İçindekiler

**[DONANIM (HARDWARE)](#_Toc377922190)** [3](#_Toc377922190)

[**SORU-1: ID3 Tekrarlı İkilikçi Ağaç, Iterative Dichotomiser 3 hakkında bilgi veriniz.** 4](#_Toc377922191)

[**SORU-2: Kosinüs Benzerliği (Cosine Similarity) hakkında bilgi veriniz.** 8](#_Toc377922192)

[**SORU-3: Cache (Önbellek) Teknolojilerinin Okuma Hızları Yönünden Karşılaştırılması hakkında bilgi veriniz.** 10](#_Toc377922193)

[**SORU-4: Son i7 ailesi, vPro AMT ve SCS teknolojileri hakkında bilgi veriniz.** 16](#_Toc377922194)

[**SORU-5: Encoder (Kodlayıcı) hakkında bilgi veriniz.** 16](#_Toc377922195)

[**SORU-6: Banker Algoritması (Banker's Algorithm) hakkında bilgi veriniz.** 19](#_Toc377922196)

[**SORU-7: Ondalıklı sayıların taban dönüşümleri hakkında bilgi veriniz.** 22](#_Toc377922197)

[**SORU-8: Mealy ve Moore Makineleri (Mealy and Moore Machines) hakkında bilgi veriniz.** 24](#_Toc377922198)

[**SORU-9: Translation Lookaside Buffer (TLB, Dönüşüm Hafızası) hakkında bilgi veriniz.** 27](#_Toc377922199)

[**SORU-10: Çok Çekirdekli İşlemciler (Multi-Core Processors) hakkında bilgi veriniz.** 30](#_Toc377922200)

[**SORU-11: Flynn Sınıflanırması (Flynn’s Taxonomy) hakkında bilgi veriniz.** 31](#_Toc377922201)

[**SORU-12: Von Neumann Makinesi hakkında bilgi veriniz.** 35](#_Toc377922202)

[**SORU-13: Ön Hafıza (Cache) hakkında bilgi veriniz.** 36](#_Toc377922203)

[**SORU-14: Turing Makinesi (Turing Machine) hakkında bilgi veriniz.** 51](#_Toc377922204)

[**SORU-15: Atomluluk (Atomicity) hakkında bilgi veriniz.** 58](#_Toc377922205)

[**SORU-16: Yükleyici (Loader) hakkında bilgi veriniz.** 59](#_Toc377922206)

[**SORU-17: Konumlandırılabilir Kod (Relocatable Code) hakkında bilgi veriniz.** 60](#_Toc377922207)

[**SORU-18: 2 geçişli çeviriciler (2 pass assemblers) hakkında bilgi veriniz.** 60](#_Toc377922208)

[**SORU-19: Kayan Nokta Sayıları (Floating Point Numbers) hakkında bilgi veriniz.** 62](#_Toc377922209)

[**SORU-20: Borulama (Pipelining) hakkında bilgi veriniz.** 65](#_Toc377922210)

[**SORU-21: Çevirici (Assembler) hakkında bilgi veriniz.** 66](#_Toc377922211)

[**SORU-22: Rastgele Erişilebilir Bellek (Random Access Memory , RAM) hakkında bilgi veriniz.** 67](#_Toc377922212)

[**SORU-23: Sıralama Algoritmaları (Sorting Algorithms) hakkında bilgi veriniz.** 68](#_Toc377922213)

[**SORU-24: Yahut (Özel Veya (exclusive or, farklılık operatörü)) hakkında bilgi veriniz.** 72](#_Toc377922214)

[**SORU-25: CRC (cyclic redundancy check, çevrimsel fazlalık sınaması) hakkında bilgi veriniz.** 72](#_Toc377922215)

[**SORU-26: Sayıcı (Counter) hakkında bilgi veriniz.** 73](#_Toc377922216)

[**SORU-27: flip flop (flipflop) hakkında bilgi veriniz.** 75](#_Toc377922217)

[**SORU-28: Salt okunur bellek (read only memory , ROM) hakkında bilgi veriniz.** 76](#_Toc377922218)

[**SORU-29: çoklayıcı (multiplexer) hakkında bilgi veriniz.** 76](#_Toc377922219)

[**SORU-30: kod çözücü (decoder) hakkında bilgi veriniz.** 79](#_Toc377922220)

[**SORU-31: tam toplayıcı (full adder) hakkında bilgi veriniz.** 82](#_Toc377922221)

[**SORU-32: Bir tümleyeni hakkında bilgi veriniz.** 84](#_Toc377922222)

[**SORU-33: İki tümleyeni hakkında bilgi veriniz.** 85](#_Toc377922223)

[**SORU-34: bit (ikil) hakkında bilgi veriniz.** 86](#_Toc377922224)

[**SORU-35: Disk Yönetimi (Disk Management) hakkında bilgi veriniz.** 87](#_Toc377922225)

# **DONANIM (HARDWARE)**

## **SORU-1: ID3 Tekrarlı İkilikçi Ağaç, Iterative Dichotomiser 3 hakkında bilgi veriniz.**

Bu yazının amacı, özellikle veri madenciliği (Data Mining) konularında sıkça kullanılan ve bir karar ağacı (decision tree) çeşidi olan ID3 ağacının nasıl çalıştığını açıklamaktır.

Klasik karar ağaçlarının iyileştirimesi olarak görülebilecek olan ID3, İngilizcedeki Iterative Dichotomiser 3 kelimelerinin baş harflerinden oluşmaktadır ve Türkçeye, tekrarlı ikililikçi ağacı kelimeleri ile çevrilebilir (Buradaki 3 sayısı ingilizcedeki ağaç kelimesine yakın olduğu için kullanılmıştır).

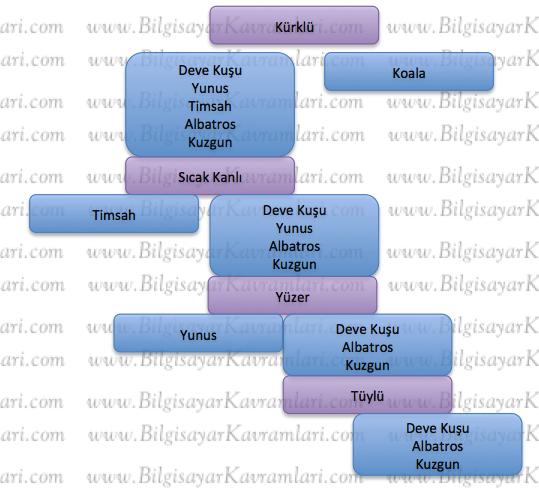
Ağacın, isminden de anlaşılacağı üzere amacı sürekli olarak ağaçtaki verileri mümkün olan en büyük iki parçaya bölmek ve böylece ağacın derinliğini azaltmaktır.

**Karar Ağacına Bir Örnek**

Bu konuyu anlatmadan önce bir karar ağacında (decision tree) aynı bilginin nasıl farklı iki şekilde tutulacağını göstermeye çalışalım. Örnek olarak aşağıdaki veri kümesini (data set) kullanmak isteyelim:



Yukarıda verilen hayvanların çeşitli özelliklerine göre aşağıdaki şekilde bir karar ağacı hazırlamak mümkündür:



Yukarıdaki ağaçta, karar ayırmlarını mor, ve karardan sonra evet olanları sağda hayır olanları ise solda göstermeye çalıştım. Buna göre ilk karardan sonra tek bir veri ayrılırken sonraki her kararda birer eleman ayrılıyor ve nihayet son karar olan Tüylü kararında hiç ayrılma olmuyor. Dolayısıyla bu sıra ile bakıldığında aslında Tüylü ayrımının bir önemi yoktur. Yani aslında 3 seviyede yapılabilecek bütün ayrımlar yapılmıştır denilebilir.

Yukarıdaki ayrımları, daha iyi anlaşılması için aşağıda uzunca yazacağım:

Koala, Kürklüdür

Timsah, Kürksüz ve Sıcak Kanlı değildir

Yunus, Kürksüzdür, Sıcak Kanlıdır ve yüzer

Deve Kuşu, Kürksüzdür, Sıcak Kanlı değildir ve yüzmez

Albatros, Kürksüzdür, Sıcak Kanlı değildir ve yüzmez

Kuzgun, Kürksüzdür, Sıcak Kanlı değildir ve Yüzmez.

Yukarıda, her elemana giden yolları uzunca yazdık. Buna göre herhangi bir elemanı ağaçta kodlayan kaçar adım olduğunu listeleyelim:

Koala -> 1 (sadece kürklüdür ayrımı yeterli)

Timsah -> 2 (hem kürksüz hem de sıcak kanlı olduğunu bilmemiz gerekiyor)

Yunus -> 3 ( hem kürksüz hem sıcak kanlı hem de yüzdüğünü bilmemiz gerekiyor)

Deve Kuşu -> 3

Albatros -> 3

Kuzgun -> 3

Yukarıdaki değerleri toplarsak, 15 adımda bütün elemanları birer kere ziyaret etmek mümkündür denilebilir.

Şimdi aynı verilerden, aşağıdaki ağacı üretelim:



Yukarıdaki yeni ağaç, aynı bilgileri tutmakta sadece karar ağacının karar adımlarının yeri değiştirilmiş. Bu durumda erişim sayılarını yazarsak:

Deve Kuşu -> 1

Albatros -> 1

Kuzgun -> 1

Koala -> 2

Timsah -> 3

Yunus -> 3

Yukarıdaki yeni ağaçta her elemana birer kere erişilmesi istendiğinde toplam 11 adım harcanması gerekmektedir. Bu durumda birinci ağaca göre daha iyi sonuç elde edileceği söyelenebilir.

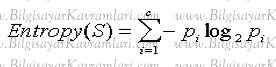
ID3 algoritması da tam bu noktada devreye girer ve Shannon’un bilgi teorisine (information theory) dayalı olarak entropi(dağınım) hesabı yapar ve ağacı bu yaptığı hesaba göre dağıtmaya çalışır.

**Algoritma**

ID3 algorimasını burada adım adım verip çalışmasını anlatmaya çalışalım:

1. Şu ana kadar ağaca dahil edilmemiş olan özellikleri alıp bu özelliklerin entropi (dağınım) değerlerini hesapla.
2. En düşük entropi değerine sahip olan özelliği seç.
3. Bu özelliği ayıran kararı ağaca ekle.

Buradaki entropi (dağınım) formülünü hatırlatmak istiyorum:



Yani bütün verilerden sadece o anda incelenen veri değeri çıkarılarak bu veri değerinin ikilik tabandaki logaritması ile çarpılır.

Bir örnek yaparak hatırlayalım. Örneğin yukarıdaki Evet / Hayır durumları için 2 Evet ve 4 Hayır olan bir durumu inceleyelim ve entropi değerini hesaplayalım:

Entropi (4E,2H) = – [(4/6) log2 (4/6)] – [(2/6) log2(2/6)] = – [0.666667 x –0.584962] – [0.3333334 x -1.56496 ] = – [-0.389975] – [ -0.528320833 ] = 0.918295

olarak bulunur.

Buradaki hesap 4 evet ve 2 hayır olması durumu için örnek olarak hazırlanmıştır. Örnek olarak aldığımız veri kümesindeki diğer kolonlar için de hesaplama yapılabilir. Ayrıca karar ağacında bazı kararlardan sonra kümedeki eleman sayısının azalacağı unutulmamalıdır.

Her adımda entropi hesabı yapıldıktan sonra en düşük entropiye sahip olan karar seçilecektir. Aslında bu adımda yapılan seçim bilgi kazanımı (information gain) açısından incelenebilir, yani bilgi kazanımı (information gain) entropinin tersidir ve bu yüzden en düşük entropiye sahip olan karar önce alınmaktadır.

Bu açıdan bakıldığında bir aç gözlü yaklaşımı (greedy approach) olarak görülebilir. Her adımda en düşük entropiye sahip olan karar ihtimali seçilmektedir. Bu seçim belki ilerideki adımların entropi değerlerinin yükselmesine sebep olacaktır ancak ID3 bununla ilgilenmeden aç gözlü olarak ilk en iyi ihtimali alır.

Neticede çizilecek olan karar ağacı aşağıdaki şekilde olacaktır:



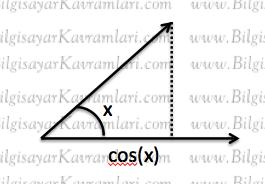
Yukarıdaki ağacın en büyük özelliği en az dağılımla karar veriyor olmasıdır. Yani Tüylü ve Sıcak Kanlı ayrımlarından sonraki ayrımların bir öneminin olmaması ve ağacı sadece 2 seviyeye indiriyor olmasıdır.

## **SORU-2: Kosinüs Benzerliği (Cosine Similarity) hakkında bilgi veriniz.**

Bu yazının amacı, bilgisayar bilimlerinin de bir alt çalışma alanı olan, metin madenciliği konularında (text mining) sıkça geçen kosinüs benzerliği (cosine similarity) konusunu açıklamaktır.

Basitçe iki farklı doküman (text) arasındaki benzerliği, trigonometrideki kosinüs (cosine) fonksiyonu üzerinden formülize etmek amaçlanmaktadır.

Metinlerin birer vektör (yöney, vector) olarak düşünüldüğü bu yaklaşımda, iki vektörün birbirine göre olan ilişkisi bir açı ile ifade edilmektedir.



Örneğin tamamen aynı yönü gösteren, yani tamamen biribiri ile aynı olan iki vektör için kosinüs değeri 1 olacaktır ( cos(0) = 1 ) . Tamamen biribiri ile ilişkisiz olan vektörler için ise kosinüs değeri 0 olacaktır ( cos(90) = 0, orthogolan vectors (dik vektörler) ). Son olarak tamamen birbirini zıddı olan dokümanlar için, ki bu dokümanlar için aralarında ilişki yoktur denemez, kosinüs değeri -1 olacaktır ( cos(180) = -1 ) Diğer bütün değerler, +1 ile -1 arasında yer alacaktır. Dolayısıyla iki doküman arasındaki benzerlik bu yöntemle sayısallaştırılabilir.

Ayrıca vektörlerin doğrusal bağımlılığı konusunda da analiz için kullanılabilir.

Örnek: Yukarıda anlatılan konuyu bir örnek üzerinden açıklamaya çalışalım. Örneğin iki dokümanımız olsun (örnek açısından kısa tutacağım) ve bu dokümanlar arasındaki kosinüs benzerliğini bulmaya çalışalım:

Dok1 : Şadi bilgisayar kavramlarında yazıyor.

Dok2 : bilgisayar kavramlarında kosinüs benzerliği yazıyor.

Bu iki dokümandan özellik çıkarımı yapılması gerekir. Bu aşamada çok çeşitli özellikler kullanılabilir ve hepsi de kosinüs benzerliğine parametre olabilir. Ben basitlik açısından terim frekansını (term frequency) kullanacağım ve iki dokümanda da geçen terimlerin sayılarını bir vektörde göstermeye çalışacağım.

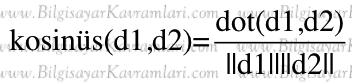
Terimlerimiz: {şadi, bilgisayar, kavramlarında, kosinüs, benzerliği, yazıyor}

Bu terimlerin dokümanlarda kaçar kere geçtiğini bir vektör ile gösterecek olursak:

Dok1: [1,1,1,0,0,1]

Dok2: [0,1,1,1,1,1]

şeklinde gösterilebilir. Şimdi bu iki vektör arasındaki kosinüs değerini hesaplayalım. Burada biraz matematik bilgilerimize geri dönüp formülü hatırlıyoruz:



Burada iki vektör arasındaki kosinüs bağlantısı için iki vektörün çarpımının (dot product) iki vektörün boylarının çarpımına oranı alınmıştır.

Şimdi çarpımı hatırlayalım. Boyutların ayrı ayrı çarpılmasından elde edilen değerlerin toplamıdır. Yani örnek vektörlerimiz için

dot(d1,d2) = [1,1,1,0,0,1].[0,1,1,1,1,1] = (1)\*(0) + (1)\*(1) + (1)\*(1) + (0)\*(1) + (0)\*(1) + (1)\*(1) = 3

sonucu elde edilir. Bu sonuç iki vektörün noktasal çarpımıdır (dot product).

Şimdi vektörlerin boyutlarını hesaplayabiliriz. Bunun için öklit uzayından (euclidean space) faydalanıyoruz ve her boyuttaki değerlerinin karelerini toplayarak toplamın karekökünü alıyoruz.

||d1|| = sqrt((1)^2 + (1)^2 + (1)^2 + (0)^2 + (0)^2 + (1)^2) = 2

||d2|| = sqrt((0)^2 + (1)^2 + (1)^2 + (1)^2 + (1)^2 + (1)^2) = 2.2360679775

olarak boyutları da hesaplanmış oluyor. Son olarak formülde değerleri yerine koyduğumuz zaman:

cos(d1,d2) = dot(d1,d2) / (||d1|| ||d2||) = 3 / ( 2 x 2.2360679775) = 3 / 4.472135955 = 0.67082039325

olarak iki doküman arasındaki kosinüs benzerliğini bulmuş oluyoruz. Yani bu iki dokümanın kosinüs benzerliğine göre %67′si benzerdir demek yanlış olmaz.

## **SORU-3: Cache (Önbellek) Teknolojilerinin Okuma Hızları Yönünden Karşılaştırılması hakkında bilgi veriniz.**

AMAÇ

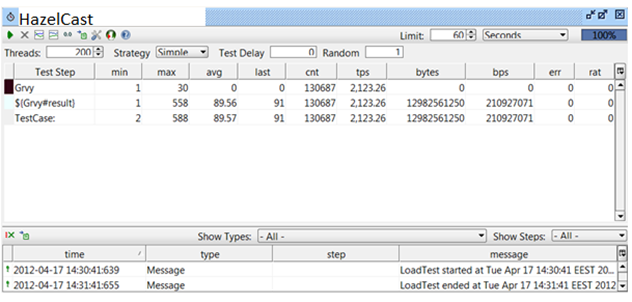
Bu çalışmada Memcached, HazelCast, Redis cache mekanizmaları 100 K lik PNG veriler kullanılarak okuma hızlarına göre karşılaştırılmıştır.

YÖNTEM

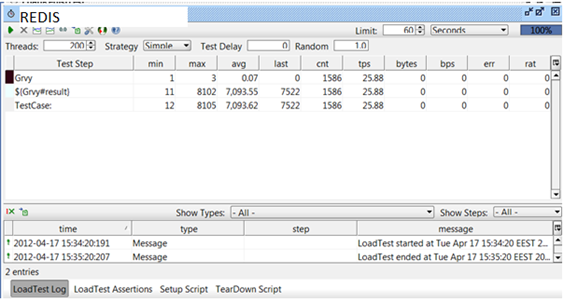
Önbelleğin (Cache) kullanıldığı ortam Java Servlet olarak belirlenmiştir. 10000 adet key-value kullanılmıştır. Tüm PNG dosyaları Cache lendikten sonra okuma işlemine başlanmıştır. Servlet kodları ekte sunulmuştur.

ÖLÇÜMLER

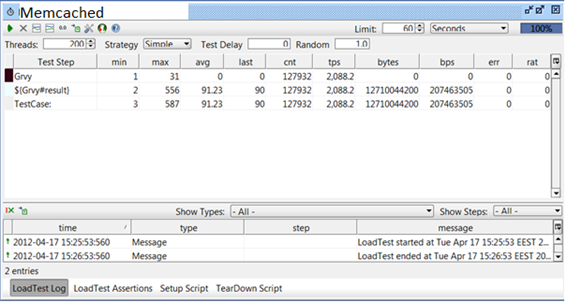
**1- HazelCast PNG OKU**



**2- Jedis PNG OKU**



**3- Memcached PNG OKU**



SONUÇ

HazelCast ve Memcached okuma hızlarına göre çok yakın sonuçlar vermiş (200 T için 90 ms) tir. Redis benzer koşullarda çok kötü değerler (200 T için 7093 ms) üretmiştir.

KAYNAK KODLAR

EK 1 – HAZEL CAST

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)[View Code](javascript:;) JAVA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 | @WebServlet(name = "Hazel", urlPatterns = {"/Hazel"})  **public** **class** Hazel **extends** HttpServlet {  **private** **static** IMap<String, **byte**[]> cache = **null**;  **private** **static** HazelcastInstance client = **null**;  **private** **static** Object \_synchObject =**new** Object() ;    **protected** **void** initHazelClient() {  **if** (cache == **null**) {  **synchronized** (\_synchObject) {  **if** (cache == **null**) {  ClientConfig clientConfig = **new** ClientConfig();  clientConfig.getGroupConfig().setName("dev").setPassword("dev-pass");  clientConfig.addAddress("localhost");  client = HazelcastClient.newHazelcastClient(clientConfig);  cache = client.getMap("pngs");  }    }    }    }    **protected** **void** processRequest(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  initHazelClient();  response.setContentType("image/png");  OutputStream out = response.getOutputStream();  **try** {    String path = request.getQueryString();  **byte**[] bytes = cache.get(path);  **if** (bytes == **null**) {  *//-------------------*  ByteArrayOutputStream baos = **new** ByteArrayOutputStream();  FileInputStream fis = **new** FileInputStream(**new** File("c:/temp/tile/" + path));  **byte**[] buf = **new** **byte**[1024];  **int** len;  **while** ((len = fis.read(buf)) > 0) {  baos.write(buf, 0, len);  }  fis.close();  baos.close();    bytes = baos.toByteArray();  cache.put(path, bytes);  *//--------*  }    out.write(bytes);  out.flush();  } **finally** {  out.close();  }}    @Override  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  processRequest(request, response);  }    @Override  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  processRequest(request, response);  }      } |

EK 2- MEMCACHED

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)[View Code](javascript:;) JAVA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | **package** cache;    **import** java.io.ByteArrayOutputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.OutputStream;  **import** java.io.PrintWriter;  **import** java.net.InetSocketAddress;  **import** javax.servlet.ServletException;  **import** javax.servlet.annotation.WebServlet;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** net.spy.memcached.MemcachedClient;    @WebServlet(name = "Memcached", urlPatterns = {"/Memcached"})  **public** **class** Memcached **extends** HttpServlet {    **private** **static** MemcachedClient cache = **null**;  **private** **static** Object \_synchObject=**new** Object();    **protected** **void** processRequest(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {    **if** (cache == **null**) {  **synchronized** (\_synchObject) {  **if** (cache == **null**) {  cache = **new** MemcachedClient( **new** InetSocketAddress("10.100.1.251", 11211));  }  }  }    response.setContentType("image/png");  OutputStream out = response.getOutputStream();    **try** {  String path = request.getQueryString();  **byte**[] bytes = (**byte**[]) cache.get(path);  **if** (bytes == **null**) {  *//-------------------*  ByteArrayOutputStream baos = **new** ByteArrayOutputStream();  FileInputStream fis = **new** FileInputStream(**new** File("c:/temp/tile/" + path));  **byte**[] buf = **new** **byte**[1024];  **int** len;  **while** ((len = fis.read(buf)) > 0) {  baos.write(buf, 0, len);  }  fis.close();  baos.close();    bytes = baos.toByteArray();  cache.set(path, 0, bytes);  *//--------*  }  out.write(bytes);  out.flush();  } **finally** {  out.close();  }  }    @Override  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  processRequest(request, response);  }      @Override  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  processRequest(request, response);  }    } |

EK 3- REDIS

[?](http://www.ericbess.com/ericblog/2008/03/03/wp-codebox/#examples)[View Code](javascript:;) JAVA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | **package** cache;    **import** java.io.ByteArrayOutputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.IOException;  **import** java.io.OutputStream;  **import** java.io.PrintWriter;  **import** javax.servlet.ServletException;  **import** javax.servlet.annotation.WebServlet;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** redis.clients.jedis.Jedis;  **import** redis.clients.jedis.JedisPool;    @WebServlet(name = "Redis", urlPatterns = {"/Redis"})  **public** **class** Redis **extends** HttpServlet {    **private** **static** **final** JedisPool jedisPool = **new** JedisPool("localhost", 6379);  **private** **static** **final** Jedis cache = jedisPool.getResource();  **protected** **void** processRequest(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {      response.setContentType("image/png");  OutputStream out = response.getOutputStream();    **try** {  String path = request.getQueryString();  **byte**[] bytes = cache.get(path.getBytes());    **if** (bytes == **null**) {  *//-------------------*  ByteArrayOutputStream baos = **new** ByteArrayOutputStream();  FileInputStream fis = **new** FileInputStream(**new** File("c:/temp/tile/" + path));  **byte**[] buf = **new** **byte**[1024];  **int** len;  **while** ((len = fis.read(buf)) > 0) {  baos.write(buf, 0, len);  }  fis.close();  baos.close();    bytes = baos.toByteArray();      cache.set(path.getBytes(), bytes);  *//--------*  }    out.write(bytes);  out.flush();  } **finally** {  out.close();  }  }    @Override  **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  processRequest(request, response);  }    } |

## **SORU-4: Son i7 ailesi, vPro AMT ve SCS teknolojileri hakkında bilgi veriniz.**

VPro teknolojisi kısaca virtualization professional kelimelerinden oluşmaktadır ve işlemci üzerinde yapılan yeniliklerle işlemcinin ağ ve dolayısıyla internet üzerinden kontrolünü ön görür.

