metin içeren bir resim

Açıklama otomatik olarak oluşturuldu**YAZ302 SİSTEM PROGRAMLAMA**

**FİNAL PROJE RAPORU**

|  |  |
| --- | --- |
| **GRUP ÜYELERİ** | **1)Gürkan GÖZTEPELİ**  **2)Vesile HAN** |

İÇİNDEKİLER

[1. Assembler Tasarımının Genel Bakışı 3](#_Toc166964973)

[2. Uygulama 5](#_Toc166964974)

[3. Kullanım Kılavuzu 6](#_Toc166964975)

[4. Karşılaşılan Zorluklar ve Çözümlemeleri 7](#_Toc166964976)

[5. Grup Üyelerinin Sorumlulukları 8](#_Toc166964977)

[6. Kaynakça 9](#_Toc166964978)

# Assembler Tasarımının Genel Bakışı

* Assembler Nedir ?

Assembler, bilgisayar programcılarının makine diline yakın kod yazmasını sağlayan düşük seviyeli bir programlama dilidir. Bilgisayarlar, yüksek seviyeli dilleri doğrudan anlayamaz, bu yüzden assembler dilleri kullanılarak makine diline çevrilir. Assembler, sembolik kodları makine diline dönüştürerek bellek yönetimi, veri transferi, aritmetik işlemler ve kontrol akışı gibi temel işlevleri yerine getirir.

Her işlemci ailesine özel assembler dili bulunur, bu nedenle programcılar hedef işlemciye uygun dili kullanmalıdır. Assembler, hızlı ve optimize edilmiş kod üretir, ancak yüksek seviyeli dillere göre daha karmaşıktır ve deneyim gerektirir. Assembler, düşük seviyeli kodlama yaparak işlemcinin tam potansiyelini kullanmayı sağlar.

* Assembler Nasıl Çalışır ?

1. Kaynak Kodun Kabulü: Assembler, programcı tarafından yazılan assembly dili kaynak kodunu kabul eder. Bu kaynak kod, insanlar tarafından anlaşılabilir sembolik komutlar ve etiketler içerir.

2. Satırların Ayrıştırılması\*\*: Assembler, kaynak kodun her satırını ayrıştırarak sembolik adları ve talimatları tanımlar. Her satır, bir talimat veya veri bildirimi olabilir. Assembler, bu sembolik adları ve talimatları anlamak için satırları tek tek analiz eder.

3. Bellek Tahsisi: Assembler, kaynak koddaki her talimat ve veriye bellek tahsis eder. Bu, programın çalışması sırasında kullanılacak bellek alanının belirlenmesini sağlar. Assembler, her talimat ve veri için uygun bellek adreslerini atar.

4. Program Sayacı Kullanımı: Assembler, bir program sayacı (PC) veya konum sayacı (LC) kullanır. Bu sayaç, assembler'ın işlemdeki bir sonraki talimatın bellek adresini takip etmesine yardımcı olur. Program sayacı, her talimat işlendiğinde güncellenir ve bir sonraki talimatın adresini gösterir.

5. Tabloların Tutulması: Assembler, sembol tablosu ve mnemonic tablosu tutar.

- Sembol Tablosu: Bu tablo, kaynak kodda kullanılan sembolik adları ve bunların karşılık gelen bellek adreslerini içerir. Sembol tablosu, programın farklı bölümlerinde kullanılan sembollerin izlenmesine yardımcı olur.

- Mnemonic Tablosu: Bu tablo, mnemonic işlem kodlarını ve bunların makine kodu karşılıklarını içerir. Mnemonic tablosu, assembler'ın kaynak koddaki talimatları makine koduna dönüştürmesine yardımcı olur.

6. Makine Koduna Dönüşüm: Assembler, sembolik işlem kodlarını makine kodu olarak adlandırılan sayısal bit yapılandırmalarına dönüştürür. Bu süreç, sembolik adların ve talimatların sayısal karşılıklarına çevrilmesini içerir. Makine kodu, bilgisayarın işlemcisi tarafından doğrudan çalıştırılabilen bit dizilerinden oluşur.

