sakarya.edu.tr/file/_8086KomutlarOrnekler.PDF
    - Komutlar açıklayıcı, detaylı ve düzenli
- https[:]//en.wikipedia.org/wiki/FLAGS_register
- https[:]//paws.kettering.edu/~jhuggins/humor/opcodes.html
- https[:]//www.cin.ufpe.br/~clac/infra_de_software/Introduction%20to%20Assembly%20Language%20Programming~tqw~_darksiderg.pdf
    - Detaylı kaynak
- https[:]www.eecg.utoronto.ca/~amza/www.mindsec.com/files/x86regs.html
- https[:]//www.geeksforgeeks.org/general-purpose-registers/