İntel ailesininde 37 ile başlayan ve daha üzer işlemcilerinde bulunur (örneğin 3820qm) ve daha önceki işlemcilerinde örneğin i7-3610, i7-3612 veya i7-3615 işlemcilerinde bulunmaz.

Senaryo basitçe şöyle işler, vPro özelliği olan bir işlemci ve bağlı olduğu bir chipset olan ana kartınız ağa bağlıdır. Bu sayede size aşağıdaki özellikler sunulur:

1. çalışnmaya karşı güvence (anti-theft technology) : Basitçe işlemcinin ağ üzerinden kontorlü, kapatılması veya bulunması hedeflenir
2. Activite yönetim teknolojisi (activate management technology):
3. Sanallaştırma teknolojisi (virtualization technology):
4. Güvenilir çalıştırma teknolojisi (tursted execution technology):

Yukarıdaki özelliklerin kullanılabilmesi için ortamda yönetim konsolu (management console), ayarlama yazılımları (configuration software) sertifika sunucusu (certificate authority) veya windows active directory benzeri yazılım ve sunucular gerekmektedir.

Burada kritik nokta, işletim sisteminin daha alt seviyesinde çalışan, chipset ve ağ arayüzü arasında çalışan AMT (activity management technology, aktivite yönetim teknolojisi) bulunmasıdır. İşletim sisteminin bu teknolojiye müdahalesi söz konusu değildir.

Bu AMT teknolojisi sayesinde örneğin bilgisayar açılıp kapatılabilir, bilgisayar, CD, DVD, sabit disk USB gibi cihazlardan başlatılabilir (boot) bilgisayarın uzaktan kontrolü (KVM teknolojisi ile grafik veya metin tabanlı olarak) mümkündür ve hatta 3PDS ismi verilen teknoloji ile (3rd party data store) AMT kısmına ilave veri konulması mümkündür. Ayrıca yapılan işlemlerin kaydının tutulması ve çeşitli durumlarda alarm oluşturulması imkanı da sunulur.

Kısacası 3 yeni teknoloji intel ailesinin yeni işlemcilerinde sunulmaktadır:

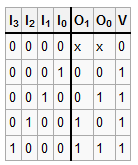
* vPro ile işletim sisteminin sanallaştırılması ve donanım ile işletim sistemi arasında ilave bir yazılım sunulması mümkündür, işlemcinin bazı bilgilerine uzaktan erişmek ve kontrol etmek mümkündür.
* AMT (active management technology) ile anakart ve ağ teknolojilerinin de dahil olduğu bir ortamda vPro özelliklerinin kullanılması sağlanır.
* SCS (Setup and Configuration Software) ile yukarıda bahsi geçen 2 teknolojinin uzaktan kontrol edilmesi mümkündür.

## **SORU-5: Encoder (Kodlayıcı) hakkında bilgi veriniz.**

Basit bir kodlayıcı, kod çözücünün (decoder) tersine üssel işlemi geri alır. Örneğin bir kod çözücüde, yapılan işlem 2n şeklinde gelen girdinin (input) üstünü almaktır. 3×8 bir kod çözücüde, gelen 3 bitlik girdinin (input) değeri n olarak kabul edilirse, kod çözücü bu değere göre 8 farklı çıktıdan (output) bir tanesini seçer.

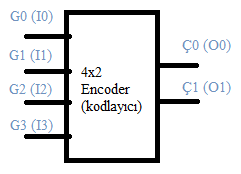
Kodlayıcı ise bu işlemin tam tersi yönde 8 farklı girdiden birisinden sinyal gelmesi halinde 3 çıktıdan (output) ilgili ihtimalleri işaretleyerek üst alma işleminin tersini (logaritma) yapar.

Örneğin aşağıda bir 4×2 kodlayıcının (encoder) [doğruluk tablosu (truth table)](http://www.bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) verilmiştir:



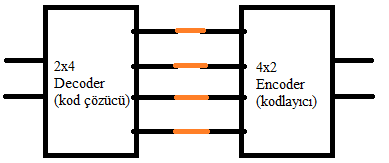
Tabloda I ile ifade edilen kolonlar girdi (input) ve O ile ilfade edilen kolonlar ise çıktı (output) değerlerdir. Örneğin 0100 değerinin 10′luk tabanda karşılığı 4 olarak yazılabilir. Bu değerin tablodaki çıktı değeri (output) 10 olarak okunacaktır. 10 değeri ise 10′luk tabanda 2 olarak yazılabilir. Gerçekten de log24 = 2 olmaktadır ve kodlayıcının bir logaritma işlemi olduğu görülebilir.

Yukarıdaki tabloyu gerçekleyen kodlayıcının genelde kullanlıan blok çizimi aşağıdaki şekildedir:



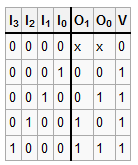
Ayrıca doğruluk tablosunda görülebileceği üzere, V biti (valid bit, kabul edilebilir) kullanılarak tanımsız durumlar ortadan kaldırılabilir. Örneğin logaritmanın tanımından bilindiği üzere 0′ın logaritması tanımsızdır. Bu durumda bütün girdi (input) bitlerinin 0 olması durumunda çıktı belirsiz olacaktır. İşte bu belirsizlik durumunda çıktının kabul edilemez (invalid) olduğunu ifade için V biti 0 değerinde verilebilir.

Şayet bir kod çözücü (decoder) ile bir kodlayıcı (encoder) arka arkaya bağlanırsa, sistemin girdi değeri, çıktı değeri olarak okunur.



Yukarıdaki devrede, soldan verilen girdi sağdan değişmeden okunurken devre tam tersine çevrilerek, sağdan bir girdi verilmesi halinde de soldan okunacaktır.

Kodlayıcı devresini, kapılar kullanarak yapmak da mümkündür. Örnek bir tasarım aşağıda verilmiştir:



Doğruluk tablosunun karnaugh haritasını (karnaugh map) çizersek:

O0 için

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | I1I0= 00 | I1I0= 01 | I1I0= 11 | I1I0= 10 |
| I3I2= 00 | X | 0 | X | 1 |
| I3I2= 01 | 0 | X | X | X |
| I3I2= 11 | X | X | X | X |
| I3I2= 10 | 1 | X | X | X |

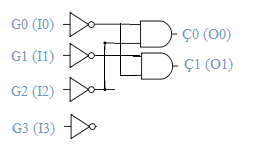
Yukarıdaki haritada, sonucu etkilemeyen (kodlayıcının çalışması belirsiz ve hiçbir şekilde girdi olarak gelemeyecek değerler) X ile ifade edilmiştir. Bu tip kodlayıcılara (encoder) özel olarak öncelik kodlayıcısı (priority encoder) ismi de verilmektedir. Bu haritada X değerleri 1 veya 0 olarak kabul edilebilir. O halde yukarıdaki tabloda mavi ile işaretlenmiş olan 4 ihtimal tek başına alınarak O0 için I0‘I2‘ sonucuna varılabilir.

O1 için

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | I1I0= 00 | I1I0= 01 | I1I0= 11 | I1I0= 10 |
| I3I2= 00 | X | 0 | X | 0 |
| I3I2= 01 | 1 | X | X | X |
| I3I2= 11 | X | X | X | X |
| I3I2= 10 | 1 | X | X | X |

Yukarıdaki tabloda da benzer şekilde O1 için I1‘I0‘ sonucuna varılabilir.

Yukarıdaki sonuçlara göre bir kodlayıcıyı (encoder) aşağıdaki şekilde çizebiliriz:



## **SORU-6: Banker Algoritması (Banker's Algorithm) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde işletim sistemi tasarımı konusunda geçen ve kaynaklar üzerindeki kilitlenmeyi (deadlock)engelleme amaçlı algoritmadır. Algoritma Dijkstra tarafından geliştirilmiştir.

Algoritmanın temel 3 durumu ve 2 şartı bulunur:

Bilmesi gerekenler:

1. Her işlem (process) ne kadar kaynağa ihtiyaç duyar?
2. Her işlem (process) şu anda ne kadar kaynağı elinde tutmaktadır?
3. Şu anda ne kadar kaynak ulaşılabilir durumdadır?

Yukarıdaki bu bilgileri bildikten sonra kaynak ayrılması sırasında aşağıdaki şartları uygular:

1. Şayet talep edilen kaynak, azami kaynaktan (maximum) fazla ise izin verme
2. Şayet talep edilen kaynak eldeki kaynaktan fazla ise, kaynak boşalana kadar işlemi (process) beklet.

Yukarıdaki algoritma detayında, kaynak olarak geçen değer, işletim sistemi için herhangi bir şey olabilir (hafıza (RAM), giriş çıkış işlemleri (I/O), gerçek sistemler için çalışma zamanı gibi)

Banker algoritması ayrıca çalışması sırasında yukarıdaki takipleri yapabilmek için bazı veri yapılarına ihtiyaç duyar. Aşağıdaki veri yapıları için n, sistemdeki işlem (process) sayısı ve m birbirinden farklı kaynak sayısı (resource) olmak üzere:

Müsait : m adet elemanı olan bir dizidir. Dizinin her elemanı o kaynak tipinden ne kadar müsait olduğunu tutar. Örneğin M[i] = 5 değerinin anlamı, i kaynak tipinden beşinin müsait olduğudur.

Azami : iki boyutlu dizi ile tutulur ve n x m boyutlarındadır. Her işlem için ilgili kaynaktan ne kadar ayrım yapıldığı dizide işaretlenir. Örneğin Az[2][3]=5 gösteriminin anlamı, 2. işlemin 3. kaynak üzerinde 5 birimlik ayrım hakkı olduğudur. Diğer bir deyişle 2. işlem, 3. kaynaktan 5 birimden fazla kullanmaz, kullanmak istemez, kullanamaz.

Ayrım : yine iki boyutlu bir dizidir ve yine boyutu n x m olarak tutulur. Her işlemin her kaynağın ne kadarını kullandığını gösterir. Örneğin Ay[2][3] = 4 gösteriminin anlamı, o anda, 2. işlemin 3. kaynaktan 4 birim kullanıyor olduğudur (ayırmış olduğudur).

İhtiyaç: Benzer şekilde nxm boyutlarında bir dizi olarak tutulur ve her işlemin her kaynaktan ne kadar ihtiyaç duyduğunu tutar. Örneğin İ[2][3] = 1 gösteriminin anlamı, 2. işlemin, 3. kaynaktan 1 birim ihtiyacı olduğudur.

Algoritmanın çıktısı güvenli veya değildir (safe state, unsafe state). Buna göre algoritma, işlemlerin çalışıp bitme ihtimali varsa güvenli sonucunu döndürürken, işlemler, birbirini kilitliyorsa bu durumda güvensiz sonucunu döndürür.

Hesaplama:

Algoritma, güveni veya güvensiz sonucuna ulaşmak için adıma bağlı olarak, aşağıdaki hesaplamaları yapar:

Müsait = Müsait – İhtiyaç

Ayrım = Ayrım + İhtiyaç

İhtiyaç = Azami – Ayrım

Bu durumu bir örnek üzerinden açıklayalım. Örneğin A,B,C isimli üç kaynağımız olsun ve bu kaynakların müsaitlik durumları aşağıdaki şekilde olsun:

Kaynaklar

A B C

10 5 7

Ayrıca 5 adet işlemimiz olsun ve bu işlemlerin anlık olarak azami ihtiyaçları, ayrılmış olan kaynaklar ve ihtiyaçları aşağıda verildiği gibi olsun:

Azami Ayrım

A B C A B C

P0 7 5 3 0 1 0

P1 3 2 2 2 0 0

P2 9 0 2 3 0 2

P3 2 2 2 2 1 1

P4 4 3 3 0 0 2

Soru bu durumun güvenli bir durum olup olmadığıdır. Diğer bir deyişle acaba bir kilitlenme riski bulunur mu? Öncelikle yukarıdaki ayrım tablosunda bulunan kaynakları toplayalım:

A kaynağı için : 0 + 2 + 3 + 2 + 0 = 7

B kaynağı için : 1 + 0 + 0 + 1 + 0 = 2

C kaynağı için : 0 + 0 + 2 + 1 + 2 = 5

yukarıdaki toplama işlemleri, ilgili kayakatan, başlangıçta ayrılmış miktarların toplamıdır ve kaynağın bulunduğu kolonun toplanması ile bulunabilir.

Şimdi başlangıçtaki kaynak miktarından bu toplamları çıkararak başlangıçta bir işlemin çalışması için müsait kaynakları bulalım:

A : 10 – 7 = 3

B : 5 – 2 = 3

C: 7 – 5 = 2

Bu kaynaklarla başlayarak acaba sistem kilitlenme olmadan çalışabilir mi?

Yukarıda verilen değerlerin üzerinden ihtiyaç tablomuzu hesaplayalım:

Azami - Ayrım = İhtiyaç

A B C A B C A B C

P0 7 5 3 0 1 0 7 4 3

P1 3 2 2 2 0 0 1 2 2

P2 9 0 2 3 0 2 6 0 0

P3 2 2 2 2 1 1 0 1 1

P4 4 3 3 0 0 2 4 3 1

Yukarıdaki işlemden anlaşılacağı üzere Azami tablosundan, Ayrım tablosu çıkarılmış ve ihtiyaçlar bulunmuştur.

Bu durumda, sistem güvenli denilebilir çünkü ihtiyaç tablosundaki hiçbir değer, müsait dizimizdeki değeri geçmemektedir. Demek ki sistem bütün ihtiyaçlara cevap verebilecek kapasitededir.

Bu durumun daha iyi anlaşılması için P1, P2, P3, P4, P0 sırası ile çalışmayı görelim:

Azami - Ayrım = İhtiyaç Müsait

A B C A B C A B C A B C

P1 3 2 2 2 0 0 1 2 2 3 3 2 3 3 2

P3 2 2 2 2 1 1 0 1 1 5 3 2

P4 4 3 3 0 0 2 4 3 1 7 4 3

P2 9 0 2 3 0 2 6 0 0 7 4 5

P0 7 5 3 0 1 0 7 4 3 10 4 7

10 5 7<<< Kaynaklar

Görüldüğü üzere işlemlerin sonucunda ilk kaynak değerlerine geri dönülmüştür (aslında bütün işlemler bitince müsait olan kaynak değeri, başlangıçtaki değerdir).  
Yukarıdaki çalışma sırası (yani P1, P2, P3, P4, P0 sırası) güvenli çalışma olarak kabul edilir. Örneğimizi biraz daha ilerletelim :  
Acaba P1 işlemi (1,0,2) ihtiyacında olsaydı yine de güvenli durumda olur muyduk?  
İhtiyaç kontrolümüz (1,0,2)<= (1,2,2) olacaktı  
ve müsait durumumuz : (1,0,2)<= (3 3 2) olacaktı  
Bu durumda tablomuz aşağıdaki şekilde olabilirdi:

Azami Ayrım İhtiyaç Müsait

A B C A B C A B C A B C

P0 7 5 3 0 1 0 7 4 3 2 3 0<<<

P1 3 2 2 3 0 2<<< 0 2 0<<<

P2 9 0 2 3 0 2 6 0 0

P3 2 2 2 2 1 1 0 1 1

P4 4 3 3 0 0 2 4 3 1

Bu durum güvenli midir?  
Evet !  
Çalışma sırası olarak P1, P3, P4, P0 ve P2 sırasında olması durumunda işlemler kilitlenme olmaksızın başarı ile çalışacaktır. Örneğin P4 için ihtiyacı (3,3,0) olarak yeniden ayarlasaydık güvenli bir durum elde edemeyecektik çünkü bu adımdaki (3,3,0) değeri o anda müsait olan (2,3,0) değerinden büyük olacaktı ve işlem (process)çalışamayacaktı.

## **SORU-7: Ondalıklı sayıların taban dönüşümleri hakkında bilgi veriniz.**

Öncelikle küsurat kısmının payda olarak değerlendirilmesi gerektiğini bilmemiz gerekir.

Normalde bir sayıyı farklı bir tabana çevirirken, sayının çevrileceği tabandaki, taban değerinin üstleri ile çarpılması beklenir.

Örneğin onluk tabandaki 13 sayını ikilik tabana çevirirsek:

1 x 101 + 3 x 100 = 1 x 23 + 1 x 22 + 0x 21 + 1 x 20

Şeklinde yazabiliriz. Yani

(13)10 = (1101)2 şeklinde yazılabilir

Diğer bir deyişle, sayılar dönüştürüldüğü tabandaki kaçıncı hane ise, o tabanın hane sayısı kadar üstü ile çarpılır.

Ondalıklı sayılar için bu durum tam tersidir. Yani üstler tabanın üstü şeklinde değil tabanın eksi üstü (veya bir bölü tabanın üstü) şeklinde düşünülmelidir.

Örneğin onluk tabandaki bir sayıyı :

0.123 = 1x 1/10 + 2x 1/100 + 3x 1/1000 şeklinde düşünebiliriz.

Benzer şekilde farklı bir tabandaki sayıyı da o tabandaki üstler olarak düşünmek gerekir:

(0.23)8 = 2 x 1/8 + 3 x 1/64

Ondalıklı sayılarda dikkat edilecek bir husus, sayının üstünün normal sayılarda olduğu gibi 0′dan değil 1′den başlamasıdır.

Sayıların anlamını yukarıdaki şekilde açıkladıktan sonra çevirme işlemine geçebiliriz. Önce basit bir iki uygulamadan başlayalım:

(13.5)10 = ( ? )2

Yukarıdaki soruda onluk tabandaki bir sayıyı ikilik tabana çevirmemiz istenmiş. Çevirme işlemini tam sayı ve küsuratı olarak iki aşamada yapabiliriz. Önce tam kısım olan 13′ü çevirelim :

(13)10 = (1101)2 şimdi küsuratı çevirelim:

0.5 = ½ = (0.1)2

Sayının 0.1 olmasının sebebi ikilik tabandaki ilk kat sayının zaten ½ olmasıdır.

Sonuç olarak

(13.5)10 = (1101.1)2

Olarak bulunur.

Biraz daha ilerleyerek farklı bir soru çözelim (bundan sonraki örneklerde sayının sadece ondalık kısmı verilecektir, tam kısım ile nasıl birleştiği anlatılmıştır)

(0.125)10 = ( ? )2

Hemen sayımızı kesirli sayıya çevirelim : 0.125 = 1/8

Bu durumda ondalık kısmına bakıyoruz: 0 x ½ + 0 x ¼ + 1 x 1/8 , o halde sayımız (0.001) 2 olarak bulunuyor.

Farklı bir örnek yapalım

(0.625)10 = ( ? )2

Sayının çevrilmiş haline bakalım (aslında 0.625 = 0.5 + 0.125 olduğunu biliyoruz) : 1 x ½ + 0 x ¼ + 1 x 1/8 , o halde sayımız (0.101) 2 olarak bulunuyor.

Buna benzer şekillerde çevirim işlemi için değerler kullanılabilir. Kolaylık olması açısından çevrilen tabanın çevrilmeden önceki tabandaki değerlerini bulmak işe yarayabilir.

Örneğin ½ , ¼ gibi sayıların onluk tabandaki karşılığını bilmek işe yarar.

Farklı bir örnek ile devam edelim. 1/8′in 0.125 olduğunu biliyoruz ve bunu kullanarak 8′lik tabana çevirim yapalım:

(0.125)10 = ( ? )8

Bildiğimiz üzere 1/8 , 8′lik tabandaki ilk çarpan, o halde çözüm (0.1) 8 olarak bulunmuş olunur.

Gelelim farklı tabanlar arası çevirime. Örneğin on altılık tabandan çevirim yapmak istiyor olalım

(0.AB)16 = ( ? )8 = ( ? )2 şeklinde sayının hem 8′lik hem de 2′lik tabandaki karşılıklarını soralım.

Bilindiği üzere onaltılık tabandaki sayıların (hexadecimal) sekizlik tabana (octal) ve ikilik tabana (binary) çevirimi sırasında bir kolaylık vardır.

Öncelikle ikilik tabandan başlayalım. Tek bir onaltılık sayı, ikilik tabanda 4 haneye tekabül eder.

A = 1010

B = 1011

Olarak yazılabilir. Bu değerleri çevirelim ve yazalım.

Örneğin sorumuz (0.A)16 = ( ? )2 şeklinde olsaydı hemen ( 0.1010 )2 yazabilirdik. Elbette sonda buluna 0 değeri hükümsüz olduğu için sonuca ( 0.101 )2 diyecektik.

Sorumuza dönersek (0.AB)16 = ( 0.10101011 )2 şeklinde bulmuş oluruz.

Gelelim sekizlik tabana. Bu sonucu, ikilik tabandan bulmak çok daha kolaydır. Burada her 3 biti (ikilik tabandaki her bir haneye bit denir) bir sayıya karşılık gelecek şekilde çeviriyoruz.

0.101 010 011 = 0. 5 2 6

Olarak çevrilir. Burada dikkat edilecek husus, çevirim işleminin tersten yapılmasıdır. Yukarıda da yazarken belirttiğim üzere hanelerin üstlerinin mutlak değeri, sağa doğru giderken büyür. Yani 0.101010011 sayısının çarpanlarının üstlerinin en küçüğü (veya bölüm olduğu için veya üstler eksi değerde olduğu için, mutlak değeri en büyük olan üst) sonda buluna 1 sayısının çarpanıdır. Ki bu sayı aşağıdaki şekilde düşünülebilir:

1 x ½ + 0 x ¼ + 1 x 1/8 + 0 x 1/16 + 1 x 1/32 + 0 x 1/64 + 0 x 1/128 + 1 x 1/256 + 1 x 1/512

Görüldüğü üzere en büyük çarpan 512 değeridir. Dolayısıyla 8′lik tabana çevirirken en önemli bit bu değerle çarpılacaktır.

Aslında literatürde bu çarpımda en küçük değere sahip bit anlamında least significant bit tabiri kullanılır. (En yüksek etkiye sahip bit için de most significant bit kullanılmaktadır).

Dolayısıyla 8′lik tabana çevirirken her üçlüyü kendi içinde ters çeviriyoruz:

0.101 010 011 = çevirim için 101 010 110 halini alır ve bunların onluk tabandaki karşılıkları, sonucu verir = 0. 5 2 6 olarak bulunmuş olur.

## **SORU-8: Mealy ve Moore Makineleri (Mealy and Moore Machines) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde sıkça kullanılan sonlu durum makinelerinin (finite state machine, FSM veya [Finite State Automaton , FSA](http://www.bilgisayarkavramlari.com/2008/08/02/sonlu-ototmatlar-finite-automaton/)) gösteriminde kullanılan iki farklı yöntemdir. Genelde literatürde bir FSM’in gösteriminde en çok moore makinesi kullanılır. Bu iki yöntem (mealy ve moore makinaları) sonuçta bir gösterim farkı olduğu için bütün mealy gösterimlerinin moore ve bütün moore gösterimlerinin mealy gösterimine çevrilmesi mümkündür.

Klasik bir FSM’de bir giriş bir de çıkış bulunur (input / output). Bu değerlerin nereye yazılacağı aslında iki makine arasındaki farkı belirler.

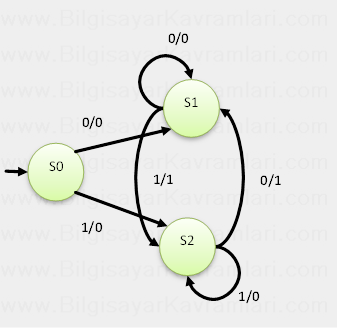
Moore makinelerinde çıkış değerleri düğümlere (node) yazılırken, giriş değerleri kenarlar (edges) üzerinde gösterilir.

Mealy makinelerinde ise giriş ve çıkış değerleri kenarlar (edges) üzerinde aralarına bir taksim işareti (slash) konularak gösterilir. Örneğin 1/0 gösterimi, girişin 1 ve çıktının 0 olduğunu ifade eder.

Basit bir örnek olarak özel veya (exclusive or, ) işlemini ele alalım ve her iki makine gösterimi ile de çizmeye çalışalım.

|  |  |  |
| --- | --- | --- |
| Girdi 1 | Girdi 2 | Çıktı |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Klasik bir XOR kapısı, yukarıdaki doğruluk çizelgesinde (truth table) gösterildiği üzere iki giriş ve bir çıkıştan oluşur. Bizim makinemiz de ilk girdiden sonra ikinci girdiyi aldığında beklenen çıktıyı vermeli. Ayrıca makine sürekli olarak çalışmaya devam edecek. Örneğin 101011 şeklinde bir veri akışı sağlanması durumunda, sonuç olarak 101011 değerini hesaplamasını isteriz.



Yukarıdaki gösterim bir mealy makinesidir. Makinede görüldüğü üzere 3 farklı durum arasındaki geçişler üzerine iki adet değer yazılmıştır. Bu değerlerden ilki giriş ikincisi ise çıkış değeridir.

Makineyi beraber okumaya çalışalım.

Makinenin boşluktan bir ok ile başlayan durumu, yani örneğimizdeki S0 durumu, başlangıç durumudur. Bu durumdan başlanarak gelen değerlere göre ilgili duruma geçilir.

Örneğimizi hatırlayalım. Giriş olarak 101011 dizgisini (string) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Burada geçiş sırasında kullanılan kenarın üzerindeki değeri okuyalım: 1/0 bunun anlamı 1 geldiğinde geçilecek kenar olması ve çıktının 0 olmasıdır. Yani şu anda çıktımız 0

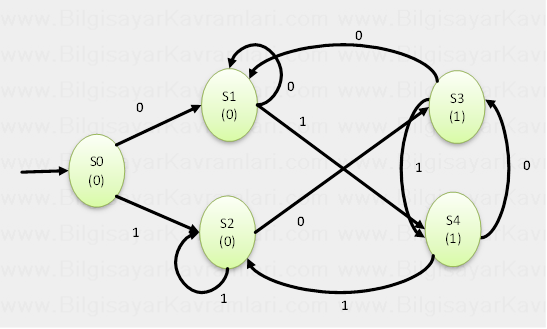
Ardından gelen değer 0 (yani şimdiye kadar 01 değerleri geldi). En son makinemizdeki durum, S2 durumuydu, şimdi yeni gelen değeri bu durumdaki kollardan takip ediyor ve S1′e giden 0/1 kenarını izliyoruz. Bu kolu izleme sebebimiz, S2 durumundan gidilen tek 0 girdisi kolu olmasıdır. Bu kol üzerindeki ikinci değer olan 1 ise, çıktının 1 olduğudur. Yani buraya kadar olan girdiyi alacak olsaydık 01 için 1 çıktısı alacaktık.

İşlemlere devam edelim ve durumları yazmaya çalışalım:

|  |  |  |  |
| --- | --- | --- | --- |
| Gelen Değer | Durum | Çıkış | Yeni Durum |
| 1 | S0 | 0 | S2 |
| 0 | S2 | 1 | S1 |
| 1 | S1 | 1 | S2 |
| 0 | S2 | 1 | S1 |
| 1 | S1 | 1 | S2 |
| 1 | S2 | 0 | S2 |

Yukarıdaki tablomuzun son halinde, çıkış değeri olarak 0 okunmuştur. Yani örneğimizin neticesi 0 olacaktır.

Aynı örneği moore makinesi olarak tasarlayacak olsaydık:



Moore makinesinde, mealy makinesine benzer şekilde boşluktan gelen bir ok, başlangıç durumunu belirtir.

Makinenin, durumlarında, mealy makinesinde olmayan değerler eklenmiştir. Bu değerler, ilgili durumdaki çıktıyı gösterir. Örneğin makinemiz S1 durumundayken çıktı 0 olarak okunabilir.

Şimdi moore makinesinde, aynı örneği çalıştırıp sonucu karşılaştıralım.

Giriş olarak 101011 dizgisini (string) almayı planlamıştık. Bu durumda ilk bitimiz 1 olarak geliyor ve S0 durumunda 1 girişi ile S2 durumuna geçiyoruz. Bu geçiş sonucunda geldiğimiz S2 durumunda okunan çıktı değeri 0 yani sonuç şimdilik 0.

Ardından gelen 0 değeri ile S3 durumuna geçiyoruz ve çıktımız 1 oluyor. Çünkü S3 durumu 1 çıktısı veren durumdur. Bu şekilde durumları ve durumlar arasındaki geçişleri izlersek, aşağıdaki tabloyu çıkarabiliriz:

|  |  |  |  |
| --- | --- | --- | --- |
| Gelen Değer | Durum | Çıkış | Yeni Durum |
| 1 | S0 | 0 | S2 |
| 0 | S2 | 1 | S3 |
| 1 | S3 | 1 | S4 |
| 0 | S4 | 1 | S3 |
| 1 | S3 | 1 | S4 |
| 1 | S4 | 0 | S2 |

Görüldüğü üzere, makinemiz, örnek girdi için S2 durumunda sonlanıyor ve bu durumda çıktımız 0 olarak okunuyor.

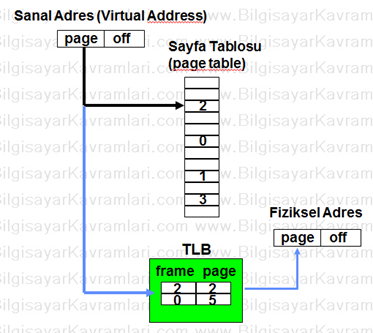
## **SORU-9: Translation Lookaside Buffer (TLB, Dönüşüm Hafızası) hakkında bilgi veriniz.**

TLB, sayfalama işleminin (paging) hızını arttırmaya yarayan bir hafıza bölümüdür. Kısaca TLB olması için sayfalama olmalıdır. Günümüzdeki çoğu bilgisayar mimarisi tarafından desteklenmektedir. TLB kullanılabilmesi için sayfalama (paging) sistemde yapılıyor olmalıdır.

Basitçe, RAM’in yetersiz olduğu durumlarda hafıza ihtiyacının diskten karşılanmasını sağlayan sanal hafızanın verimli kullanılması için, diskte tutulan veri ve RAM’de duran veri arasında bir dönüşüm işlemi gerekmektedir. Bu işin hızlanması için dönüşümü TLB üzerinde tutar ve hız artışı sağlarız.

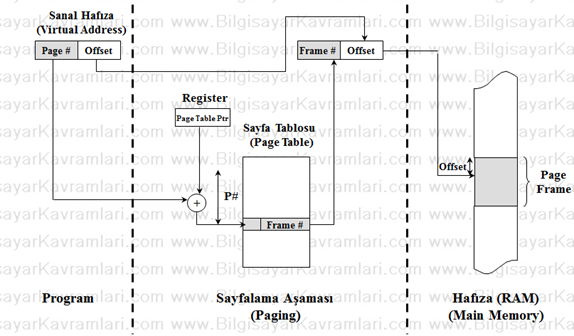
Bu yapıp, disk ile RAM arasında bir önbellek (cache) olarak düşünmek mümkündür. Basitçe diskte olan bir veriye erişilmek istendiğinde TLB üzerinde bu veri aranır. Şayet veriye ulaşılırsa buna TLB bulma (TLB hit) ismi verilir ve dönüşüm sağlanır. Şayet veri bulunamazsa buna TLB kayıp (TLB miss ) ismi verilir ve sanki TLB hiç yokmuş gibi klasik sayfalama tablosunda (page table) arama işlemi yapılarak bu hafıza sayfasına(page) erişilmeye çalışılır. Elbette bu işlem çok daha maliyetlidir. Dolayısıyla TLB üzerindeki bulma işlemini (hit) arttırmak isteriz.

TLB yapısını, aşağıdaki şekil üzerinden anlamaya çalışalım:



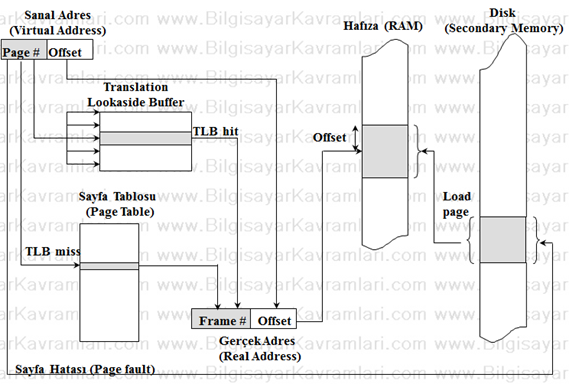
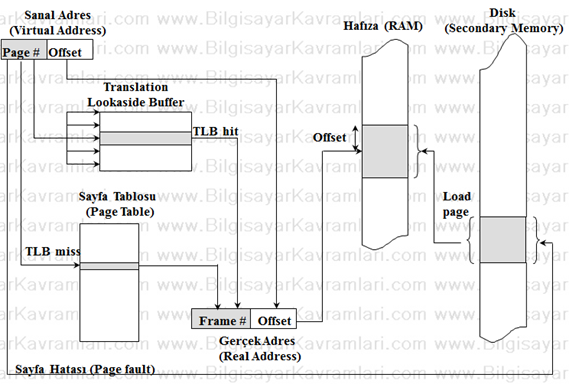
Yukarıdaki örnekte, bir sanal adresin, fiziksel adrese dönüşmesi sırasında geçtiği yol gösterilmiştir. Örneğin gelen sanal adres talebimiz, 2 veya 0 ise, bu değerler TLB üzerinden karşılanır. Şayet talep 0 veya 1 ise bu değerler sayfa tablosu üzerinde aranarak karşılanır. Sayfa tablosu üzerindeki arama işlemi, TLB üzerindeki arama işlemine göre çok daha uzun sürmektedir.

Konuyu, Sayfalama (paging) ile birlikte ele almak için, öncelikle sayfalama işlemini hatırlayalım.



Yukarıda görüldüğü üzere, sayfalama işlemi, programın kendi adresinden, gerçek adrese (fiziksel adres) dönüşümü sırasında kullanılır. Bu sırada, sayfalama yaklaşımı hafızadaki bilgilerin dağılımını tutan bir sayfalama tablosu bulundurur.

Şayet yukarıdaki şekle, TLB ve sanal hafıza (virtual memory) konularını da eklersek, aşağıdaki şekli elde ederiz:



Yukarıdaki yeni şekilde, bir sanal adresin, talep edilmesi durumunda yaşananları anlatmaya çalışalım.

Öncelikle bu adresin sayfa numarası, (ki sayfalama konusundan hatırlanacağı üzere sayfa numarası, adresin sayfa boyuna (page size) bölünmesi ile elde edilir).

Bu sayfa numarası, önce TLB üzerinde aranır. Bulunursa çerçeve numarasına ofset eklenerek hafızadan alınır.

Şayet TLB üzerinde bulunamazsa, TLB kayı (TLB miss) olur ve bu durumda Sayfa Tablosu üzerinde aranır. Şayet sayfa tablosunda bulunursa, yine ofset eklenir ve hafızadaki yerine ulaşılmış olunur.

Şayet Sayfa Tablosu üzerinde de bulunamazsa bu durumda sayfa hatası (Page fault) olur ve bunun anlamı, bu bilginin hafızada olmadığıdır. Artık bilgi diskten aranır ve bulunur. Ardından hafızaya (RAM) yüklenir ve sayfa tablosunda ilgili güncellemeler yapılır.

Yukarıdaki yazıda bazı terminolojik problemleri çözmek için bu notu eklemeyi önemli buluyorum. Bazı kaynaklarda, yukarıda sanal adres (virtual address) olarak geçen terime mantıksal adres (logical address), gerçek adres (real address) olarak geçen terime ise fiziksel adres (physical address) isimleri verilmektedir.

**TLB tablosunun güncellenmesi**

Klasik bir TLB tablosunda, bir verinin TLB üzerinde bulunamaması halinde bu veri güncellenir. Yani şayet veri TLB üzerinde bulunuyorsa, bu durumda bir güncellemeye ihtiyaç duyulmaz. Ama bir şekilde TLB kayıp oluşursa (TLB miss), bu durumda talebin Sayfa Tablosundan (Page Table) veya diskten karşılanması halinde TLB tablosu güncellenir.

Ancak daha önce de bahsedildiği üzere TLB aslında bir önbellek olarak düşünülebilir ve klasik önbellek güncelleme algoritmaları veya erişimli önbellek algoritmaları (associative cache) TLB için de kullanılabilir.

## **SORU-10: Çok Çekirdekli İşlemciler (Multi-Core Processors) hakkında bilgi veriniz.**

Bilgisayar mimarisi konusunda kullanılan bir teknolojidir. İşlemcinin alt çekirdeklerinin oluşturulması ve yükün bu çekirdekler üzerine dağıtılarak, aynı anda işlenmesini ve böylece işlem gücü elde edilmesini hedefler.

Çok çekirdekli işlemci mimarisinin anlaşılabilmesi için öncelikle çekirdek (core) kavramını anlamalıyız. Çekirdek, temel olarak bir işlemcinin okuma ve çalıştırma yapan kısmıdır. (read and execute)

Klasik bütün işlemciler tek çekirdeklidir. Bunun anlamı, klasik bir işlemcide, tek bir okumadan ve çalıştırmadan sorumlu parça bulunur. Dolayısıyla klasik bir işlemci, aynı anda tek bir işlem çalıştırabilir.

Çok çekirdekli mimarilerde, birden fazla çekirdek (genelde 2 4 veya 6 gibi 2′nin katları şeklinde) çekirdek, entegre devre olarak bulunur. Bu entegre devre yaklaşımından dolayı, literatürde, CMP (chip multi processor) tabiri de, çok çekirdekli işlemciler için kullanılmaktadır.

Çok çekirdekli işlemciler, oldukça geniş alanlarda kullanılabilir. Örneğin resim işleme, bilimsel verilerin işlenmesi veya dijital sinyal işleme gibi alanlarda kullanılabilirler. Problemlerin paralelleştirilebilmesi, işlemcideki hız artışı ile doğru orantılıdır. Örneğin, çok çekirdekli bir işlemci, paralel hale getirilemeyen bir problem için bir hız avantajı sağlamayabilir.

Dikkat edilmesi gereken bir nokta, çok çekirdekli mimariler ile, çok işlemcili mimariler arasında fark olduğudur. Çok çekirdek ile kast edilen bir işlemcinin içerisinde birden fazla çekirdek bulunmasıdır.



Yukarıdaki şekilde görüldüğü üzere, Tek bir veriyoluna bağlı çok sayıda çekirdek ve her çekirdeğin kendi içerisinde 1. seviye önbellek (level 1 cache, L1 cache) bulunmakta, 2. seviye önbellek ise, veri yolu üzerinde durmaktadır.

Bu sayede önbellek açısından (cache), her çekirdek kendi belleğini kullanırken, aynı zamanda, veri yolu üzerinde paylaşılmış bir önbellek kullanımı da mümkündür.

## **SORU-11: Flynn Sınıflanırması (Flynn’s Taxonomy) hakkında bilgi veriniz.**

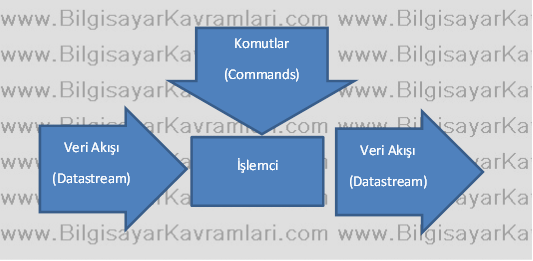
Micheal Flynn tarafından 1966 yılında yapılan bu sınıflandırma, bilgisayar mimarilerini 4 ana grup altında toplar. Bu gruplar aşağıdaki şekilde sıralanabilir:

* SISD (Single Instruction Single Datastream) Tek Komut Tek Veri akışı
* SIMD (Single Instruction Multiple Datastream) Tek Komut Çok Veri akışı
* MISD (Multiple Instruction Single Datastream) Çok Komut Tek Veri akışı
* MIMD (Multiple Instruction Multiple Datastream) Çok Komut Çok Veri akışı

Yukarıdaki bu sınıfların açıklamaları, aşağıda sunulmuştur.

**SISD**

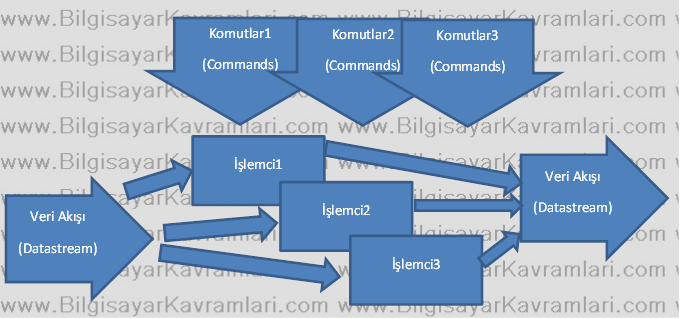
Bu sınıfta, tek işlemci ve tek hafıza bulunmaktadır.



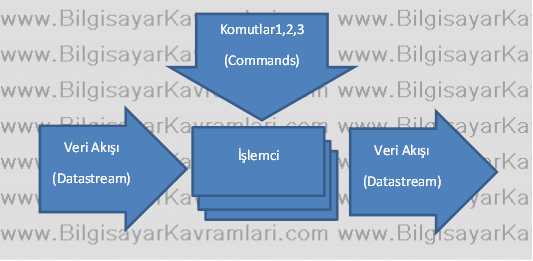
Yukarıda görüldüğü üzere, tek işlemci bulunmakta, bu tek işlemciye bilgisayarda bulunan veri yolu (BUS) sayesinde komutlar ve veriler akmaktadır. İşlemci bu akan veriler ve komutları alarak, işlenmiş verileri üretmektedir. Bu mimari klasik olarak Von Neumann makinesidir.

**MISD**

Bu sınıfta, birden fazla işlemci bulunmaktadır. Veri tek bir kaynaktan akmakta ve verinin işlenmiş hali yine tek bir kaynakta toplanmaktadır.



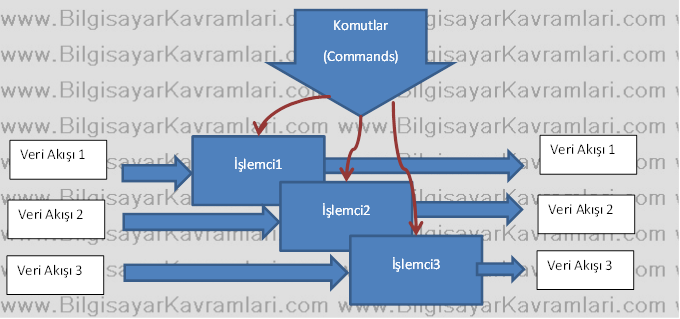
Bu anlamda, yukarıdaki işlemci modeli, bir önceki modelde olan SISD modelinde bulunan işlemci kutusunun açılmışı olarak düşünülebilir. Yani kutuya dışarıdan bakan kişinin, sistemden beklentileri ve sisteme etkisi, sanki tek işlemci bulunuyormuş gibidir. Ancak sistemin içerisinde, işlemci kutusu alt işlemciler tarafından desteklenmektedir.



Burada unutulmaması gereken, verinin tek olması ancak komutların çoklu olmasıdır. Bu mimari, gerçek uygulamalarda çok kullanılmaz. Biraz daha teorik bir tasarımdır. Bunun en büyük sebebi, kaynakları kullanımda MISD yaklaşımının, MIMD veya SIMD kadar başarılı olamamasıdır.

**SIMD**

Şayet veri farklı kaynaklardan geliyor ancak işlemci tek ise ve bu farklı kaynaklardan gelen veri, aynı anda işlemcide işleniyorsa, bu modele SIMD ismi verilir



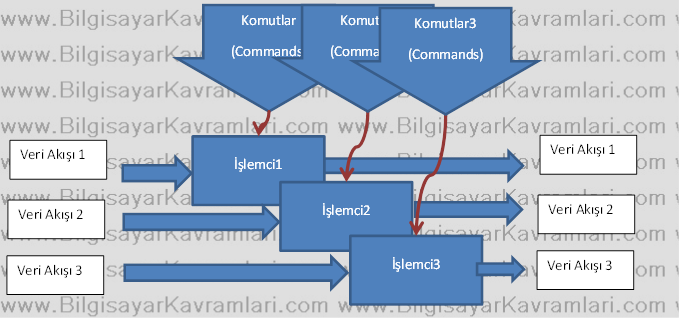
Yukarıdaki şekilde görüldüğü üzere, birden fazla veri akışı üzerinde tek komut listesi çalışmaktadır. Bu tek komutlu yaklaşım, SIMD mimarisinde işlemcilere uygun şekilde bölünmelidir.

SIMD mimarisinin en büyük avantajı, büyük, özdeş verilerin işlenmesinde ortaya çıkar. Örneğin resim işleme sırasında imgecik (pixel) bazlı yapılan işlemlerin dağıtılması ve birden fazla işlemci üzerine aynı komutun verilmesi mümkündür. Örneğin resimin aydınlatılması için, resimde bulunan her imgeciğin (pixel), belirli oranlarda aydınlatılması isteniyorsa, resimdeki imgecikler farklı işlemcilerde aynı komut ile paralel olarak aydınlatılabilir.

Diğer bir örnek matris çarpımı olabilir. Büyük bir matrisin, farklı işlemcilere bölünmesi ve bu bölünmüş işlemcilere aynı komutun verilmesi sonucunda matris üzerinde çarpma gibi işlemler gerçekleştirilebilir.

**MIMD**

Birden çok veri kaynağı, birden çok komut girişi ile çalışmaktadır.



Bu mimari modelinde, işlemciler bağımsız olarak ve asenkron olarak çalışabilmektedir. Bunun anlamı, bir işlemcinin diğer işlemcileri beklemesi gerekmemesi ve sonuçların birbirinden bağımsız olarak işlenebilmesidir.

MIMD modelinde, SISD modelinde oluşturulmuş birden fazla makine varmış gibi düşünülebilir. Bu düşünce, doğru olmakla birlikte, çoklu SISD modeline göre avantajlar vardır. Örneğin işlemcilerin aynı donanım içerisinde olması sayesinde, veri iletişimi kolaylaşabilmektedir.

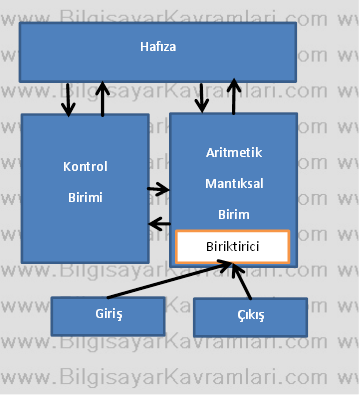
Zaten MIMD mimarisi de, kendi içerisinde paylaşılmış hafıza (shared memory) ve dağıtılmış hafıza (distributed memory) olarak iki grupta incelenir.

Paylaşılmış hafıza için durum, tek bir hafıza üzerinde çalışan birden fazla işlemcidir. Burada hafızdan kasıt, bilgisayarın birincil hafızası (primary memory, veya RAM) olarak düşünülebilir. Aynı problem işlemcilerin ön hafızaları (cache) için de bulunur. Yani işlemcilerin birden fazla önbellek kullanmaları veya tek önbellek üzerinden, birden fazla işlemcinin çalışıyor olması birer tasarım kriteridir.

Paylaşılmış hafıza olarak UMA, COMA veya NUMA mimari alternatifleri bulunur.

## **SORU-12: Von Neumann Makinesi hakkında bilgi veriniz.**

Bilgisayar bilimleri açısından, von neumann makineleri, günümüz klasik bilgisayarlarının temellerini oluşturur. Bu makine tanımında basitçe bir işlemci, bir hafıza ve giriş / çıkış sistemleri modellenmiştir.



Yukarıdaki şekilde görüldüğü üzere, hafıza (memory) ile bir kontrol birimi (control unit) ve bir de aritmetik mantıksal birim (artihmetic logical unit) arasında iletişim kurulmaktadır. Ayrıca aritmetik mantıksal birim içerisinde bir adet biriktirici bulunmaktadır.

Von neumann makinesi, tasarım olarak, evrensel Turing makinesi (universal Turing Machine) problemlerini çözebilmektedir.

Von Neumann makineleri, teoride ” kayıtlı program ” olarak geçen (stored programm) problemi çözebilmeleri açısından, “kayıtlı program makinesi” (stored programm machine) olarak geçmektedir. Bunun anlamı, program tarafından kontrol edilebilen bilgisayarların bir üst seviyesi olarak, programın ve giriş çıkışların kayıt edilebilmesidir.

Von Neumann makinelerinin ayrıca bir üst seviyesi olarak, yeniden programlanabilir makineler (reprogramming) olarak kullanılabilmesidir. Örneğin bir hesap makinesi için geliştirilen cihaz, üzerinde bir program bulundurur. Bu program, hesap işlemlerinin yapıldığı temel özellikleri içerir ve donanımla doğrudan ilişkilidir. Von Neumann makineleri, bu kavramı yıkarak aynı cihaz üzerinden, hesaplama işlemlerinin yapıldığı bir programı tamamen farklı amaçlara hizmet eden bir programla değiştirmeye izin verir. Örneğin bir oyun veya bir tablolama yazılımına geçilmesi cihaz üzerinde ciddi yenileme işlemleri gerektirir.

Yukarıda anlatılan bu yeniden programlama özelliğinin desteklenebildiği ilk bilgisayar tasarımı Von Neumann bilgisayarlarıdır. Bu bilgisayarlar üzerinde hafızada kayıtlı bir program amaca yönelik olarak yüklenebilmektedir. Elbette aynı anda birden fazla programın çalışabilmesi (multi processing) çok daha sonradan geliştirilmiştir.

## **SORU-13: Ön Hafıza (Cache) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde sıklıkla kullanılan ve herhangi bir hafıza işleminin nispeten daha küçük ve daha hızlı dolayısıyla da daha pahalı ilave bir hafızada yapılmasını ifade eden terimdir.

Aslında kelime anlamı olarak Türkçedeki zula kelimesi ile de karşılanabilen cache kelimesi, değerli şeylerin saklandığı yer anlamına gelmektedir. Bilgisayar bilimlerinde ise sık erişilen ve dolayısıyla bizim için daha değerli olan bilgilerin saklandığı yere verilen isimdir.

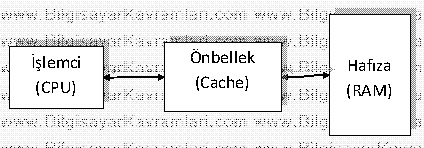
Kısacası bir ön hafıza (cache) gerçek hafızaya erişmeyi azaltmak ve daha hızlı bir şekilde çözmek için tasarlanır. Gerçek hafızadan daha hızlı ve daha pahalıdır. Gerçek hafızadaki her şeyi içeremez dolayısıyla küçük bir kısmını içerir.

En çok kullanıldığı yerler aşağıdaki şekilde sıralanabilir:

* İşlemci önbelleği
* Disk önbelleği
* Web önbelleği

Elbette yukarıdakiler dışında farklı alanlarda da veriye erişim hızını arttırmak için önbellekler kullanılabilir.

Örneğin aşağıda, RAM (hafıza) ile işlemci (CPU) arasına yerleştirilmiş ve işlemci ön belleği (CPU Cache) ismi verilen erişim şekli gösterilmiştir:



Buradaki amaç, işlemcinin hafızada ihtiyaç duyduğu verilerin bir kısmını önbellekten karşılamasıdır. Benzer şekilde hafızadaki veriler ile disk arasında da önbellek olabilir. Bu durumda bir verinin diskten ihtiyaç duyulması halinde diskten daha hızlı olan ön bellek devreye girerek hafızaya yükleme işlemi gerçekleştirebilir. Elbette burada iki ihtimal vardır:

Hit: isabet, işlemcinin isteğinin önbellekten karşılanması

Miss: kayıp, işlemcinin isteğinin önbellekten karşılanamayarak hafızaya erişim yapılması

Bu sayıların oranlanması ile de aşağıdaki değerler bulunur:

İsabet oranı (hit ratio) = isabet (hit) / Toplam talep (veya isabet + kayıp)

Kayıp oranı (miss ratio) = kayıp (miss) / Toplam talep (veya isabet + kayıp)

Örneğin 45 hafıza erişim talebinin olduğunu (request) ve bunların 21′inin önbellekten karşılandığını düşünelim. Bu durumda

Kayıp (miss) = 45 – 21 = 24

İsabet oranı (hit ratio) = 21 / 45 = %47

Kayıp oranı (miss ratio) = 23 / 45 = %53

Olarak hesaplanabilir.

**Önbellek yazma politikası (write policy)**

Önbelleklerin birinci görevi, ihtiyaç duyulan verinin hızlı karşılanmasıdır. Ancak bu durum veri okunurken avantaj sağlar. Verinin değiştirilmesi veya yazılması gibi durumlarda önbellekte nasıl bir politika izleneceğine de yazma politikası (write policy) ismi verilir. Buna göre 3 farklı politika izlenebilir:

Üzerine yazma (write through) : her yazma işlemi, önbellekte bir isabet olsa bile hafızada güncelleme gerektirir. Buna göre önbellekteki veri ile hafızadaki veri birebir yanı olur. Birisindeki değişiklik diğerini etkiler ve verinin iki ayrı kopyası arasında bir eşleşme gerektirmez.

Geri yazma (write back): Bu yazma politikasında, bir bilginin değiştirilmesi durumunda, bilgi ön bellekte bulunuyorsa, önbellek üzerinde değişiklik yapılır ve hafızadaki kopya hemen değiştirilmez. Dolayısıyla anlık olarak bilginin iki farklı kopyası bulunur. Bilginin iki kopyasının birbirinden farklı olduğunu göstermek için de kirli (dirty) bit kullanılır. Dolayısıyla bu politikada, ön bellek üzerinde ilave bir bite ihtiyaç vardır.

Bu yazma politikasında veriler ön bellekten kaldırılırken hafızadaki veri ile güncellenir. Yani ön bellek sürekli olarak değiştirilmiş veriyi tutar, veri ön bellekten kaldırılıp yerine yeni veri geleceği zaman, bu veri hafızadaki verinin üzerine yazılarak iki kopya aynı hale getirilir.

Bu yazma politikasının hafızayı değiştirmesindeki tutumundan dolayı tembel yazma (lazy write) ismi de verilir.

Bu yazma politikasında ayrıca herhangi bir verinin önbellekten kaldırılıp yerine yeni veri gelmesi durumunda iki kere hafıza erişimi gerekir. Birincisi eski verinin hafıza ile güncellenmesi ikincisi ise yeni verinin hafızaya yüklenmesi için. Üzerine yazma politikasında ise sadece tek erişim yeterlidir.

Yazmama politikası (no-write allocation): bu politikada, önbellek üzerinde bir yazma işlemi yapılmaz. Veri önbellekte sadece okunmak için tutulur ve bir yazma işlemi gerçekleştiğinde bu işlem doğrudan hafızaya yazılır.

Bu durumda önbellekteki verinin doğruluğunda problem olacaktır. Yani hafızadaki veri daha güncel ve önbellekteki veri eski kalmış olacaktır. Bunu belirtmek için önbellek üzerinde bir doğruluk biti (valid bit) kullanılır. Bu bit, verinin değiştiğini ve önbellekteki verinin eski kaldığını belirtir. Herhangi bir şekilde bu değişen veriye okumak için erişim olmazsa önbellekte güncellemeye gerek de kalmaz (örneğin bu veri bir daha hiç erişilmeden önbellekten kaldırılabilir) ancak bir şekilde bu veriye tekrar okumak için erişim olursa bu durumda hafızadaki veri ile güncellenmesi için hafızadan verinin okunup önbelleğe yeni halinin yazılması gerekir.

Bu politikada bir veri değiştirildiğinde ardından gelen yazma işlemlerinin önbellek tarafından karşılanması gerekmez. Ancak bir yazma işleminden sonraki okuma işlemlerinin hafızadan yeni veriyi önbelleğe yüklemesi gerekir.

**Önbellek ilişkilendirmesi (Cache Associativity)**

Ön bellek kullanılırken bir verinin önbellek üzerinde nereye yazılacağını belirleyen yöntemlere verilen isimdir. Bu yöntemler temel olarak üç grupta toplanabilir.

* Doğrudan haritalama (direct mapped cache) yönteminde veri önbellek üzerinde tek bir yere yazılabilir.
* Küme ilişkilendirme (set associativity) yönteminde veri önbellek üzerinde birden fazla yere yazılabilir.
* Tam ilişkilendirme (full associativity) yönteminde ise veri önbellekteki her yere yazılabilir.

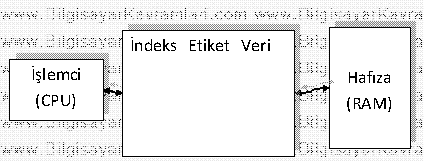
Yukarıdaki bu ilişkilendirme yöntemlerini aşağıda daha detaylı açıklamaya çalışalım.

Doğrudan haritalama (direct mapped cache)

Bu yöntemde, hafızada tutulan veriler ile önbellek adresleri arasında tam bir bağlantı vardır. Örneğin 4 satırdan oluşan bir önbelleğimiz olsun (yani önbelleğimizin kapasitesi 4 veri tutmaya yetiyor). Bu durumda önbelleğin adres için ayrılan bit sayısı 2 olur (22 = 4)

Yine örnek olarak hafızadaki verilerin uzunluğu da 8 bit olsun.

En basit doğrudan haritalama için mod alma işlemi kullanılabilir. Yani örneğin adresin son iki bitine göre (yada ilk iki bitine göre) hafızada erişilmek istenen adresi önbellekte adresleyeceğiz.



Örneğin yukarıdaki şekilde görüldüğü üzere ön bellek 3 sütundan oluşmuştur. Bu sütunların ilki önbellek satır numarasını belirtmeye yarayan ve bir bilginin önbellekte olması durumunda hangi satırda tutulduğunu gösteren sütundur.

Etiket kısmı, önbellekteki verinin aslında hafızadaki gerçek adresini belirten sütundur.

Veri sütunu ise önbellekte duran veridir. Bilginin önbellek tarafından karşılanması durumunda buradaki veri sütununda bulunan veri okunur.

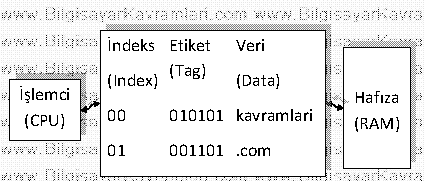
Yukarıda anlatıldığı üzere kullanılan yazma politikasına göre önbellekte ilave sütunlar bulunabilir (kirli (dirty) veya doğru (valid) bitleri gibi).

Örneğin adres olarak tutulan veri 8 bit uzunluğunda olsun ve önbellekte bu verinin ilk iki bitine göre indeksleme yapılıyor olsun.

Örnek hafıza erişimlerini aşağıdaki şekilde ele alalım:

|  |  |
| --- | --- |
| Hafıza adresi | Veri |
| 11010100 | www. |
| 10101001 | bilgisayar |
| 00010101 | kavramlari |
| 01001101 | .com |

Bu verilerin ilk iki bitleri birbirinden farklıdır ve hepsi önbellekte farklı sıralara yerleştirilecektir. Bu verilerin yerleştirilmiş hali aşağıdadır:



Yukarıda görüldüğü üzere, veriler önbellek üzerinde doğru yerlere yerleştirilmiş ve etiket kısmında adresi tamamlayıcı bilgiler bulunmaktadır. Bu yerleştirme işleminden sonra herhangi bir veriye erişilmek istendiğinde önbellek şu şekilde çalışır:

Örneğin erişilecek hafıza adresi 01001101 olsun.

Bu durumda ilk iki bitine bakılacak ve 01 indeksinde bulunan etiket kontrol edilecektir.

Buradaki etiket değeri ulaşılmak istenen adresin son 6 biti ile aynı olduğu için ( 001101 = 001101 olduğu için) verinin önbellekte bulunduğuna karar verilip hafıza erişimi önbellekten karşılanacaktır ve veri olarak “.com” işlemciye iletilecektir.

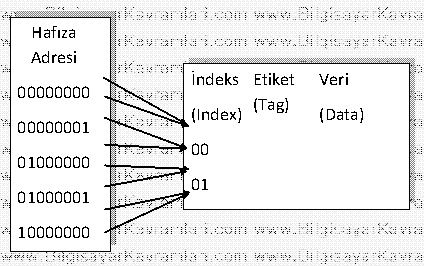
Örneğin erişilecek hafıza adresi 01001111 olsun.

Bu durumda ilk iki bitine bakılacak ve 01 indeksinde bulunan etiket kontrol edilecektir. Bu etiket ulaşılmak istenen hafıza adresi ile uyuşmamaktadır ( 001101 != 001111 olduğu için).

Dolayısıyla veri önbellekte olmadığı için verinin hafızadan karşılanması gerekir. Hafızadan karşılanan bu veri, önbellek değiştirme algoritmasına göre (aşağıda anlatılacaktır) önbellekte bir değiştirmeye sebep olur veya olmaz ancak her halükarda bir hafıza erişimi gerekecektir.

Örneğin hafıza erişiminin ardından verinin değiştirileceğini düşünelim. Bu durumda önbellekteki 01 indeksinde olan veri değiştirilecek ve hafızadan yüklenen veri bu indekse yazılacaktır. Doğrudan erişim algoritmasında verinin yazılabileceği tek adres bulunur.

Doğrudan haritalama yöntemini örnek bir şekil üzerinden gösterecek olursak:



Yukarıda görülen şekilde hafızada, örnek adresler verilmiştir. Elbette 8 bitlik hafızada daha fazla adres vardır ancak yukarıdaki temsili şeklin amacı durumu anlatmak olduğu için ilk iki biti aynı olan ikişer adres üzerinden örnek gösterilmiştir.

Yukarıda görüldüğü üzere hafızadaki adresler doğrudan önbellek üzerinde tek bir indekse haritalanmış ve birden fazla hafıza adresi aynı indekse haritalanmış durumdadır.

Küme ilişkilendirme (Set Associativity)

Bu erişim yönteminde bir önceki doğrudan erişimden farklı olarak veri kesin ve net bir şekilde bir yere eklenmez. Bunun yerine bir bilginin önbellek üzerinde gidebileceği birden fazla yer bulunur.

Bu küme ilişkilendirme yönteminde bir bilginin önbellek üzerinde gidebileceği alana göre sayılar belirtilir. Örneğin ön bellek üzerinde 2 farklı yere eklenebiliyorsa 2 yönlü küme ilişkilendirme (2-way set associativity) veya 4 farklı yere erişilebilirse dört yönlü küme ilişkilendirme (4-way set associativity) ismi verilir. Bu durumu aşağıdaki örnek üzerinden anlamaya çalışalım.

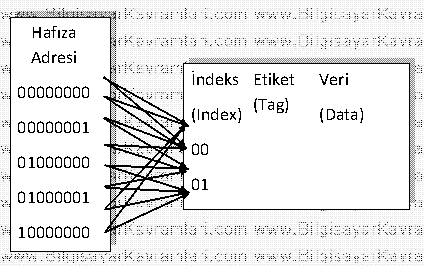
Örneğimizde bir önceki örnekle aynı yapıyı kullanalım ancak farklı olarak veri, hafıza adresinin ilk iki biti ve ilk iki biti +1 indekslerine yazılabilsin.

Yani bir bilgi aşağıdaki iki adresten birisinde indekslenecektir:

A : ilk iki bit değerindeki indeks

B: (ilk iki bit + 1) mod önbellek boyutu değerindeki indeks.

Bu durumda erişim aşağıdaki şekilde haritalanmış olacaktır:



Yukarıda görüldüğü üzere ilk iki bitine göre veri hem bu adrese hem de bu adresin bir sonraki adresine haritalanmıştır. Bu durumu aşağıdaki tablo ile de gösterebiliriz:

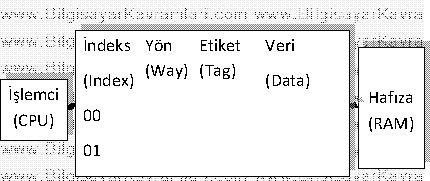
|  |  |  |
| --- | --- | --- |
| Hafıza adresi | Önbellek 1 | Önbellek 2 |
| 00 | 00 | 01 |
| 01 | 01 | 10 |
| 10 | 10 | 11 |
| 11 | 11 | 00 |

Yukarıdaki tabloda da gösterildiği üzere bir hafıza adresi iki farklı önbellek adresi ile haritalanmıştır.

Bu adresleme için örnek veri erişim tablomuz aşağıdaki şekilde olsun:

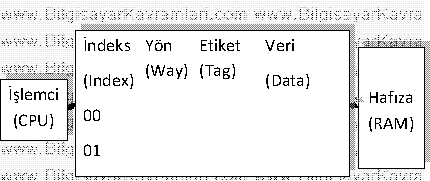
|  |  |
| --- | --- |
| Hafıza adresi | Veri |
| 11010100 | www. |
| 10101001 | bilgisayar |
| 10010101 | kavramlari |
| 01001101 | .com |
| 11010101 | Sadi |
| 00101111 | Evren |

Yukarıda verilen sırayla erişimi aşağıdaki şekiller üzerinden açıklamaya çalışalım:

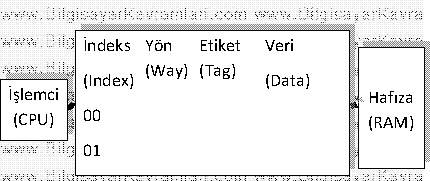


Yukarıdaki şekilde, daha önceki doğrudan haritalamadan farklı olarak yön (way) bitleri tutulmuştur. Bu bitlerin amacı bu indekste bulunan verinin aslında hangi yönden geldiğini tutmaktır. Örneğin 11 indeksine, 11 veya 10 yönünden veri yazılabilir bu durumda verinin hafıza adresini tam olarak bulabilmek için hangi yönden verinin yazıldığı tutulmalıdır.

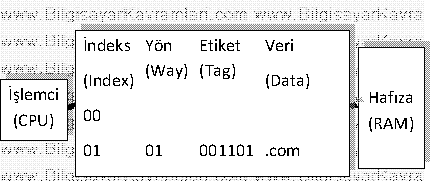
İlk olarak 11 indeksine veriyi yerleştiriyoruz. Ardından gelen 10101001 adresindeki veri ilk iki biti itibariyle 10 adresine yerleştiriliyor.



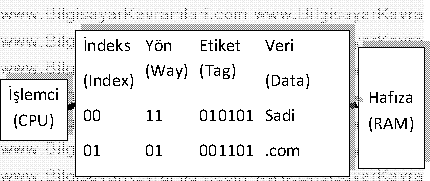
Sonraki verimiz 10010101 adresinde. Bu verinin yerleştirileceği ilk adres, 10 indeksi dolu, 2 yönlü haritalama kullandığımız için bir sonraki alternatif adrese bakıyoruz ve 11 adresi kontrol ediliyor. Bu adres de dolu olarak görülüyor. Sonuçta önbellek üzerinde yerleştirilebilir bütün alanlar dolu olduğu için önbellek değiştirme algoritması devreye giriyor ve önbellekteki bilgilerden birisinin değiştirilmesi gerekiyor. Örneğin kullanacağımız değiştirme algoritması FIFO (ilk giren ilk çıkar (first in first out)) olsun. Bu durumda 10 ve 11 adreslerinden eski olanı ile yeni gelen veri değiştirilecektir. Şu anda eski olan bilgi 11 adresindeki bilgidir dolayısıyla önbelleğin yeni hali aşağıdaki şekilde olacaktır:



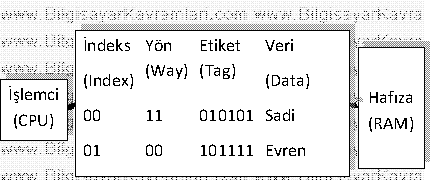
Yukarıdaki önbellek değiştirme (cache replacement) işleminin ardından 01001101 adresine erişim yapılıyor. Bu adres boş olduğu için ön bellek burada bilgiyi tutabilir:



Ardından gelen 11010101 erişimi ise yine önbellekte dolu olan indekse yapılmaktadır. Ancak 2 yönlü kümeleme kullanıldığı için veriyi diğer alternatifi olan ve şu anda boş olan 00 indeksine koyabiliriz:



Sonraki erişim 00101111 erişimidir. Burada yine FIFO önbellek değiştirme algoritması (cache replacement algorithm) gereği 00 ve 01 indekslerinden önce girenin değiştirilmesi gerekir. Bu bilgilerden 01 indeksindeki veri daha önce önbelleğe alındığı için veri buraya yerleştirilir:



Yukarıdaki önbellek yerleştirilmesinden sonra herhangi bir bilgiye erişilmek istendiğinde ilk iki bitine bakılır, ardından bu iki bitin girebileceği indeksler taranır.

Örneğin erişilmek istenen veri 11010101 adresinde olsun. Bu durumda ilk iki bite bakılıp 11 değeri ile arama yapılacak. 11 değeri ise 2 yönlü küme ilişkilendirmesi gereği 11 ve 00 adreslerinde tutulabilecek bilgidir. Bu durumda önbellekte iki yere de bakılacak 00 indeksindeki yön + etiket bilgisi aranan adresin bu indekste olduğunu gösterecek ve veri buradan karşılanacaktır.

Küme ilişkilendirme yöntemlerinde, verinin birden fazla yere yazılabilmesi, veriye erişim sırasında bütün bu alanların aranmasını gerektirir. Dolayısıyla önbellek üzerinde birden fazla erişime sebep olarak belki de önbellekte hiç tutulmayan bir veri için uzun süreli bir arama ile sonuçlanabilir.

Küme ilişkilendirme yöntemlerinin avantajı ise hiç erişilmeyen hafıza alanlarının önbellekte durmasını engellemesidir. Örneğin yukarıdaki şekil için, 00 ile başlayan hafıza alanına uzun süre hiç erişim olmayacağını düşünelim. Bu durumda doğrudan haritalama (Direct mapping) yöntemi bu önbellek alanını hiçbir zaman kullanmaz. Oysaki küme ilişkilendirme yönteminde alternatif bir önbellek indeksi bu alanı kullanabilir. Böylelikle önbelleğin daha verimli kullanılması sağlanır.

Bütün küme ilişkilendirme (full set associativity)

Bu yaklaşımda önbellekteki her indekste her adres durabilir. Yani bir önceki örnekte iki yönlü küme ilişkilendirme sırasında bir hafıza adresi, sadece iki indeksten birisine gidebilirken şimdi bütün önbelleğe yerleştirilebilir. Bu durumda önbellekteki arama süresi uzarken, önbellekte atıl kalan indeks miktarı azaltılmış olunur.

**Önbellek değiştirme algoritmaları (Cache replacement algorithms)**

Bu algoritmaların amacı, önbellek üzerinde küme ilişkilendirilmesi kullanıldığında ve verinin önbellekte birden fazla alana yazılabileceği durumlarda nereye yazılacağını belirlemektir.

Temel olarak aşağıdaki algoritmalar en bilinenleri olarak sayılabilir:

FIFO , first in first out , ilk giren ilk çıkar

LRU, least recently used, en eski erişilen

LFU, least frequently used, en seyrek erişilen

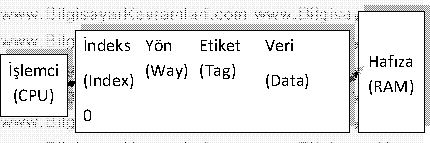
MRU, most recently used, en son erişilen

Belady’s, Belady’s algorithm, Belady algoritması

Yukarıdaki bu algoritmaları sırasıyla örnek üzerinden anlatmaya çalışalım. Örneğimizdeki önbelleğin iki indeksi bulunsun ve bütün küme ilişkilendirmesi (fully set associative) yapısında olsun. Buna göre aşağıdaki veriler önbelleğe geldiğinde sırasıyla yukarıdaki değiştirme algoritmalarının nasıl çalıştığını görelim:

|  |  |
| --- | --- |
| Hafıza adresi | Veri |
| 11010100 | www. |
| 10101001 | bilgisayar |
| 00010101 | kavramlari |
| 01001101 | .com |

Yukarıdaki verileri aşağıdaki önbellek yapısına sırasıyla yerleştireceğiz. 2 satırdan oluşan önbelleği indekslemek için tek bit yeterlidir. Bu durumda hafıza adresinin ilk bitini önbellek indekslemesi için kullanacağız.

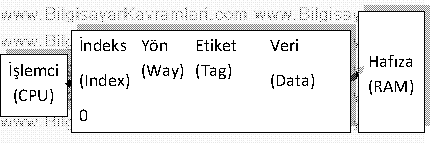


Bu durumda yukarıdaki ön bellek üzerinde bütün küme ilişkilendirme kullanılacaktır. Aslında önbellek iki satırdan oluştuğu için 2 yönlü küme ilişkilendirme de denilebilir.

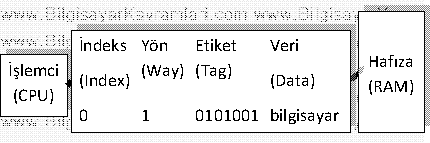
FIFO (First in first out, ilk giren ilk çalışır)

Bu algoritmada, önbellekte bir bilgi yazılacağı sırada, önbelleğe ilk girmiş, en eski verinin üzerine yazılması tercih edilir.

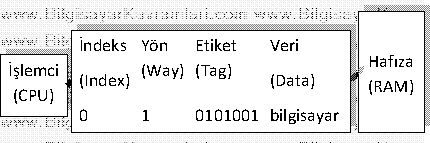
Yukarıdaki verilere erişimi sıra ile aşağıdaki şekiller üzerinden anlamaya çalışalım. İlk erişim 11010100 adresine yapılıyor.



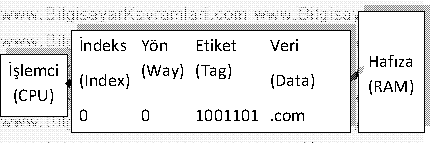
Bu adres 1 ile başladığı için 1. İndekse yerleştiriyoruz. Ardından gelen veri de 1 ile başladığı için 1. Adrese yerleşmesi gerekiyor ancak bu adres dolu ve 0. Adres boş, o halde boş olan yere yerleştiriliyor:



Şimdi gelen veri ise hem 0 hem de 1 e yerleşebilir (bütün küme ilişkilendirmesi olduğu için) dolayısıyla FIFO devreye giriyor ve en eski olan “www.” Bilgisi kaldırılıp yerine yazılıyor.



Sonraki veri yine iki indekse de yazılabilecek durumda ve veri en eski olan “bilgisayar”‘ın üzerine yazılıyor.



LRU (Least Recently Used, En Eski Erişilen)

Bu önbellek değiştirme algoritmasında amaç, önbellekte yapılan erişimleri takip etmek ve en geç erişilmiş olan veriyi değiştirmektir. Örneğimizi yine yukarıdaki gibi 2 satırlı bir önbellek üzerinden ve aşağıdaki erişim sırasıyla takip etmeye çalışalım:

1010 1010

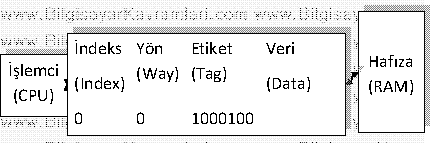
0100 0100

1010 1010

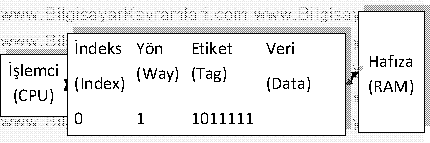
1101 1111

0100 0100

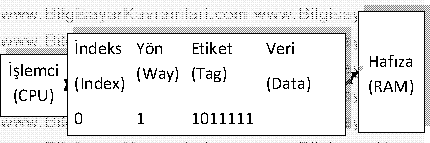
Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en eski erişilmiş olan verinin üzerine yazılır:



Şu anda en son erişilen veri yeni yazdığımız veridir ve yeni gelen veri diğer verinin üzerine yazılır:



Görüldüğü üzere bu algoritmada, her zaman için en eski erişilmiş olan verinin üzerine yeni gelen veri yazılmaktadır.

LFU (Least frequently used, en nadir kullanılan veya en seyrek kullanılan)

Bu algoritmada amaç ön bellekte bulunan verilere yapılan erişim miktarlarını saymak ve bu erişim miktarlarından en az olanını değiştirmektir.

Bu durumu yine bir önceki alt başlıkta incelediğimiz örnek üzerinden anlamaya çalışalım.

1010 1010

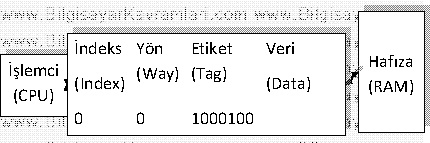
0100 0100

1010 1010

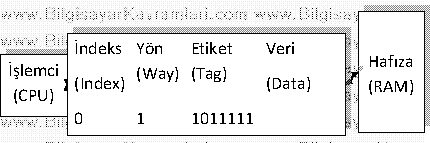
1101 1111

0100 0100

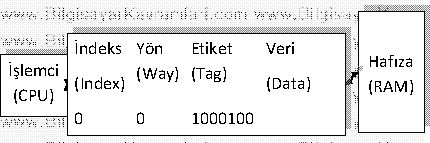
Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en az erişilmiş olan verinin üzerine yazılır. Burada en az erişilen veri 0. İndekste olan veridir.



Yeni yüklenen veri, şu andaki en az erişilmiş olan veridir. Dolayısıyla değiştirme işlemi yine 0. Satırdaki veri üzerine yapılır.



Görüldüğü üzere değiştirme işlemi sürekli olarak, o ana kadar en az erişilen önbellek alanında olmaktadır.

MRU (Most Recently Used, En sık kullanılan)

Bu değiştirme algoritmasında amaç en son erişilen veriyi değiştirmektir. Bu algoritma ilk başta çok anlamlı gelmeyebilir, sonuçta önbellekteki en taze bilgi en son erişilen bilgidir ve dolayısıyla işlemcinin bir sonraki adımda erişme ihtimali yüksek olan veridir.

Ancak bazı durumlarda anlamlı olabilir. Örneğin bir dosyadan sürekli olarak okuma yapıldığını veya bir sıkıştırma algoritmasının büyük bir dosyayı açmak için uğraştığını dolayısıyla hafızada yüklü bu dosyanın sürekli işlemci üzerinde işlendiğini ve işlenen bir veriye bir daha geri dönülmeyeceğini düşünün.

Örneğin 100MB boyutundaki sıkıştırılmış bir dosyayı açmak istiyoruz. Bu durumda dosyadan okunan ve işlenen bir bilgi kısmı açılacak ve işlemcinin dosyanın bu kısmı ile olan işi bitecektir.

Bundan sonraki adımlarda önbelleğe yüklenen verilerin tamamı taze olacak ve daha önceden yüklenen bir veriye ihtiyaç duyulmayacaktır.

Bu durumda önbelleğin bir daha kullanılmayacak veriler ile işgal edilmesi yerine önbellek üzerinde en son erişilen verinin değiştirilmesi ve diğer işlemlerin önbellek üzerinde kullandıkları verilerin daha sonraki erişimler için saklanması mantıklı olur.

Bu algoritmanın çalışmasını yine bir örnek üzerinden inceleyelim.

1010 1010

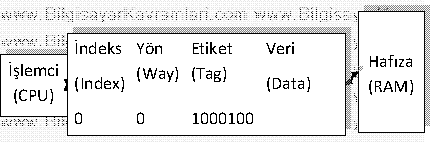
0100 0100

1010 1010

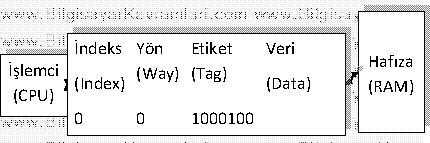
1101 1111

0100 0100

Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Buradan anlaşılacağı üzere en son erişilen veri ikinci satırdaki veridir. Yeni gelen veri en son erişilmiş olan verinin üzerine yazılır. Burada en az erişilen veri 1. İndekste olan veridir.



Yeni yüklenen veri, şu andaki en son erişilmiş olan veridir. Gelen erişim talebi ise zaten önbellekte bulunan veridir ve herhangi bir değiştirme işlemi gerekmeden çalışma sonuçlanır.

Belady’s Algorithm (Belady algoritması)

Teorik olarak tanımlı olan bu algoritmanın gerçekte uygulanması mümkün değildir. Bu algoritma temel olarak ileride en çok erişilecek hafıza adreslerini önbellekte tutmayı hedefler. Elbette gerçekte bir çalışma olmadan ileride neyin çalışacağı bilinemeyeceği için de bu algoritmanın kullanılması imkansızdır.

Yine bir örnek üzerinden algoritmanın çalışmasını inceleyelim:

1010 1010

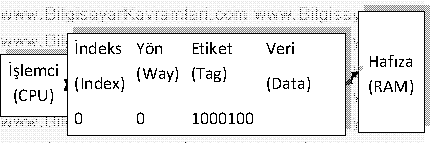
0100 0100

1010 1010

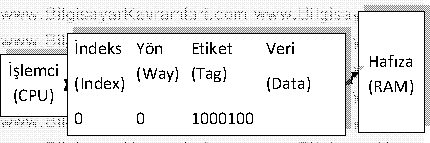
1101 1111

0100 0100

Yukarıdaki bu erişim sıralarına göre önbellek üzerindeki yerleşimler aşağıdaki şekilde olacaktır. İlk gelen iki veri sırasıyla önbelleğe yerleştirilecektir, buraya kadar herhangi bir değiştirme algoritmasına ihtiyaç duyulmaz.



Ardından 1. İndekste bulunan veriye ikinci kere erişilir. Daha sonra gelen erişimler incelendiğinde sıradaki erişim olan 11011111, şu anda 1. İndekste olan değer ile değiştirilmelidir çünkü 0100 0100 adresine ileride erişim olacaktır ve bu bilgi ile değiştirilmesi durumunda ileride yine bir önbellek değiştirme işlemi gerekecektir.



Görüldüğü üzere ileride yapılacak değişiklik önceden hesaplanmış ve bunu engellemek için en makul değişiklik yapılmıştır.

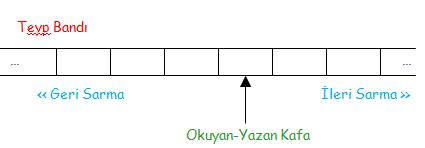
Sonuçta bu algoritma ile her zaman en az önbellek kayıp oranı (cache miss) yakalanır.

## **SORU-14: Turing Makinesi (Turing Machine) hakkında bilgi veriniz.**

Bilgisayar bilimlerinin önemli bir kısmını oluşturan otomatlar (Automata) ve [Algoritma Analizi (Algorithm analysis)](http://www.bilgisayarkavramlari.com/category/algoritma-analizi-teory-of-algorithms/) çalıştırmalarının altındaki dil bilimin en temel taşlarından birisidir.1936 yılında Alan Turing tarafından ortaya atılan makine tasarımı günümüzde pekçok teori ve standardın belirlenmesinde önemli rol oynar.

**Turing Makinesinin Tanımı**

Basitçe bir kafadan (head) ve bir de teyp bandından (tape) oluşan bir makinedir.



Makinede yapılabilecek işlemler

* Yazmak
* Okumak
* Bandı ileri sarmak
* Bandı geri sarmak

şeklinde sıralanabilir.

**Chomsky hiyerarşisi ve Turing Makinesi**

Bütün teori bu basit dört işlem üzerine kurulmuştur ve sadece yukarıdaki bu işlemleri kullanarak bir işin yapılıp yapılamayacağı veya bir dilin bu basit 4 işleme indirgenip indirgenemeyeceğine göre diller ve işlemler tasnif edilmiştir.



Bu sınıflandırma yukarıdaki venn şeması ile gösterilmiştir. Aynı zamanda [chomsky hiyerarşisi (chomsky hierarchy)](http://www.bilgisayarkavramlari.com/2009/06/27/chomsky-hiyerarsisi-chomsky-hierarchy/) için 1. seviye (type-1) olan ve Turing makinesi ile kabul edilebilen diller bütün tip-2 ve tip-3 dilleri yani içerk bağımsız dilleri ve düzenli dilleri kapsamaktadır. Ayrıca ilave olarak içerik bağımsız dillerin işleyemediği (üretemediği veya parçalayamadığı (parse) ) anbncn şeklindeki kelimeleri de işleyebilmektedir.  Düzenli ifadelerin işleyememesi konusunda bilgi için [düzenli ifadelerde pompalama savı (pumping lemma in regular expressions)](http://www.bilgisayarkavramlari.com/2009/03/22/duzenli-ifadelerde-pompalama-onsavi-pumping-lemma-for-regular-expressions/) ve [içerik bağımsız dillerin işlemeyemesi için de içerik bağımsız dillerde pompalama savı (pumping lemma for CFG)](http://www.bilgisayarkavramlari.com/2009/03/22/icerik-bagimsiz-gramerler-icin-pompalama-onsavi-pumping-lemma-for-context-free-grammers/) başlıklı yazıları okuyabilirsiniz.

**Turing Makinesinin Akademik Tanımı**

Turing makineleri literatürde akademik olarak aşağıdaki şekilde tanımlanır:

turing_akademik

Burada M ile gösterilen makinenin parçaları aşağıda listelenmiştir:

Q sembolü sonlu sayıdaki durumların kümesidir. Yani makinenin işleme sırasında aldığı durumlardır.

Γ sembolü dilde bulunan bütün harfleri içeren alfabeyi gösterir. Örneğin ikilik tabandaki sayılar ile işlem yapılıyorsa {0,1} şeklinde kabul edilir.

Σ sembolü ile makineye verilecek girdiler (input) [kümesi](http://www.bilgisayarkavramlari.com/2009/06/24/kume-teorisi-set-theory/) gösterilir. Girdi [kümesi](http://www.bilgisayarkavramlari.com/2009/06/24/kume-teorisi-set-theory/) dildeki harfler dışında bir sembol taşıyamayacağı için Σ ⊆ Γ demek doğru olur.

δ sembolü dilde bulunan ve makinenin çalışması sırasında kullanacağı geçişleri (transitions) tutmaktadır.

◊ sembolü teyp bandı üzerindeki boşlukları ifade etmektedir. Yani teyp üzerinde hiçbir bilgi yokken bu sembol okunur.

q0 sembolü makinenin başlangıç durumunu (state) tutmaktadır ve dolayısıyla q0 ⊆ Q olmak zorundadır.

F sembolü makinenin bitiş durumunu (state) tutmaktadır ve yine F ⊆ Q olmak zorundadır.

**Örnek Turing Makinesi**

Yukarıdaki sembolleri kullanarak örnek bir Turing makinesini aşağıdaki şekilde inşa edebiliriz.

Örneğin basit bir kelime olan a\* düzenli ifadesini (regular expression) Turing makinesi ile gösterelim ve bize verilen aaa şeklindeki 3 a yı makinemizin kabul edip etmediğine bakalım.

Tanım itibariyle makinemizi aşağıdaki şekilde tanımlayalım:

M = { {q0,q1} , { a } , { a,x } , { q0 a→a R q0 , q0 x→x L q1} , q0 , x , q1 }

Yukarıdaki bu makineyi yorumlayacak olursak:

Q değeri olarak   {q0,q1} verilmiştir. Yani makinemizin ik idurumu olacaktır.

Γ değeri olarak { a,x } verilmiştir. Yani makinemizdeki kullanılan semboller a ve x’ten ibarettir.

Σ değeri olara {a} verilmiştir. Yani makinemize sadece a girdisi kabul edilmektedir.

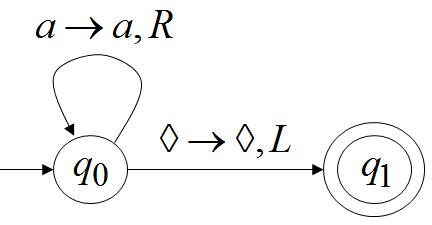
δ değeri olarak iki geçiş verilmiştir { q0 a→a R q0 , q0 x→x L q1} buraadki R sağa sarma L ise sola sarmadır ve görüleceği üzere Q değerindeki durumlar arasındaki geçişleri tutmaktadır.

◊ değeri olarak x sembolü verilmiştir. Buradan x sembolünün aslında boş sembolü olduğu ve bantta hiçbir değer yokken okunan değer olduğu anlaşılmaktadır.

q0 ile makinenin başlangıç durumundaki hali belirtilmiştir.

F değeri olarak q1 değeri verilmiştir. Demek ki makinemiz q1 durumuna geldiğinde bitmektedir (halt) ve bu duruma gelmesi halinde bu duruma kadar olan girdileri kabul etmiş olur.

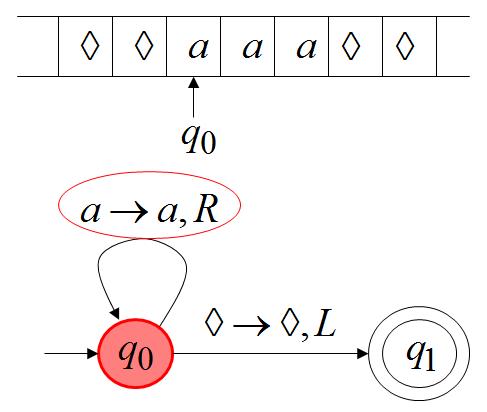
Yukarıdaki bu tanımı görsel olarak göstermek de mümkündür:



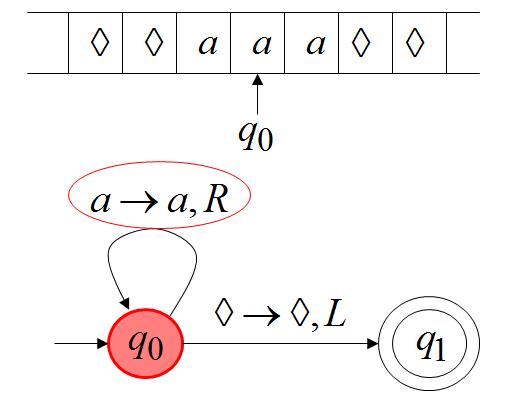
Yukarıdaki bu temsili resimde verilen turing makinesi çizilmiştir.

Makinemizin örnek çalışmasını ve bant durumunu adım adım inceleyelim.

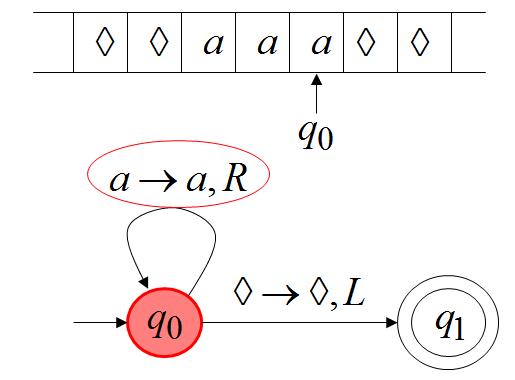
Birinci adımda bandımızda aaa (3 adet a) yazılı olduğunu kabul edelim ve makinemizin bu aaa değerini kabul edip etmeyeceğini adım adım görelim. Zaten istediğimiz de aaa değerini kabul eden bir makine yapabilmekti.



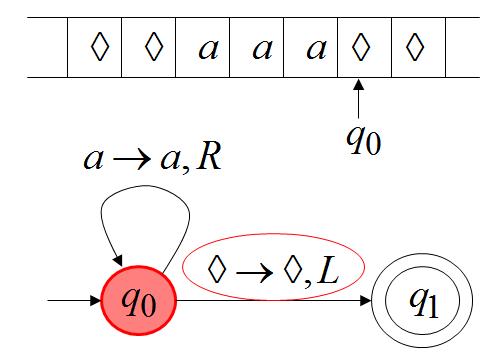
Yukarıdaki ilk durumda bant üzerinde beklenen ve kabul edilip edilmeyeceği merak edilen değerimiz bulunuyor. Makinemizin kafasının okuduğu değer a sembolü. Makinemizin geçiş tasarımına göre q0 halinde başlıyoruz ve a geldiğinde teybi sağa sarıp yine q0 durumunda kalmamız gerekiyor.



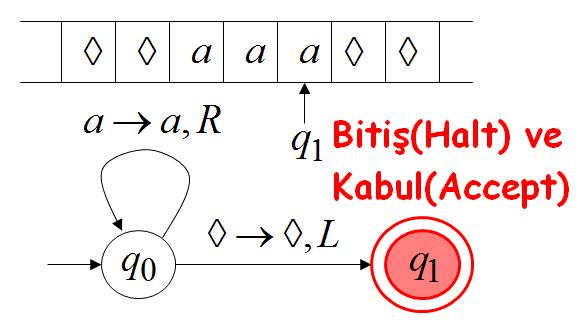
Yeni durumda kafamızın okuduğu değer banttaki 2. a harfi ve bu durumda yine q0 durumundayken teybi sağa sarıp yine q0 durumunda kalmamız tasarlanmış



3. durumda kafamızın okuduğu değer yine a sembolü olmakta ve daha önceki 2 duruma benzer şekilde q0 durumundayken a sembolü okumanın sonucu olarak teybi sağa sarıp q0 durumunda sabit kalıyoruz.



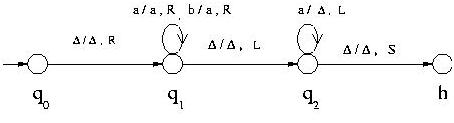
4. adımda teypten okuduğumuz değer boşluk sembolü x oluyor. Bu değer makinemizin tasarımında q1 durumuna gitmemiz olarak tasarlanmış ve teybe sola sarma emri veriyoruz.



Makinenin son durumunda q1 durumu makinenin kabul ve bitiş durumu olarak tasarlanmıştı ( makinenin tasarımındaki F kümesi) dolayısıyla çalışmamız burada sonlanmış ve giriş olarak aaa girdisini kabul etmiş oluyoruz.

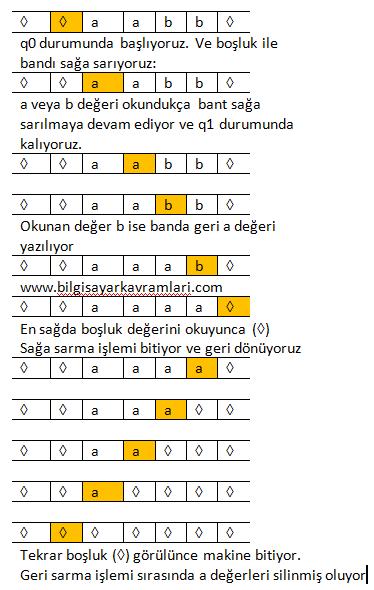
**2. Örnek**

Hasan Bey’in sorusu üzerine bir örnek makine daha ekleme ihtiyacı zuhur etti. Makinemiz {a,b} sembolleri için çalışsın ve ilk durum olarak bandın en solunda başlayarak bantta bulunan sembolleri silmek için tasarlansın. Bu tasarımı aşağıdaki temsili resimde görülen otomat ile yapabiliriz:



Görüldüğü üzere makinemizde 4 durum bulunuyor, bunlardan en sağda olan h durumu bitişi (halt) temsil ediyor. Şimdi bu makinenin bir misal olarak “aabb” yazılı bir bantta silme işlemini nasıl yaptığını adım adım izah etmeye çalışalım.

Aşağıda, makinenin her adımda nasıl davranacağı bant üzerinde gösterilmiş ve altında açıklanmıştır. Sarı renge boyalı olan kutular, kafanın o anda üzerinde durduğu bant konumunu temsil etmektedir.



Netice olarak Hasan Bey’in sorusuna temel teşkil eden ve örneğin q1 üzerindeki döngülerden birisi olan b/a,R geçişi, banttan b okunduğunda banta a değerini yaz manasındadır.

## **SORU-15: Atomluluk (Atomicity) hakkında bilgi veriniz.**

Latince bölünemez anlamına gelen atom kökünden üretilen bu kelime, bilgisayar bilimlerinde çeşitli alanlarda bir bilginin veya bir varlığın bölünemediğini ifade eder.

Örneğin programlama dillerinde bir dilin atomic (bölünemez) en küçük üyesi bu anlama gelmektedir. Mesela C dilinde her satır (statement) atomic (bölünemez) bir varlıktır.

Benzer şekilde bir verinin bölünemezliğini ifade etmek için de veri tabanı, veri güvenliği veya veri iletimi konularında kullanılabilir.

Örneğin veri tabanında bir işlemin (transaction) tamamlanmasının bölünemez olması gerekir. Yani basit bir örnekle bir para transferi bir hesabın değerinin artması ve diğer hesabın değerinin azalmasıdır (havale yapılan kaynak hesaptan havale yapılan hedef hesaba doğru paranın yer değiştirmesi) bu sıradaki işlemlerin bölünmeden tamamlanması (atomic olması) gerekir ve bir hesaptan para eksildikten sonra, diğer hesapa para eklenmeden araya başka işlem giremez.

Benzer şekilde işletim sistemi tasarımı, paralel programlama gibi konularda da bir işlemin atomic olması araya başka işlemlerin girmemesi anlamına gelir.

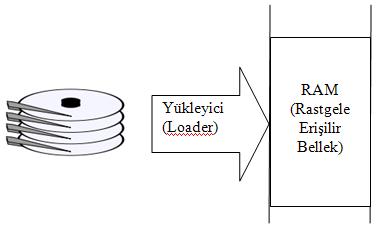
Örneğin sistem tasarımında kullanılan check and set fonksiyonu önce bir değişkeni kontrol edip sonra değerini değiştirmektedir. Bir değişkenin değeri kontrol edildikten sonra içerisine değer atanmadan farklı işlemler araya girerse bu sırada problem yaşanması mümkündür. Pekçok işlemci tasarımında buna benzer fonksiyonlar sunulmaktadır.

Genel olarak bölünemezlik (atomicity) geliştirilen ortamda daha düşük seviyeli kontroller ile sağlanır. Örneğin işletim sistemlerinde kullanılan [semafor’lar (semaphores)](http://www.bilgisayarkavramlari.com/2009/03/30/semafor-semaphore-flama-isaret/), kilitler (locks), koşullu değişkenler (conditional variables) ve monitörler (monitors) bunlar örnektir ve işletim sisteminde bir işlemin yapılması öncesinde bölünmezlik sağlayabilirler.

Kullanılan ortama göre farklı yöntemlerle benzer bölünmezlikler geliştirilebilir. Örneğin veritabanı programlama sırasında koşul (condition) veya kilit (lock) kullanımı bölünmezliği sağlayabilir.

## **SORU-16: Yükleyici (Loader) hakkında bilgi veriniz.**

Yükleyiciler basitçe bir programı diskten alıp hafızaya yüklemekle sorumlu programlardır.



Bir program yazıldıktan ve [derlendikten (compile)](http://www.bilgisayarkavramlari.com/2008/01/03/derleyici-compiler/) sonra programın makine dilindeki karşılığı elde edilir. Bu karşılık tam bir kod olmayıp harici kütüphanelerden faydalanıyor olabilir. Bu kütüphaneler de programa dahil edilip tam bir program elde edildikten sonra (yani [bağlandıktan sonra (linker)](http://www.bilgisayarkavramlari.com/2008/01/03/baglayici-linker/) ) program artık çalıştırılmaya hazırdır.

Programın çalışması ise programın CPU üzerinde yürütülmesi ile olabilir ve bunun için programın öncelikle [hafızaya (RAM)](http://www.bilgisayarkavramlari.com/2008/11/07/rastgele-erisilebilir-bellek-random-access-memory-ram/) yüklenmesi gerekir. Burada yükleyici (loader) devreye girer. Yükleyici makine dilindeki bu kodu alarak işletim sisteminin işaret ettiği adrese programı yükler. Buradan sonrası işletim sistemi tarafından yürütülür.

## **SORU-17: Konumlandırılabilir Kod (Relocatable Code) hakkında bilgi veriniz.**

Üretilecek olan makine kodunun [hafızada (RAM)](http://www.bilgisayarkavramlari.com/2008/11/07/rastgele-erisilebilir-bellek-random-access-memory-ram/) herhangi bir yere yerleştirilebilir olması durumudur. Buna göre normalde her programın sabit bir adres tanımı ve bu adres tanımına göre kod içerisinde adres değerlerini içeren komutların bulunması mümkündür. İşte konumlandırılabilir kod bu zorunluluğu kaldırarak hafızanın istenildiği gibi kullanılabilmesini sağlar ve kodun istenen hafıza adresin yüklenmesine izin verir. Dolayısıyla kodda bulunan adreslerin yeni yüklenen adreslere dönüşmesi gerekir.

http://www.bilgisayarkavramlari.com/wp-content/uploads/2008/12/dat.jpg

Yukarıdaki şekilde de tasvir edildiği üzere giren assembly dilindeki kaynak kodu makine dilinde bir kod üretirken bunun yanında Doğrudan Adres tablosu (DAT, Direct Address Table) ismi verilen bir tablo daha üretir. Bu tablo sayesinde koddaki adreslerin gerçek adreslere dönüşümü yapılabilmektedir.

## **SORU-18: 2 geçişli çeviriciler (2 pass assemblers) hakkında bilgi veriniz.**

Bir çeviricinin (assembler), Assembly dilinde yazılmış kaynak kod üzerinden iki kere geçen halidir. Buna göre tek geçişli olan çeviricilerden farklı olarak dilde etiket (label) tanımları yapılabilmekte ve bu etiketlerin anlamları bir tabloda tutularak daha sonra kullanılabilmektedir.

2 geçişli bir çevirici sırasıyla şu işlemleri yapar:

1. Geçişte koddaki semboller (symbols) ve ifadeler (literals) tanımlanarak hafızada saklanır

2. Geçişte hedef kod (makine dilinde) üretilir

Bu sırada kullanılan veri yapıları (data structures) aşağıda açıklanmıştır:

Konum Sayacı (Location Counter (LC)): Programın mevcut durumunu ve bundan sonra çalıştıracağı kodu tutar.

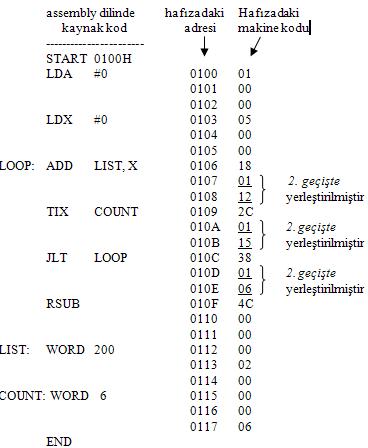
İşlem Kodu Tercüme Tablosu (Op-code translation table): Kaynak kodda bulunan komutların ve dile özgü işlemlerin çevirimlerini tutan tablodur.

Dizgi saklama belleği (String Storage Buffer (SSB)): Kaynak kodda geçen dizgilerin (kelime ve yazıların (strings)) ASCII kod karşılıklarının durduğu tablodur.

İşaret tablosu (Configuration Table): Dizgi saklama belleğinde saklanan dizgilerin üretilecek olan kodda tam olarak nereye tekabül ettiğini tutan tablodur. (bir dizginin (string) anlamı belirlenince bu tablo marifetiyle üretilecek koddaki yeri bulunarak buraya yazılacaktır)

Yukarıdaki şekilde iki geçişli bir çeviricinin ( two pass assembler) her iki aşaması tasvir edilmiştir.  Buna göre assembly dilindeki kaynak kodun üzerinden bir kere geçen çevirici (Assembler) dildeki sembolleri ve dizgileri (Strings) algılayarak geçici bir veri yapısında tutar. İkinci geçişte ise bu veri yapılarını da kullanarak makine dilini üretir.

Aşağıda örnek bir kod ve bu kodun çalışması anlatılmıştır:



Yukarıdaki temsili kod’un çalıştırılması sırasında oluşturulan tablolar aşağıda verilmiştir:

**Sembol Tablosu**

|  |  |
| --- | --- |
| ***Symbol*** | ***Address*** |
| LOOP | 0106 |
| LIST | 0112 |
| COUNT | 0115 |

Yukarıdaki tabloda, kodda bulunan her sembol için ( dilde tanımsız kelimeler için) birer kayıt tutulması ve bu kelimelerin geçtiği adreslerin tutulmasıdır.

**İşaret Tablosu:**

|  |  |
| --- | --- |
| ***Adresi (offset)*** | ***Sembolun SSB işaretçisi*** |
| 0007 | DC00 |
| 000A | DC05 |

İşaret tablosunda (configuration table) da kodumuzda karşılığı belirsiz olan sembollerin adresleri bulunur. Örneğin LIST kelimesi SSB içerisinde DC00 adresindedir. LIST kelimesinin işaret tablosuna yerleştirilme sebebi ise henüz tanımlanmadan kodun içerisinde kullanılmış olmasıdır.

**SSB:**

|  |  |  |
| --- | --- | --- |
| DC00 | 4CH | L |
| DC01 | 49H | I |
| DC02 | 53H | S |
| DC03 | 54H | T |
| DC04 | 5EH | Ayırma sembolü ASCII olarak |
| DC05 |  | C |

## **SORU-19: Kayan Nokta Sayıları (Floating Point Numbers) hakkında bilgi veriniz.**

Bilgisayar yapılarında ondalıklı sayıları ( floatingpoint numbers ) iki farklı bilginin tutulması ile gösterilebilir:

mantis x kök üst

yukarıda verilen formüle göre bir ondalıklı sayıyı önce bir ondalık çarpan sonra da bir kök’ün verilen üstü ile çarpımı olarak göstermek mümkündür.

Örneğin [ikilik tabanda](http://www.bilgisayarkavramlari.com/2007/11/24/bit-ikil/) 1101.11 küsurlu sayısını ele alalım (“.” işaretinden sonraki kısım küsurudu). Bu sayıyı göstermek için öncelikle bütün sayıyı önce 4 hane ilerleterek (kaydırarak, float) sayının tamamını küsurlu hale getirelim:

.110111

Yukarıdaki bu sayının orjinal değerini taşıyan gösterimi

.110111 x 2 4

olmalıdır çünkü orjinal sayının 4 hane kaydırılmış halidir. Dolayısıyla sayımızı aşağıdaki iki tamsayıyı tutarak göstermek mümkündür:

mantisa (mantissa) : 110111

üst (exponent) :  0100 (onluk tabandaki 4′ün karşılığı)

Merve Hanımın soruları üzerine aşağıdaki kısmı eklemenin gerekli olduğu anlaşılmıştır:

**Kayan noktaların gösterimi**

Konuyu örnek bir sayı üzerinden anlamaya çalışalım. Örneğin sayımız 123.321 olsun.

Sayımızın tam kısmı 123′tür küsurat kısmı aşağıdaki şekilde yazılabilir:

3\*10 -1 + 2\*10 -2 + 1\*10 -3

Bu düşünce, bizim alıştığımız onluk sistemdeki yaklaşımdır. Bu sayının ikilik tabana nasıl çevrileceği ile ilgili olarak [“ Ondalık sayıların taban dönüşümü”](http://www.bilgisayarkavramlari.com/2011/04/01/ondalikli-sayilarin-taban-donusumleri/) başlıklı yazıyı okuyabilirsiniz.

Ayrıca yukarıdaki sayımızı bilimsel gösterimde aşağıdaki şekilde yazabiliriz:

1.23321 \* 10 2

Tek hassasiyetli (single precision) kayan nokta sayısı, 32 bitlik bir paket olarak düşünülebilir. Bu yapı aşağıdaki tabloda temsil edilmiştir:

|  |  |  |
| --- | --- | --- |
| X | XXXXXXXX | XXXXXXXXXXXXXXXXXXXXXXX |
| **Yön**  **Sign** 1 bit | **Üst**  **Exponent** 8 bit | **Kök**  **Mantissa** 23 bit |

**Yön biti (sign bit)**

Bu bit 1 olduğu zaman, sayımız – değerdedir. Şayet 0 veya artı değerde bir sayı ise, bu bit 0 olur.

**Üst Biti (Exponent Bit)**

Yukarıdaki örnekte gösterilen bilimsel yazıma göre, sayının üst değeri bu alanda tutulur. Buna göre bir sayının 2′nin kaçıncı kuvveti ile çarpılabileceğini gösterir.

Burada dikkat edilecek bir husus, üst değerinin de eksi olabileceğidir. Örneğin sayımız 0.000123 ise, bu sayının bilimsel gösterimi : 1.23 \* 10 -4 olacaktır.

Bu durumda 8 bitlik alanın ilk biti yön bilgisi tutmaktadır. O halde geriye kalan 7 bit, üst bilgisi tutabilir. Buradan çıkan sonuç, sayımızın en fazla 127. üstüne kadar olan değerin tutulabileceğidir (2 7 =128 olduğunu ve bu bilginin ilk değerinin 0 için ayrıldığını dolayısıyla, sayımızın 0 – 127 arasında olacağını hatırlayınız).

**Kök biti (mantissa bit)**

23 bit alan kaplayan kısımdır. Bu alana bazı durumlarda belirgin anlamına gelen (significand) ismi de verilir. Bu kısım bilimsel gösterimin kök kısmıdır. Örneğin 1.23 \* 10 -4 sayısı için kök kısmı 123 olacaktır.

**Örnek sayı**

Yukarıdaki alanların, bir örnek sayı için nasıl gösterildiğine bakalım:

21,25 sayısını çevirmeye çalışalım. Öncelikle bu sayının ikilik tabandaki karşılığını bulalım:

10101.01

Bu sayıyı, bilimsel gösterime çevirelim:

21,25 = 2,125 \* 10 -1 = 2125 \* 10 -3

Ayrıca son bir bilgi : (2125) 10 = (1000010001101) 2

Şimdi bu sayının hafızadaki tutuluşuna bakalım:

Yön bitimiz, sayımız pozitif olduğu için 0 olacak.

Üst bitlerimiz, sayının üstü -3 olduğu için 100000011 olacak. Buradaki son iki bit 3 değerini, baştaki bit ise yönün eksi olduğunu göstermektedir.

Geriye kalan alan ise kök değerini tutacaktır.

(21) 10 = (10101) 2 olduğuna göre,sayımız aşağıdaki şekilde gösterilebilir:

|  |  |  |
| --- | --- | --- |
| 0 | 10000011 | 01010100000000000000000 |
| **Yön**  **Sign** 1 bit | **Üst**  **Exponent** 8 bit | **Kök**  **Mantissa** 23 bit |

Sonuç olarak sayımız:

01000001101010100000000000000000

şeklinde bulunmuş olunur.

**Çift hassasiyet ve tek hassasiyet ( double precision , single precision)**

Yukarıdaki yazı, tek hassasiyet dikkate alınarak hazırlanmıştır. Bunun anlamı sistemin 32 bit olmasıdır. Sistem 64 bit olabilir. Bu durumda çift hassasiyetten bahsedilmelidir. İki sistem arasındaki fark, aşağıdaki şekildedir:

1 bit yön biti (sign bit)  
8 bit üst değeridir (exponent)  
23 bit kök değeridir (mantissa)

64bitlik bir sistemde ise durum aşağıdaki şekildedir:  
1 bit yön bitidir (sign bit)  
11 bit üst değeridir (exponent)  
52 bit kök değeridir (mantissa)

**Kayan nokta gösteriminden Onluk sisteme çevirim**

Bu çevirimi anlamak için bir örnek üzerinden konuyu anlatalım.

Çevirmek istediğimiz sayı, 0xC0B40000 olsun. Bu sayı onaltılık tabandadır (hexadecimal) ve aşağıdaki şekilde ikilik tabana (binary) çevirilebilir:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Onaltılık | C | 0 | B | 4 | 0 | 0 | 0 | 0 |
| İkilik | 1100 | 0000 | 1011 | 0100 | 0000 | 0000 | 0000 | 0000 |

Yukarıdaki çevirimden çıkan ve tek hassasiyetli (single precision) gösterim aşağıda verilmiştir:

|  |  |  |
| --- | --- | --- |
| 1 | 10000001 | 01101000000000000000000 |

Demek ki sayımız eksi bir değermiş (en baştaki yön biti 1 olduğu için) ve ayrıca üstü de eksi imiş (yön bitini tutan grup 1 ile başladığı için).

Gelelim kök kısmının (mantissa) dönüşümüne.

Kök kısmımızdaki değer, 01101 olarak bulunuyor. Bu değeri yukarıdaki bilgiler ile birleştirirsek aslında sayımız aşağıdaki şekildedir:

-1.01101 \* 2 2

Artık problemimiz bu sayıyı onluk tabanda göstermektir. Sayının her basamağının üst değerini yazalım:

-(2 0 + 2 -2 + 2 -3 + 2 -5) \* 2 2 = -(2 2 + 2 0 + 2 -1 + 2 -3)

değerleri hesaplayalım:

-(4 + 1 + .5 + 0.125) = -5.625

## **SORU-20: Borulama (Pipelining) hakkında bilgi veriniz.**

ismini boru hatlarının işleyişinden alan yaklaşım, kısaca bir işlem borunun sonundayken, borunun başından yeni bir işin konulabileceğini anlatmaktadır.

Buna göre örneğin bir [işin (process)](http://www.bilgisayarkavramlari.com/2007/11/18/islem-process/) çalışması için 4 farklı safhadan geçmesi gerekiyor olsun:

Fetch (almak)

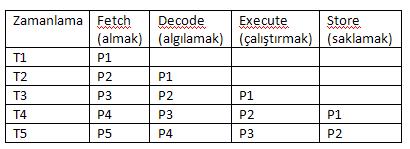
Decode (algılamak)

Execute (çalıştırmak)

Store (saklamak)

Bu dört safhanın her birisi her işlem için tekrarlanacaktır. Yani örneğin P1 isimli işlem sırasıyla Fetch, Decode, Execute ve store aşamalarından geçecektir.

Bu durum bize her aşamada ayrı bir işlemi çalıştırmak gibi bir avantaj doğurur. Aşağıdaki temsili resimde bu durum açıklanmıştır:



Yukarıdaki tabloda, zaman en solda gösterilmiştir. Yani anlık olarak T1 zamanından sonra T2 zamanı gelmektedir. Bu akışda göre en üstte olan ilk satırda sisteme P1 işlemi alınmaktadır. İkinci zamanda (T2) ise P1 işlemi decode aşamasına geçtiğinde P2 işlemi sisteme alınarak Fetch edilmektedir.

Bu işlem böylece devam etmekte en sonunda T5 zamanında P1 işi bitirilmiş, P2 işi Store edilmekte iken P5 işi işlenmeye başlanarak fetch edilmektedir.

Görüldüğü üzere yapılacak iş 4 aşamaya bölündüğünde aynı anda 4 farklı iş işlenmektedir. Bu işleme yöntemi sanki bir borudan akar gibi (örneğin T5 anında, P1 borudan (pipeline) çıkmış, P2 borunun bir ucunda P5 ise diğer ucunda gibi düşünülebilir) olduğu için bu sisteme borulama (pipelining) ismi verilmiştir.

Günümüz işlemci mimarilerinde de oldukça yaygın olarak kullanılan bu yaklaşım sayesinde işlem gücünün arttırılması mümkündür.

## **SORU-21: Çevirici (Assembler) hakkında bilgi veriniz.**

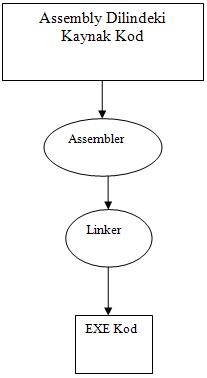
Bilgisayar bilimlerinde iki farklı kavram için assembler kelimesi kullanılmaktadır. Birincisi Assembly dili adı verilen ve makine diline (machine language) çok yakın düşük seviyeli (low level language) için kullanılan ve nesne kodunu (object code) makine koduna (machine code) çeviren dildir. İkincisi ise birleştirmek, monte etmek anlamında örneğin nesne yönelimli dillerde nesnelerin birleştirilmesi monte edilip büyük parçaların çıkarılması anlamında kullanılmaktadır.

Assembly dili, CPU üzerinde programcıya tanınmış olan yönergeleri (instructions) alarak bunları işlemcinin doğrudan çalıştırdığı makine kodları haline getirir. CPU üzerindekik bu yönergeler genelde üretici tarafından belirlenir ve tamamen teknoloji bağımlıdır.  Diğer bir deyişle aynı kod farklı yapıdaki işlemciler üzerinde çalışmaz. Hatta çoğu zaman aynı mimarinin farklı nesilleri arasında bile sorun yaşanır.

Assembly dili macro destekleyen ve desteklemeyen olarak ikiye ayrılır. Tarihi gelişim sürecinde ihtiyaç üzerine dilde macro yazmak ve yazılan bu macroları tekrar tekrar kullanmak imkanı doğmuştur. Bu tip assembly dilini makine diline çeviren çeviricilere de macroassembler adı verilir.

Assembly dilleri temel olarak fonksiyon(function) veya prosedür (procedure) desteği barındırmazlar. Hatta döngü ve koşul operatörleri de yok denilebilir. Bunun yerine programın içerisinde istenen bir satıra (adrese) gidilmesini sağlayan GOTO komudu kullanırlar.

[Yapısal programlama (structured programming)](http://www.bilgisayarkavramlari.com/2007/12/18/yapisal-programlama-structured-programming/) dokusuna ters olan bu yaklaşım dilin performans kaygısı ve düşük seviye olmasından kaynaklanmaktadır.



Yukarıdaki şekilde assembly kaynak kodunda verilen bir girişin assembler’dan geçerek bağlayıcı (linker) marifetiyle çalışabilir (Exe code) haline gelişi tasvir edilmiştir.

## **SORU-22: Rastgele Erişilebilir Bellek (Random Access Memory , RAM) hakkında bilgi veriniz.**

Bilgisayarların en önemli parçalarından birisidir. Özel bazı bilgisayarları dikkate alamazsak gündelik hayatta karşılaşılabilecek hemen her bilgisayarda bulunması gereken bir donanım parçasıdır. Birincil bellek (primary memory) ismi de verilen rasgele erişilebilir belleğin temel fonksiyonu işlemcinin (Merkezi işlem birimi ( Central processing unit (CPU)) program çalıştırırken geçici olarak verileri sakladığı ve sırası geldikçe bu verileri kullandığı alan olmasıdır.

Buna göre bir işlemci (CPU) çok basit toplama çıkarma seviyesinde işlemler yaparak programları çalıştırır. Komplike bir programın ise bu nevi basit işlemlere indirgenmesi mümkün olsa da bu indirgenme sonucunda çok sayıda işlemin yapılması gerekir. İşte CPU anlık olarak bu işlemlerden sadece birisini yapabilir. Geri kalan işlemler ise sırasını beklemek ve bu sırada bi yerde saklanmak zorundadır.

Kısaca bilgisayarda çalışan her programın CPU’da çalışmak için bekletildiği ve o ana kadar çalışması sonucunda biriken verilerinin saklandığı hafıza ünitesidir.

Disklere göre daha hızlı (ve dolayısıyla daha pahalı) olan veri ünitelerinin içinde bulunan bilgiler elektrik kesilmesi sonucu kaybolur. Bu yüzden ikincil hafıza (secondary memory) ismi verilen [sabit disklere (hard discs)](http://www.bilgisayarkavramlari.com/2007/10/23/disk-yonetimi-disk-management/) her zaman ihtiyaç vardır ve bilgisayarın kapanıp açılması sonucu verilerin korunması için ramdeki bilgiler diske yazılabilir.

Çok uzun süredir RAM’de bulunan verilerin elektrik kesintisinden sonra da hafızada kalması için üzerinde çalışılan ve çok uzun süredir bir ürün olarak alınabilen non-volatile (geçici olmayan) hafızalar ne yazık ki fiyatları yüzünden son kullanıcının alabileceği seviyelerde henüz değildir.

## **SORU-23: Sıralama Algoritmaları (Sorting Algorithms) hakkında bilgi veriniz.**

Bilgisayar bilimlerinde verilmiş olan bir grup sayının küçükten büyüğe (veya tersi) sıralanması işlemini yapan algoritmalara verilen isimdir. Örneğin aşağıdaki düzensiz sayıları ele alalım:

5 9 2 3 7 11 -4 6

Bu sayıların sıralanmış hali

-4 2 3 5 6 7 11

olacaktır. Bu sıralama işlemini yapmanın çok farklı yolları vardır ancak bilgisayar mühendisliğinin temel olarak üzerine oturduğu iki performans kriteri buradaki sonuçları değerlendirmede önemli rol oynar.

* Hafıza verimliliği (memory efficiency)
* Zaman verimliliği (Time efficiency)

Temel olarak algoritma analizindeki iki önemli kriter bunlardır. Bir algoritmanın hızlı çalışması demek daha çok hafızaya ihtiyaç duyması demektir. Tersi durumda da bir algoritmanın daha az yere ihtiyaç duyması daha yavaş çalışması demektir. Ancak bir algoritma hem zaman hem de hafıza olarak verimliyse bu durumda diğer algoritmalardan başarılı sayılabilir.

Genellikle verinin hafızada saklanması sırasında veriyi tutan bir berlirleyici özelliğinin olması istenir. Veritabanı teorisinde birincil anahtar (primary key) ismi de verilen bu özellik kullanılarak hafızada bulunan veriye erişilebilir. Bu erişme sırasında şayet berlileyici alan sıralı ise erişimin logaritmik zamanda olması mümkündür. Şayet veri sıralı değilse erişim süresi doğrusal (linear) zamanda olmaktadır.

Aşağıda bazı sıralama algoritmaları verilmiştir:

* [Seçerek Sıralama (Selection Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/secerek-siralama-selection-sort/)
* [Hızlı Sıralama Algoritması (Quick Sort Algorithm)](http://www.bilgisayarkavramlari.com/2008/08/09/hizli-siralama-algoritmasi-quick-sort-algorithm/)
* [Birleştirme Sıralaması (Merge Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/birlestirme-siralamasi-merge-sort/)
* [Yığınlama Sıralaması (Heap Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/yiginlama-siralamasi-heap-sort/)
* [Sayarak Sıralama (Counting Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/sayarak-siralama-counting-sort/)
* [Kabarcık Sıralaması (Baloncuk sıralaması, Bubble Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/kabarcik-siralamasi-baloncuk-siralamasi-bubble-sort/)
* [Taban Sıralaması (Radix Sort)](http://www.bilgisayarkavramlari.com/2008/11/09/taban-siralamasi-radix-sort/)
* [Sokma Sıralaması ( Insertion Sort)](http://www.bilgisayarkavramlari.com/2008/12/12/sokma-siralamasi-ekleme-siralamasi-insertion-sorting/)
* [Sallama Sıralaması (Shaker Sort)](http://www.bilgisayarkavramlari.com/2008/12/18/sallayici-siralamasi-shaker-sort/)
* [Kabuk Sıralması (Shell Sort)](http://www.bilgisayarkavramlari.com/2008/12/20/kabuk-siralama-shell-sort/)
* [Rastgele Sıralama (Bogo Sort)](http://www.bilgisayarkavramlari.com/2009/10/31/bogo-siralama-bogosort/)
* [Şanslı Sıralama (Lucky Sort)](http://www.bilgisayarkavramlari.com/2008/08/09/siralama-algoritmalari-sorting-algorithms/)
* [Serseri Sıralaması (Stooge Sort)](http://www.bilgisayarkavramlari.com/2009/12/26/serseri-siralamasi-stooge-sort/)
* [Şimşek Sıralaması (Flahs Sort, Bora Sıralaması)](http://www.bilgisayarkavramlari.com/2012/05/25/flash-sort-simsek-siralamasi-bora-siralamasi/)
* [Tarak Sıralaması (Comb Sort)](http://www.bilgisayarkavramlari.com/2012/05/27/tarak-siralamasi-comb-sort/)
* [Gnome Sıralaması (Gnome Sort)](http://www.bilgisayarkavramlari.com/2012/05/28/gnome-siralamasi-gnome-sort/)
* [Permütasyon Sıralaması (Permutation Sort)](http://www.bilgisayarkavramlari.com/2012/06/01/permutasyon-siralamasi-permutation-sort/)
* [Strand Sort (İplik Sıralaması)](http://www.bilgisayarkavramlari.com/2013/04/02/strand-sort-iplik-siralamasi/)

Yukarıda verilen veya herhangi başka bir sıralama algoritması genelde küçükten büyüğe doğru (ascending) sıralama yapar. Ancak bunun tam tersine çevirmek (descending) genelde algoritma için oldukça basittir. Yapılması gereken çoğu zaman sadece kontrol işleminin yönünü değiştirmektir.

Ayrıca sıralama işleminin yapılması sırasında hafızanın kullanımına göre de sıralama algoritmaları :

* [Harici Sıralama (External Sort)](http://www.bilgisayarkavramlari.com/2009/03/10/harici-siralama-external-sort/)
* Dahili Sıralama (Internal Sort)

şeklinde iki grupta incelenebilir.

Algoritmaların karşılaştırılması için aşağıdaki tablo hazırlanmıştır:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Algoritma | İngilizcesi | Algoritma Analizi | | | Kararlılı | Yöntem | Açıklama |
| En İyi | Ortalama | En Kötü |
| Seçerek Sıralama | Selection Sort | n2 | n2 | n2 | Kararsız | Seçerek |  |
| Hızlı Sıralama | Quick Sort | n log(n) | n log(n) | n2 | Kararsız | Parçala Fethet |  |
| Birleştirme Sıralaması | Merge Srot | n log(n) | n log(n) | n log(n) | Kararlı | Parçala Fethet |  |
| Yığınlama Sıralaması | Heap Sort | n log(n) | n log(n) | n log(n) | Kararsız | Seçerek |  |
| Sayarak Sıralama | Counting Sort | n + 2k | n + 2k | n + 2k | Kararsız | Sayarak | k ikinci dizinin boyutu. |
| Kabarcık Sıralaması | Bubble Sort | n | n2 | n2 | Kararlı | Yer Değiştirme |  |
| Kokteyl Sıralması | Coctail Sort | n | n2 | n2 | Kararlı | Yer Değiştirme | Çift Yönlü kabarcık sıralaması (bidirectional bubble sort) olarak da bilinir ve dizinin iki ucundan işleyen kabacrık sıralamasıdır. |
| Taban Sıralaması | Radix Sort | n(k/t) | n(k/t) | n(k/t) | Kararlı | Gruplama / Sayma | k, en büyük eleman değeri, t ise tabandır |
| Sokma Sıralaması | Insertion Sort | n | d+n | n2 | Kararlı | Sokma | d yer değiştirme sayısıdır ve n2 cinsindendir |
| Sallama Sıralaması | Shaker Sort | n2 | n2 | n2 | Kararsız | Seçme | Çift yönlü seçme sıralaması (bidirectional selection sort) olarak da bilinir. |
| Kabuk Sıralaması | Shell Sort | n3/2 | n3/2 | n3/2 | Kararsız | Sokma |  |
| Rastgele Sıralama | Bogo Sort | 1 | n n! | sonsuz | Kararsız | Rastgele | Algoritma olduğu tartışmalıdır. Knuth karıştırması (knuth shuffle) süresinde sonuca ulaşması beklenir. |
| Bozo Sıralaması | Bozo Sort | 1 | n! | sonsuz | Kararsız | Rastgele | Rastegele sıralamanın özel bir halidir. Rastgele olarak diziyi karıştırdıktan sonra, dizi sıralanmamışsa, yine rastgele iki sayının yeri değiştirilip denenir. |
| Goro Sıralaması | Goro Sort | 2(log(d) / log(2)) | 2(log(d) / log(2)) | 2(log(d) / log(2)) | Kararsız | Rastgele | 2011 Google kod yarışı (google code jam) sırasında ortaya çıkmıştır. Sıralanana kadar her alt küme permüte edilir. Buradaki performans değeri ispatlanmamıştır ve d dereceyi ifade eder. |
| Şanslı Sıralama | Lucky Sort | 1 | 1 | 1 | Kararsız | Rastgele | Algoritma olarak kabul edilmemelidir. |
| Serseri Sıralama | Stooge Sort | ne | ne | ne | Kararsız | Yer değiştirme | e, doğal logairtma sayısıdır (2,71) |
| Şimşek Sıralaması | Partial Flash Sort | n | n + d | n + d | Kararsız | Yer Değiştirme | d, kullanılan ikinci algoritmanın performansıdır, bu algoritma bu listedekilerden herhangi birisi olabilir. |
| Permütasyon Sıralması | Perm Sort | n | n n! | n n! | Kararsız | Yer Değiştirme |  |
| Bazı Yegane Sıralaması | Several Unique Sort | n | n2 | n2 | Kararsız | Yer Değiştirme | Bir bilgisayar programı tarafından bulunmuştur. |
| Tarak Sıralaması | Comb Sort | n log(n) | n log(n) | n log(n) | Kararsız | Yer Değiştirme | Kabarcık ve hızlı sıralama algoritmalarının birleşimi şeklinde düşünülebilir |

Yukarıdaki yazıda geçen kararlılık kolonu ile, bir algoritmanın bitiş kontrolüne dayanmaktadır. Örneğin sıralı bir dizi verilse bile sıralama işlemi yapmaya çalışır mı?

## **SORU-24: Yahut (Özel Veya (exclusive or, farklılık operatörü)) hakkında bilgi veriniz.**

İki kaziye (önerme) arasındaki farklılık durumuna göre çalışan operatördür. Yani sonuçların aynılığı durumunda yanlış, farklılığı durumunda doğru döndüren operatördür. Basitçe ikili tabanda iki sayının [(bit)](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) farklı olup olmadığını kontrol için de kullanılabilir.

Dilimizde bu işlemi karşılayan kelime “yahut” kelimesidir. Yani a yahut b doğruysanın anlamı a veya b’den birisi doğru diğeri yanlış olacak demektir (ikisi de doğru veya ikisi de yanlış olamaz)

Giren sayılar aynı ise 0, farklı ise 1 sonucu üretir. Aşağıda doğruluk tablosu verilmiştir:

A B Xor  
-------  
0 0 0  
0 1 1  
1 0 1  
1 1 0

## **SORU-25: CRC (cyclic redundancy check, çevrimsel fazlalık sınaması) hakkında bilgi veriniz.**

Hata algılama yaklaşımlarından birisidir. Bu yöntemde işlenmekte olan verinin dışında ilave bir kontrol verisi daha bulunur. Bu ilave bilgi ile bütün bilgi kontrol edilerek bilgide bir bozulma olup olmadığı kontrol edilir. Örneğin ağ iletişiminde gidip gelen bilginin kontrol edilmesinde veya CD gibi kayıt ortamlarında verinin bozulup bozulmadığının kontrol edilmesinde kullanılır.

**Çalışması:**  
Her iki tarafın da bildiği bir sayı bölen olarak kullanılır. Örneğin bölen sayımız 13 olsun ve bunu her iki tarafta başlangıçta biliyor olsun. (Bu sayı stanarda bağlıdır lütfen standartların anlatılıdığı kısma bakınız.)  
Gidecek olan verilerimiz aşağıda yazılmış  
19 54 89 22 03 44 19  
CRC uygularken bu verilerin toplamı alınır : 250  
CRC hesaplanırken daha önceden bildiğimiz sayıya bu sonuç bölünür ve kalan alınır: 250 % 13 = 3  
Dolayısıyla yukarıdaki veriler yollanırken CRC olarak 3 sayısı yollanmaktadır.  
Veri alındıktan sonra kontrol edilmesi:  
yukarıdaki veriler alındıktan sonra alan taraf da verileri toplar, bu toplamdan CRC bilgisini çıkarır ve 250-3= 247 sayısını bulur. Daha önceden bildiği 13 sayısına böler ve 0 sonucunu bulursa iletim hatasızdır yargısına varır 247 % 13 = 0  
şayet hata olsaydı sonuç 0′dan farklı çıkardır. Burada görüldüğü üzere CRC’nin de hata yapma ihtimali vardır örneğin veri bozulması verinin ilk kısmının 1 fazla olmas şeklindeyse:  
20 54 89 22 03 44 19  
verisi alındığında bu hata algılanır : 251 – 3 = 248 % 3 = 1 ve sonuç 0 olmadığı için hata kararına varılacaktı ancak hata miktarı daha önceden bilinen sayının (ki bu örnekte 13) katı şeklinde olursa hatanın yakalanması imkansızır. Örneğin veri bozulması sonucu:  
32 54 89 22 03 44 19  
sayıları çıkmış olsun. Bu sayıların toplamı 263 olacak ve CRC kısmı olan 3 değeri çıktıktan sonra 260 % 13 = 0 olacaktır. Görüldüğü üzere orjinal verimizden farklı olmasına rağmen hatasız olarak kabul edilmiştir.

## **SORU-26: Sayıcı (Counter) hakkında bilgi veriniz.**

Bilgisayar devrelerinde sayıcı (counter) kavramı çok farklı alanlarda kullanılamaktadır. Bu alanlardan birisi de mantıksal devre tasarımıdır. Buna göre mantıksal devremiz ikilik tabandaki sayıları 1′er arttırarak yeni sayılar üretmeli ve bu işlemi bir döngü halinde yapmalıdır. Yani aşağıda verilen durum geçiş diyagramı (state transition diagram) olduğu gibi her durumdan diğer durumlara geçişi başarılı bir şekilde yapmalıdır.

Basitçe her durumdan bir sonraki duruma geçiş bir sayıcı devre için verilmiştir. Buna göre örneğin bir sayıcının 100 durumundan sonra alması istenen durum 101 olmalıdır. Bu sayıcının ikili tabanda çalıştığına dikkat edilmelidir.

Yukarıda verilen bu sayıcının geçiş tablosu aşağıdaki şekilde doldurulabilir. Burada her durumdan bir sonraki duruma geçiş verilmiştir.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Şimdiki  Durum | | | Sonraki  Durum | | |
| A2 | A1 | A0 | A2 | A1 | A0 |
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

Yukarıdaki diyagramda üç [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) için çalışan bir sayıcının önceki ve sonraki durumları verilmiştir. Bu sayıcının [T Flip Flop](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/) ile tasarlanmış hali için T [flip flop](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/)‘un karakteristik tablosuna bakılarak aşağıdaki değerler yazılabilir:

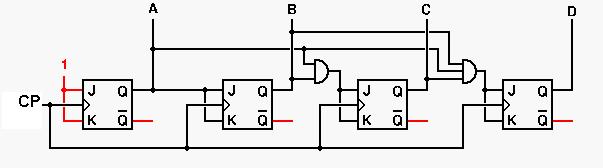
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Şimdiki  Durum | | | Sonraki  Durum | | | FlipFlop  Girişleri | | |
| A2 | A1 | A0 | A2 | A1 | A0 | TA2 | TA1 | TA0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

Yukarıdaki tabloda her durum geçişi için 3 farklı [flip flop](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/)‘un (ki her flip flop ayrı bir [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) için kullanılmıştır) giriş değerlerinin alması gereken değerler verilmiştir. Bu devrenin çizimi aşağıdadır:

Benzer bir sayıcıyı bu defa 4 [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) için ve [JK flip flop](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/) kullanarak ve onluk tabanda yapmayı deneyelim [Flip Flopun](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/) [doğruluk çizelgesi](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) aşağıda verilmiştir:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Durumlar** | | | | **10luk**  **Tabanda**  **Karşılığı** |
| **D** | **C** | **B** | **A** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 2 |
| 0 | 0 | 1 | 1 | 3 |
| 0 | 1 | 0 | 0 | 4 |
| 0 | 1 | 0 | 1 | 5 |
| 0 | 1 | 1 | 0 | 6 |
| 0 | 1 | 1 | 1 | 7 |
| 1 | 0 | 0 | 0 | 8 |
| 1 | 0 | 0 | 1 | 9 |
| 1 | 0 | 1 | 0 | 10 |
| 1 | 0 | 1 | 1 | 11 |
| 1 | 1 | 0 | 0 | 12 |
| 1 | 1 | 0 | 1 | 13 |
| 1 | 1 | 1 | 0 | 14 |
| 1 | 1 | 1 | 1 | 15 |

Bu tabloyu veren [JK flip flop](http://bilgisayarkavramlari.com/2007/12/20/flip-flop-flipflop/) çizimi aşağıdadır:

[](http://bilgisayarkavramlari.com/wp-content/uploads/2008/01/jk_flip_flop_ikili_sayici.jpg)

## **SORU-27: flip flop (flipflop) hakkında bilgi veriniz.**

Flip Flop kavramı temel olarak 1 [bitlik](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) bilginin tutulduğu ünitedir. Bu devre elamanında her zaman iki çıkış olur (Q ve tersi olan Q’). Bu çıkışların değerleri kalıcıdır. Yani bir sonraki giriş değerine kadar geçici süre ile sabitlenmekte ve yeni giriş değerleri alınana kadar sabit olarak kalmaktadır. Bu durum geçici bir hafıza olarak kabul edilebilir. Aşağıda en çok kullanılan Flip Flop tipleri verilmiştir. Bu Flip Flop tipleri temel olup bunların üzerinde değişiklikler yapılarak veya harmanlanarak daha farklı amaçlar için Flip Floplar da üretilebilmektedir.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| FlipFlop İsmi | Sembolü | [Doğruluk Çizelgesi](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) | [Mantıksal Denklem](http://bilgisayarkavramlari.com/2007/11/29/onermeler-mantigi-propositional-logic/) | İç Diyagramı |
| **SR** | [sr-fl.gif](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/sr-fl.gif) | |  |  |  | | --- | --- | --- | | **S** | **R** | **Q(sonra)** | | 0 | 0 | Q | | 0 | 1 | 0 | | 1 | 0 | 1 | | 1 | 1 | ? | | **Q(sonra) = S + R’QSR = 0** |  |
| **JK** | [jk-fl.gif](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/jk-fl.gif) | |  |  |  | | --- | --- | --- | | **J** | **K** | **Q(sonra)** | | 0 | 0 | Q | | 0 | 1 | 0 | | 1 | 0 | 1 | | 1 | 1 | Q’ | | **Q(sonra) = JQ’ + K’Q** |  |
| **D** | [d-fl.gif](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/d-fl.gif) | |  |  | | --- | --- | | **D** | **Q(sonra)** | | 0 | 0 | | 1 | 1 | | **Q(sonra) = D** |  |
| **T** | [t-fl.gif](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/t-fl.gif) | |  |  | | --- | --- | | **T** | **Q(sonra)** | | 0 | Q | | 1 | Q’ | | **Q(sonra) = TQ’ + T’Q** | [T Flip Flop](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/t_flip_flop.jpg) |

Yukarıdaki tablo tahlil edilirken göz önünde bulundurulması gereken önemli bir nokta doğruluk çizelgeleridir. Bu çizelgeler okunurken giriş değeri ve sonucunda alınan çıkış değeri (Q) verilmiştir.

Devrelerin iç diyagramları verilirken dikkat edilirse bütün Flip Floplar için temel olan RS Flip Flop türüdür. Gerçekten de RS Flip Flop en basit Flip Flop örneği olup diğer bütün karmaşık Flip Floplara temel teşkil etmektedir.

Flip Flop devreleri hafızalara ve sayıcılara temel teşkil etmektedir.

## **SORU-28: Salt okunur bellek (read only memory , ROM) hakkında bilgi veriniz.**

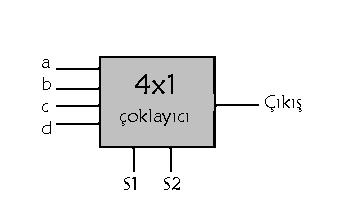
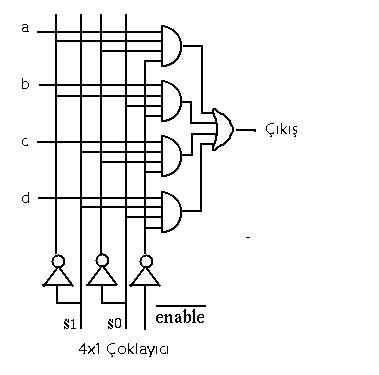
üzerine sadece bir kereye mahsus yazılabilen ve daha sonra istenildiği kadar okuma işlemi yapılabilen hafıza tipidir. Buna göre bellek üretim sırasında üzerine yazılan bilgiyi saklar ve bu bilgiyi değiştirmenin bir yolu yoktur. Daha sonraları çıkan teknolojik ilerlemeler ile aşağıdaki salt okunur bellek türleri kullanıcılara farklı alternatifler sunmaktadırlar:  
PROM (programlanabilir salt okunur bellek, programmable read only memory): Bu hafıza tipi programının müdahalesi sayesinde üzerine bilgi yazılabilen bir yapıya sahiptir. Buna göre programcı hafıza üzerindeki sigortaları program vasıtasıyla attırarak verileri yazabilmektedir. Bu hafıza tipi de sadece bir defaya mahsus programcı tarafından yazılabilmektedir. salt okunur bellekten farkı üzerinde veri yazılı olmadan üretilmekte ve üzerine veriyi programcı yazabilmektedir.

EPROM (silinebilir porgramlanabilir salt okunur bellek, erasable programmable read only memory): Bu hafıza tipi bir onceki PROM’dan farklı olarak ultraviole (mor ötesi) ışıkta kendi üzerine yazılmış bilgileri fabrika çıkışı haline getirmektedir. Yani programcı bu hafıza üzerine veriyi yazabilmekte ve istediği zaman bu bilgileri silerek tekrar programlayabilmektedir.

EEPROM (elektronik olarak silinebilir ve programlanabilir bellek, electrically erasable programmable read only memory): Bu hafıza tipi bir önceki EPROM’dan farklı olarak elektrik akımıyla silinebilmektedir. Bu sayede devreyi kesin ve hızlı bir şekilde sıfırlamak ve daha sonra üzerine yeni bilgileri programlamak mümkün olmaktadır.

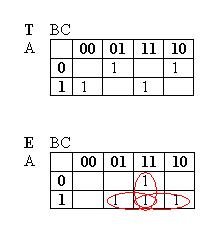
Bir salt okunur belleğin tasarım şeması aşağıda verilmiştir.  
  
yukarıdaki resim morris mano’nun kitabından bir alıntıdır. Bu resimde bir kod çözücü devreye bağlanmış veya kapıları görülmektedir. Buna göre kod çözücü giriş sayısına göre bütün olası sonuçları üretmekte ve bu sonuçları veya kapıları ile birleştirmektedir. Şekildeki kabloların bağlantısı bir 2 boyutlu diziyi andırmaktadır. Buna göre hangi satırın seçileceği kod çözücünün seçilen bacağına göre belirlenmektedir. Bu satırdaki kısa devre seçenekleri ise aşağıdaki veya kapılarının sonucu olarak okunmaktadır.

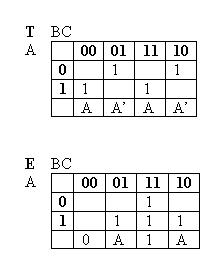
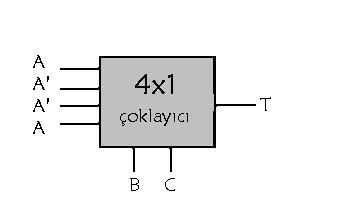
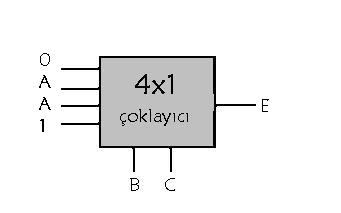
## **SORU-29: çoklayıcı (multiplexer) hakkında bilgi veriniz.**

Çok sayıdaki girişin tek bir giriş üzerinden taşınmasıdır. Amaç çok sayıdaki girişin (örneğin 4 giriş) tek bir çıkışa düşürülmesidir. Çalışma mantığı, anlık olarak tek bir girişi çıkışa vermek şeklindedir. Yani 4 girişten sadece bir tanesi çıkış ile kısa devre halindedir, diğer girişler ise ihmal edilir. Hangi girişin çıkışa verileceğini belirlemek için bir seçme işlemi yapılması gerekmektedir. Bu seçme işlemini yapan bitlere seçici bit(select bit) adı verilmektedir.  
Örneğin 4 girişi olan bir devre için (Girişler a,b,c ve d isminde olsun) 2 adet seçici bit gerekmektedir çünü 4 bit ancak 2 bit ile adreslenebilir (2 üzeri 2) bu durumu gösteren temsili tablo aşağıda verilmiştir:  
  
S1 S2 Ç  
-- -- -  
0 0 a  
0 1 b  
1 0 c  
1 1 d  
  
yukarıdaki tablonun çalıştığı devrenin görüntüsü aşağıda verilmiştir.  
[](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/coklayici-multiplexer/)  
Yukarıdaki şeması verilen devrenin girişlerinden hangisini çıkış ile bağlanacağına seçici bitler karar verirler. Bu devrenin tasarımı aşağıda verilmiştir:  
[](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/kapilar-ile-coklayici/)

Yukarıda tanımı verilen özellik kullanılarak devre tasarımında kısaltmaya gidilebilir. Buna göre karnaugh haritasında verilen her hücre tasarımın sonucunda bir girişin bağlanması ile yapılmaktadır. Bu özellik kullanılarak bir tam toplayıcıyı, kod çözücü devre yardımı ile tasarlayalım:  
Öncelikle [tam toplayıcı devrenin](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/) çalışmasını hatırlayalım:

[tam toplayıcı devrenin](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/) [doğruluk çizelgesini](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) hatırlayalım (3 bitlik giriş için 2 bit çıkışı olan ve girişlerin sayısal değerlerinin toplandığı devre idi):  
  
A B C E T  
- - - - -  
0 0 0 0 0  
0 0 1 0 1  
0 1 0 0 1  
0 1 1 1 0  
1 0 0 0 1  
1 0 1 1 0  
1 1 0 1 0  
1 1 1 1 1  
  
dolayısıyla yukarıda A B ve C girişlerinin toplam değerleri T ve E bitlerinde verilmiştir.

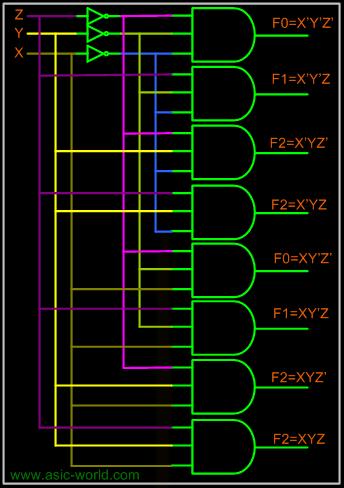
bu doğruluk çizelgesinin [karnaugh haritası](http://bilgisayarkavramlari.com/2007/12/07/karnaugh-haritasi-karnaugh-map/) aşağıda verilmiştir:  
[](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/tam-toplayici-devrenin-ozel-veya-xor-ile-tasarimi/)

Şimdi bu karnaugh haritasının her sütunu için geçerli olan terimleri bulalım:  
[](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/coklayici-ile-tam-toplayici-haritasi/)  
yukarıdaki tabloda, bir tam toplayicinin karnaugh haritasının üzerinde sütun bazlı olarak işlem yapılmıştır. Buna göre her sütunda (yain BC ikili ihtimali için) çıkan sonuç en alt satırda gösterilmiştir. Örneğin E biti için, BC ikilisi 11 olduğunda sonuç her zaman 1 çıkmaktadır (A bitinin sonuca bir etkisi yoktur) benzer şekilde yine E biti için BC ikilisi 0 olduğunda sonuç her zaman 0 olmaktadır (yine A bitinin sonuca bir etkisi yoktur) örneğin E biti için BC ikilis 01 olduğunda sonuç A bitine bağlıdır. Şayet A biti 1 ise sonuç 1 , A biti 0 ise sonuç 0 çıkmaktadır. Bu durumda da BC ikilisinin 01 olduğu durum için sonuç A’dır denilebilir.  
O halde yukarıda verilen bu özelliklerden faydalanarak bir çoklayıcı (multiplexer) devresi ile tam toplayıcı tasarlayalım.  
Bu işlemden önce dikkat edilmesi gereken bir husu, karnaugh haritalarında sütun numaraları yazılırken 00, 01, 11, 10 sıralaması ile gitmesidir. Oysaki çoklayıcı devrenin doğruluk çizelgesine dikkat edilirse sıralama 00, 01, 10, 11 şeklinde gitmektedir. Bu yüzden karnough haritasındaki son iki sütun çoklayıcı devrede yer değiştirmektedir. Okuyucu buna dikkat etmelidir.  
[](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/coklayici-ile-tam-toplayici-devrenin-t-bitini-veren-sema/)  
[](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/coklayici-ile-tam-toplayici-devrenine-bitini-veren-sema/)  
Yukarıdaki iki ayrı devrede iki ayrı çıkış değeri için çözüm yapılmıştır. Buna göre [kod çözücü](http://bilgisayarkavramlari.com/2007/12/09/kod-cozucu-decoder/) devre veya [“ve” ve “veya” kapıları ile tasarlanan bir tam toplayıcı](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/) ile aynı işi yapan yukarıdaki devrede çoklayıcının bize sağlamış olduğu avantaj kullanılarak daha az elemanla tasarım yapılmıştır.

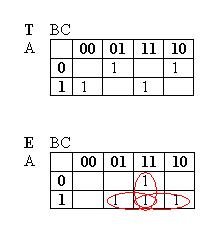
Çoklayıcı devreler günümüzde kullanılan pekçok devrenin temelinde bulunmaktadır. Örneğin ağ iletişiminde kullanılan HUB cihazının tasarımı basit bir multiplexerdır.

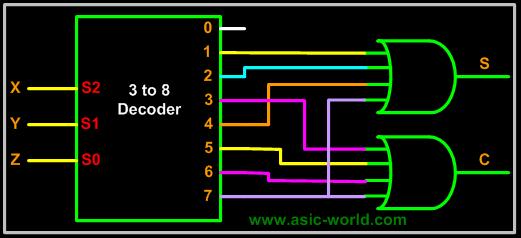
## **SORU-30: kod çözücü (decoder) hakkında bilgi veriniz.**

mantıksal devre tasrımının önemli parçalarından birisi olan kod çözücülerin çalışma mantığı giriş bitlerinin farklı bacaklara dağılmasıdır. Yani örneğin 2 girişli bir kod çözücünün 4 farklı çıkışı olur (2 üzeri 2) ve her çıkış sadece bir giriş ihtimali için çalışır. Daha basit anlatmak gerekirse aşağıdaki [doğruluk çizelgesini](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) inceleyelim:  
  
A B a b c d  
- - - - - -  
0 0 1 0 0 0  
0 1 0 1 0 0  
1 0 0 0 1 0  
1 1 0 0 0 1  
  
yukarıdaki [doğruluk çizelgesinde](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) A ve B girişleri için 4 farklı çıkış değeri (a b c ve d) verilmiştir. Dikkat edilirse giriş değerlerinin alabileceği her ihtimal için farklı bir çıkış değeri 1 olmuş geri kalanlar 0 olmuştur.  
Bu durumda kod çözücülerin ana amacı farklı girişlerin mantıksal “ve” işlemini almaktır. Yani yukarıdaki çizelgede her satırda farklı bir ihtimalin “ve” sonucu görülmektedir. Bu durum aşağıdaki tabloda gösterilmiştir:  
  
A B a b c d  
- - - - - -  
0 0 A'B' -> a  
0 1 A'B -> b  
1 0 AB' -> c  
1 1 AB -> d  
  
yukarıdaki tabloda her satırda mantıksal olarak farklı bir ihtimal gösterilmiş ve sonucun hangi çıkış bacağında görüldüğü verilmiştir. Buna göre 2x4lük bir kod çözücünün a bacağı her zaman A’B’ [önermesini](http://bilgisayarkavramlari.com/2007/11/29/onermeler-mantigi-propositional-logic/) verir. Bir kod çözücünün kapılar ile tasarımı aşağıdaki şekildedir.

[](http://bilgisayarkavramlari.com/2007/12/09/kod-cozucu-decoder/kod-cozucu-decoder/)  
Bu özellik kullanılarak devre tasarımında kısaltmaya gidilebilir. Çünkü klasik yaklaşımda [karnaugh haritası](http://bilgisayarkavramlari.com/2007/12/07/karnaugh-haritasi-karnaugh-map/) ile çizilen devrelerde bir [“ve” kapısı](http://bilgisayarkavramlari.com/2007/11/29/ve-kapisi-and-gate/) grubu bir de [“veya” kapısı](http://bilgisayarkavramlari.com/2007/11/29/veya-kapisi-or-gate/) grubu bulunmaktadır (önce ve kapısında eşleşen girişler daha sonra veya kapısı ile toplanmaktadır bkz. tam toplaycı veya yarı toplayıcı) Bu yaklaşımındaki “ve” kapı grubunu kod çözücü ile kaldırarak sadece “veya” kapı grupları ile kod çözücünün çıkış bacaklarını birbirine bağlamak mümkündür. Çünkü yukraıdaki örnekte de görüleceği üzere girişlerin tamamının bütün muhtemel “ve” kapısı sonuçları kod çözücü ile bulunabilmektedir.

Örnek olarak bir [tam toplayıcı devrenin](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/) kod çözücü ile tasarımı aşağıda verilmiştir:

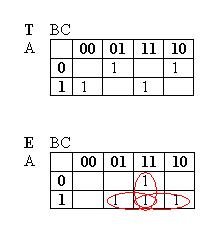
[tam toplayıcı devrenin](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/) [doğruluk çizelgesini](http://bilgisayarkavramlari.com/2007/12/08/dogruluk-cizelgesi-truth-table/) hatırlayalım (3 bitlik giriş için 2 bit çıkışı olan ve girişlerin sayısal değerlerinin toplandığı devre idi):  
  
A B C E T  
- - - - -  
0 0 0 0 0  
0 0 1 0 1  
0 1 0 0 1  
0 1 1 1 0  
1 0 0 0 1  
1 0 1 1 0  
1 1 0 1 0  
1 1 1 1 1  
  
dolayısıyla yukarıda A B ve C girişlerinin toplam değerleri T ve E bitlerinde verilmiştir.  
bu doğruluk çizelgesinin [karnaugh haritası](http://bilgisayarkavramlari.com/2007/12/07/karnaugh-haritasi-karnaugh-map/) aşağıda verilmiştir:  
[](http://bilgisayarkavramlari.com/2007/12/08/tam-toplayici-full-adder/tam-toplayici-devrenin-ozel-veya-xor-ile-tasarimi/)  
şimdi yukarıdaki haritaya ve doğruluk çizelgesine bakıldığında görülür ki aslında doğruluk çizelgesinin her satırı kod çözücünün farklı bir bacağıdır. Bu durumda tam toplayıcı devre aşağıdaki şekilde de tasarlanabilir:

[](http://bilgisayarkavramlari.com/2007/12/09/kod-cozucu-decoder/kod-cozucu-decoder-ile-tam-toplayici/)  
Yukarıdaki devrede S, T bitine ve C ise E bitine karşılık gelmektedir. {bu devreyi çizip orjinal resim konulacak}

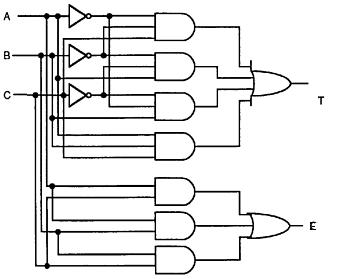
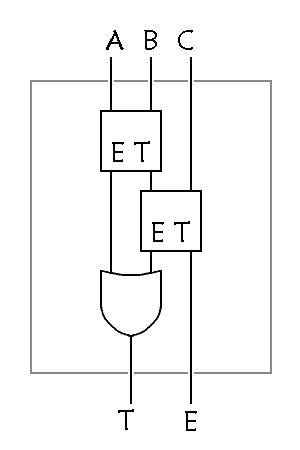
## **SORU-31: tam toplayıcı (full adder) hakkında bilgi veriniz.**

3 bitlik giriş değerlerinin (iki tabanındaki girişler) toplamını veren devredir. Buna göre A, B ve C girişleri için aşağıdaki tablo elde edilir. (aşağıdaki tablodaki + işareti önermeler arası veya ile karıştırılmamalıdır. + işareti toplamayı ifade eder)

A B C E T  
- - - - -  
0 0 0 0 0  
0 0 1 0 1  
0 1 0 0 1  
0 1 1 1 0  
1 0 0 0 1  
1 0 1 1 0  
1 1 0 1 0  
1 1 1 1 1

yukarıda verilen çizelgede A, B ve C sayılarının toplamı E ve T değerlerinde gösterilmiştir. Örneğin A 1, B 1 ve C 1 değerleri için (çizelgenin son satırı) toplam 3 olmaktadır (1+1+1=2) bu değerin ikilik tabandaki karşılığı 11′dir. 2 çıkış olmasının sebebi toplanan sayıların tek [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) ile ifade edilememesindendir.  
Yukarıda verilen toplam değerlerini veren devreyi tasarlarken E ve T [ikillerini (Bit)](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) ayrı ayrı düşünmek gerekir. Bu toplama işlemini yapan devrenin tasarımında karnaugh haritalarından faydalanılırsa T ve E ikilleri (bit) için aşağıdaki haritalar çizilebilir:  
[](http://www.bilgisayarkavramlari.com/?attachment_id=119)  
yukarıdaki haritada, üç giriş için toplam değerlerini veren iki farklı çıkış (T toplam ve E elde var) bitlerinin değerleri gösterilmektedir. Bu değerler arası komşuluklar kırmızı daire içerisine alınmıştır. Bu 1lerden oluşan adalar sadeleştirildiğinde:  
T=A’B'C+A’BC’+ABC+AB’C’  
E=AB+AC+BC  
olarak bulunur.

Bu devrenin tasarımı aşağıdaki şekilde verilmiştir:

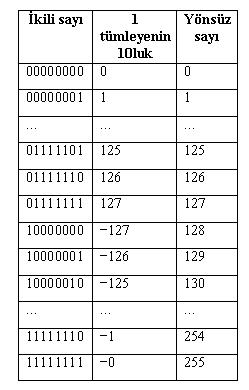
[](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/tamtoplayicidevre2.jpg)  
Yukarıdaki devre [“ve”](http://bilgisayarkavramlari.com/2007/11/29/ve-kapisi-and-gate/) ve [“veya](http://bilgisayarkavramlari.com/2007/11/29/veya-kapisi-or-gate/)” kapıları kullanılarak yapılan devre tasarımıdır. Bu devre “özel veya” ve [“yarım toplayıcı”](http://bilgisayarkavramlari.com/2007/12/08/yarim-toplayici-half-adder/) devreleri kullanılarak da yapılabilir. Aşağıda bu devre tasarımları da verilmiştir:  
**Bir tam toplayıcının  yahut devreleri (“özel veya” (xor)) ile tasarımı**  
  
**Bir tam toplayıcıyı iki adet** [**yarım toplayıcı**](http://bilgisayarkavramlari.com/2007/12/08/yarim-toplayici-half-adder/) **ile tasarlamak da mümkündür.**  
[](http://bilgisayarkavramlari.com/wp-content/uploads/2007/12/tamtoplayici_yarimtoplayiciile.jpg)

Tam toplayıcı aynı zamanda [kod çözücü (decoder)](http://bilgisayarkavramlari.com/2007/12/09/kod-cozucu-decoder/) ve [çoklayıcı (multiplexer)](http://bilgisayarkavramlari.com/2007/12/09/coklayici-multiplexer/) ile de tasarlanabilir.

## **SORU-32: Bir tümleyeni hakkında bilgi veriniz.**

Konunun diğer isimleri : (1 tümleyeni, İşaretli sayı gösterimi, Ones’ Complement, 1′s Complement, Signed number representations)

ikilik tabandaki bir sayının 1 tümleyeni her sayının tersidir. Örneğin sayı:  
  
10110011  
  
olarak verilmiş olsun. Bu sayının 1 tümleyeni:  
  
01001100  
  
olarak bulunur.  
Bu bilgi [2 tümleyeni](http://bilgisayarkavramlari.com/2007/11/28/iki-tumleyeni/)nin hesabında da kullanılmaktadır.

Bir tümleyeni aynı zamanda sayının eksi değer olarak gösterilmesine de yaramaktadır. Aşağıdaki tabloda sayıların alabileceği değerler ve bu sayıların ikili ve onluk tabandaki gösterimleri verilmiştir:  
[](http://bilgisayarkavramlari.com/2007/11/28/bir-tumleyeni/bir-tumlyeni-ve-karsiligi-onluk-sayi/)  
yukarıdaki tabloda ikilik sistemde bir takım sayılar verilmiştir. Tablonun ikinci kolonunda bu sayıları 1 tümleyeni olarak yorumladığımızda 10luk sistemdeki karşılıkları, üçüncü kolonda ise bu sayıları normal birer ikilik sayı gibi görüp, 10luk sisteme çevirince çıkan karşılıkları verilmiştir.  
Buna göre 0 sayısının bütün bitlerinin tersi -0 veya örneğin 10 sayısının bütün bitlerinin tam tersi -10 olmaktadır.  
Bu ayrımı ilk bit belirlemektedir bu bit’e sign bit (yön biti) de denilmektedir.

## **SORU-33: İki tümleyeni hakkında bilgi veriniz.**

Konunun diğer başlıkları: 2 tümleyeni, two’s complement

Bilgisayar bilimlerinde, sayılar genelde ikilik tabanda tutulmaktadır. Değerleri ikilik tabanda göstermenin bir devamı olarak eksi sayı ve artı sayıları da ayırmak gerekmektedir. [bir tümleyeni](http://bilgisayarkavramlari.com/2007/11/28/bir-tumleyeni/) gibi iki tümleyeni de eksi sayıları göstrem biçimlerinden birisidir. iki tümleyenini almak için önce [bir tümleyeni](http://bilgisayarkavramlari.com/2007/11/28/bir-tumleyeni/) alınır ardında sayıya ikilik tabanda 1 eklenir.  
Örneğin  
11011001   
sayısının [bir tümleyeni](http://bilgisayarkavramlari.com/2007/11/28/bir-tumleyeni/) aşağıda verilmiştir:  
00100110   
bu sayıya 1 eklenerek, iki tümleyeni elde edilir:  
00100111

Bu sayı aynı zamanda orjinal sayı olan 11011001 sayısının da negatifi gösterimi olarak kullanılabilir.  
Bunu bir örnek ile göstermek gerekirse, aşağıdaki çıkarma işlemini ele alalım:

11001001

10110101

-

---------

00010100

bilindiği üzere aslında çıkarma işlemini, çıkarılan sayının negatifini alıp toplama olarak da yorumlayabiliriz.  
Bu durumu aynı örnek için tecrübe edelim. Öncelikle çıkarılan sayı olan 10110101 sayısının negatifini alalım, yani iki tümleyenini:  
01001010 sayısı elde edilir. Şimdi bu sayının gerçekten negatif olduğunu yukarıdaki örneği toplamaya çevirerek görelim:

11001001

01001010

+

---------

100010100

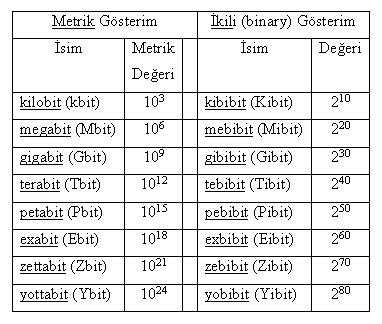
Görüldüğü üzere elde edilen sonucun başında bulunan 1 atılırsa, ilk işlemden çıkan sonuç ile aynıdır.

## **SORU-34: bit (ikil) hakkında bilgi veriniz.**

Bilgisayar dünyasında ikili tabandaki (binary) tek haneli bir sayıyı ifade eder. Yani bir bit değeri 1 veya 0 olabilir. Bu aslında elektronik sinyali olarak yüksek (1) veya düşük (0) gerilimde akım demektir.

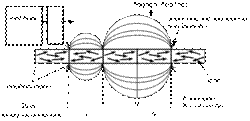
bir bit, 1 veya 0 değeri alabildiğine göre her bit değerinin 2 farklı değer alması mümkündür. Bu durumda örneğin iki sistemde yazılmış 8 haneli bir sayı (8 bitlik bir sayı) 2 üzeri 8 farklı değer = 256 farklı değer alabilir.  
Örneğin 11010010 sayısı, 8 bitlik bir sayıdır.

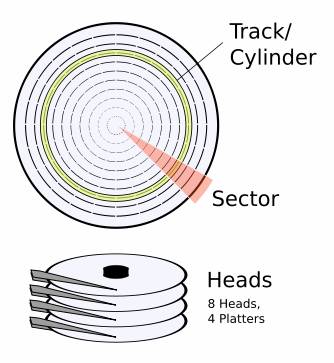
8 bitlik sayılara bilgisayar dünyasında kısaca byte adı da verilmektedir.

Bit değerleri için kullanılan birimler aşağıda listelenmiştir:  


## **SORU-35: Disk Yönetimi (Disk Management) hakkında bilgi veriniz.**

Aşağıdaki resimde klasik bir sabit diskin parçalarının isimleri gösterilmiştir:





Temel kavramlar:

Kafa (head) : Güncel sabit diskler genelde birden fazla disk içermektedirler. Her diskin üzerine işlem yapan (okuyan, yazan veya hareket eden) manyetik uca kafa denilir. Güncel disklerin iki yüzü de işlenebilir olduğu için bir disk için 2 kafa kullanılır.

İz (track) Disk üzerinde kafanın hareket ettiği ve manyetik olarak işlenebilen alanlardır.

Silindir (cylinder): Diskler daire şeklinde olduğu için, disk üzerindeki izler çembere benzetilebilir. Birden fazla disk olduğu için ve her disk üzerinde aynı koordinatlara çember olduğu düşünülürse, bu çemberlerin birleştirilmesi bir silindir görüntüsünde olur. Dolayısıyla silindir birden fazla diskte aynı yere düşen izler demektir.

Sektör (sector) Disk üzerindeki izlerin bölündüğü alt parçalardır. Bu parçalar sayesinde izler üzerinde bilgi gruplanabilmektedir. Yuvarlak bir pasta (disk) üzerindeki dilimler gibi düşünülebilir.

Diskin üzerinde bilgilerin durması manyetik alan yönüne göre olur. Örneğin aşağıdaki resimde sağa doğru olan yükleme 1, sola doğru olan yükleme 0 olarak kabul edilmşitir:

Blok (Block) Sektör ve izlerin kesişimidir. Disk üzerindeki en küçük birimdir.

Yukarıdaki bilgiler ışığında aşağıdaki formüller elde edilebilir:

Diskin bir yüzündeki blok sayısı: diskteki silindir sayısı \* diskteki sektör sayısı

Disk başına blok sayısı: Diskin bir yüzündeki blok sayısı \* diskin yüz sayısı

Disk başına blok sayısı: Diskin bir yüzündeki blok sayısı \* disk başına kafa sayısı

Disk başına blok sayısı: diskteki silindir sayısı \* diskteki sektör sayısı \* disk başına kafa sayısı

Toplam blok sayısı: Silindir sayısı \* disk başına kafa sayısı \* sektör sayısı \* disk sayısı

Toplam blok sayısı: silindir \* kafa \* sektör

Örnek: 100 silindiri 2 kafası ve 20 sektörü olan bir diskin kaç bloğu vardır?

Toplam blok sayısı: 100 \* 2 \* 20

Toplam blok sayısı: 4000 olarak bulunur.

Şayet bir blok boyutu 512 byte olarak kabul edilirse diskin kapasitesi:

4000 \* 512 = 2MB olarak bulunur.

LBA ( logical block addressing) mantıksal blok adreslemesi:

Disk üzerindeki blokların adreslenmesi için kullanılır. Diskte bulunan ilk blok lba0 , ikinci blok lba1 şeklinde adreslenir.

CHS (cylinder head sector) Silindir Kafa Sektör

Disk üzerinde blokların adreslenmesi için kullanılan ikinci yöntemdir. Diskte bulunan ilk blok 0 0 0 şeklinde ikinci blok 0 0 1 şeklinde adreslenir. 3 sayı tutar, silindir, kafa ve sektör bilgisi ayrı ayrı durur.

Örnek dönüşüm tablosu:

LBA Değeri

CHS Satırı

0

0, 0, 1

1

0, 0, 2

2

0, 0, 3

62

0, 0, 63

63

0, 1, 1

64

0, 1, 2

65

0, 1, 3

125

0, 1, 63

126

0, 2, 1

127

0, 2, 2

188

0, 2, 63

189

0, 3, 1

190

0, 3, 2

16,063

0, 254, 62

16,064

0, 254, 63

16,065

1, 0, 1

16,066

1, 0, 2

16,127

1, 0, 63

16,128

1, 1, 1

16,450,497

1023, 254, 1

16,450,558

1023, 254, 62

16,450,559

1023, 254, 63

Yukarıdaki tabloya göre bir disk CHS kullanılarak en fazla:

1024 \* 255 \* 64 \* 512 ( byte) = 8.4GB olabilir.

CHS sistemi MS-DOS ve NT4.0 versiyonu bilgisayarlarda kullanılan bir sistemdi ancak yukarıda da görüldüğü üzere adreslemedeki yetersizlik yüzünden ilerletilmesi gereken bir sistemdir.

ATA sürücülerinde kullanılan yeni teknoloji sayesinde ECHS (enhanced CHS) silindir sayısı arttırılabilmektedir:

LBA Değeri

CHS Satırı

0

0, 0, 1

1

0, 0, 2

2

0, 0, 3

62

0, 0, 63

945

0, 15, 1

1007

0, 15, 63

1008

1, 0, 1

1070

1, 0, 63

1071

1, 1, 1

1133

1, 1, 63

1134

1, 2, 1

2015

1, 15, 63

2016

2, 0, 1

16,127

15, 15, 63

16,128

16, 0, 1

32,255

31, 15, 63

32,256

32, 0, 1

16,450,559

16319, 15, 63

16,514,063

16382, 15, 63

Yukarıdaki dönüşüm tablosunu kullanan bir disk:

16384 \* 16\* 42 blok işleyebilmektedir bu değer de 128 GB alan karşılık gelmektedir. ATA-6 standardı ile bu değer 28 [bitlik](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) bilgiden 48 [bitlik](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) bilgiye çıkartılmıştır. Dolayısıyla adreslene bilen bilg 128PB olabilmektedir. 64 [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/)’e çıkan son adresleme yöntemi ile de bu bilgi 9 tirilyon GB’a kadar çıkabilmektedir. Burada BIOS’un bu adresleme şeklini desteklemesi gerektiği unutulmamalıdır.

Arayüz

Standart CHS

Extended CHS (ECHS) / Large

Logical Block Addressing

Entegre disk kontrolune eklenen basit diskler

Fiziksel geometri

Fiziksel geometri

Fiziksel geometri

BIOS’a entegre edilmiş disk kontorlu

Mantıksal Geometri

Mantıksal Geometri

Mantıksal Blok Adresleme(LBA)

BIOS’tan işletim sistemi ve uygulamalara geçiş ( Int 13h ile)

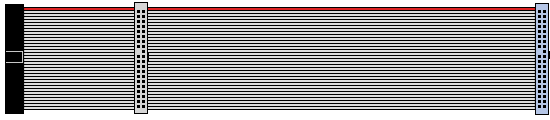
Mantıksal Geometri

Çevirilmiş Geometri

Çevirilmiş Geometri

IDE (Integrated Drive Electronics) sabit disk bağlantılarını (ana kart ile) standartlaştırmak için Western Digital firması tarafından geliştirilmiş olan 40 [bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/)lik standarttır. Daha sonraları SATA (serial ATA) ile ayırmak için PATA (paralel ATA) isminde de kullanılmıştır.

EIDE: IDE teknolojisinin adreslemede yetersiz kalması üzerine (max. 520mb adreselenebilmekteydi) 8.4 GB’a kadar adresleme yapabilen ve yine western digital firması tarafından geliştirilen standarttır.

  
  
Yukarıdaki resimde bir ide kablosu ve ana kart üzerindeki bağlantı yuvası gösterilmiştir.

Standardın ismi

Diğer ismi

Transfer şekli (MB/s)

Maximum disk boyutu

Diğer yeni özellikler

pre-ATA

IDE

PIO 0

2.1 GB

22-[bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) logical block addressing (LBA)

ATA-1

ATA, IDE

PIO 0, 1, 2 (3.3, 5.2, 8.3)  
Single-word DMA 0, 1, 2 (2.1, 4.2, 8.3)  
Multi-word DMA 0 (4.2)

137 GB

28-[bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) logical block addressing (LBA)

ATA-2

EIDE, Fast ATA,  
Fast IDE, Ultra ATA

PIO 3, 4: (11.1, 16.6)  
Multi-word DMA 1, 2 (13.3, 16.6)

ATA-3

EIDE

S.M.A.R.T., Security

ATA/ATAPI-4

ATA-4, Ultra ATA/33

Ultra DMA 0, 1, 2 (16.7, 25.0, 33.3)  
aka UDMA/33

AT Attachment Packet Interface (ATAPI), i.e. support for CD-ROM, tape drives etc.,  
Optional overlapped and queued command set features,  
Host Protected Area (HPA)

ATA/ATAPI-5

ATA-5, Ultra ATA/66

Ultra DMA 3, 4 (44.4, 66.7)  
aka UDMA/66

80-wire cables

ATA/ATAPI-6

ATA-6, Ultra ATA/100

UDMA 5 (100)  
aka UDMA/100

144 PB

48-[bit](http://bilgisayarkavramlari.com/2007/11/24/bit-ikil/) LBA, Device Configuration Overlay (DCO),  
Automatic Acoustic Management

ATA/ATAPI-7

ATA-7, Ultra ATA/133

UDMA 6 (133)  
aka UDMA/133  
SATA/150

SATA 1.0, Streaming feature set, long logical/physical sector feature set for non-packet devices

ATA/ATAPI-8

ATA-8

—

Hybrid drive featuring non-volatile cache to speed up critical OS files

Disk Erişim Hızları.

Diskin üzerindeki bir veriye okumak veya yazmak için erişilmesi için belirli bir vakit geçmektedir. Bu geçen vakit aşağıdaki parçalara ayrılabilir:

Rotational Delay (döme gecikmesi)

Diskin ilgili sektörünün kafanın altına getirilmesi için disklerin döndürülmesi süresidir.

İki türlü olabilir :

Constant Angular Velocity: Sabit Açısal Hız. Bu çeşit disklerde açısal hız sabittir yani disk her zaman aynı hızda döner.

Constant Linear Velocity: Sabit Doğrusal Hız: Bu çeşit disklerde kafanın okuma hızına göre disk hızı yavaşlayıp artar. Örneğin diskin merkezine yakın bölgelerinde daha çevre uzunluğu kısaldığı için, bu bölgelerden okuma yapıldığı sırada doğrusal hız sabit tutulmak için diskin hızı azaltılır. Veya diskin dışına doğru uzunluk arttığı için diskin hızı arttırılır.

Seek Time: Arama Zamanı

Kafanın bir track (silindir, iz) üzerine gelmesi için kafanın hareket etmesi süresidir.

Latency Time (erişim süresi) verinin manyetik ortamdan okunup işlenecek hale çevrilmesi süresidir.

Bir bilginin diskten okunması veya yazılması için Dönme gecikmesi + arama zamanı + latency kadar vakit geçmelidir.