7. Makine Kodunun Kaydedilmesi: Makine kodu, makine kodu talimatları, veri değerleri ve bellek adreslerinden oluşur. Assembler, bu makine kodunu ayrı bir dosyada saklar. Bu dosya, daha sonra bilgisayarın ana belleğine yüklenir ve işlemci tarafından çalıştırılır.

8. Hata Takibi ve Çözümleme: Assembler, kaynak kodu işlerken sözdizimi ve mantıksal hataları, çözümlenmemiş semboller ve diğer sorunları da takip eder. Hatalar tespit edildiğinde, assembler bu hataları raporlar ve programcıya düzeltme imkânı sunar. Bu, programın doğru ve hatasız bir şekilde çalışmasını sağlar.

Assembler'ın çalışma süreci kısaca,kaynak kodun kabul edilmesinden başlayarak, talimatların ayrıştırılması, bellek tahsisi, program sayacının kullanımı, sembol ve mnemonic tablolarının tutulması, makine koduna dönüşüm, makine kodunun kaydedilmesi ve hata takibi gibi adımlar, assembler'ın temel işlevlerini oluşturur. Bu süreç, programcıların yazdığı assembly kodunun bilgisayarın işlemcisi tarafından çalıştırılabilir hale getirilmesini sağlar.

* İki Geçişli Yöntem ve Neden Gerekli Olduğu Hakkında Bilgi :

İki geçişli assembler yöntemi, assembly dilindeki kaynak kodunu makine koduna dönüştürmek için kullanılan bir stratejidir. Bu yöntem, genellikle ileri referanslarla (henüz tanımlanmamış değişkenler veya alt programlar) başa çıkmak için tercih edilir. İşte bu yöntemin detayları:

İlk Geçiş (Pass 1):

1. Bilgi Toplama: Assembler, kaynak kodu tarar ve sembolik isimlerle adresler arasındaki ilişkileri belirler. Bu aşamada sembolik etiketler ve adreslerin tanımlanması sağlanır.

2. Bellek Tahsisi ve Sembol Tablosu Oluşturma: Assembler, her sembolik isme bir adres tahsis eder ve bu bilgileri sembol tablosunda saklar. Bellek tahsis edilirken ileri referanslar için geçici değerler kullanılabilir.

3. İleri Referansları İşleme: Eğer bir sembolik isim henüz tanımlanmamışsa, bu isim geçici bir değerle işaretlenir.

İkinci Geçiş (Pass 2):

1. Makine Kodu Üretme: İlk geçişte oluşturulan sembol tablosu kullanılarak sembolik işlem kodları makine diline dönüştürülür ve makine kodu üretilir.

2. MOT (Makine İşlem Kodu Tablosu) Kullanımı\* Bu aşamada, makine işlem kodlarının (opcode) karşılık geldiği bilgileri içeren MOT kullanılır. Bu tablo, sembolik kodların uzunlukları, bit yapılandırmaları ve formatları gibi bilgileri içerir.

3. Makine Kodunu Oluşturma ve Kaydetme: Assembler, her sembolik işlem kodunu makine diline dönüştürerek makine kodunu üretir ve bu kodları MOT'ta saklar.

**İki Geçişli Yöntemin Kullanılma Sebepleri:**

1. İleri Referansları Çözme: İlk geçişte ileri referanslar geçici değerlerle işaretlenir ve ikinci geçişte doğru değerler atanır. Bu, henüz tanımlanmamış değişkenlerin ve alt programların doğru bir şekilde işlenmesini sağlar.

2. Optimizasyon ve Doğruluk: İki geçişli yöntem, makine kodunun optimize edilmesini sağlar ve doğruluğu artırır. Ayrıca, sembolik işlem kodları daha sonra kolaylıkla değiştirilebilir veya güncellenebilir.

3. Hızlı Montaj Süreci: İleri referansları çözme ve makine kodu üretme işlemleri ayrı geçişlerde gerçekleştiği için montaj süreci hızlanır.

Aynı zamanda kullanılan bu yöntem, programcıların yazdığı assembly kodunu bilgisayarın işlemcisi tarafından doğru ve verimli bir şekilde çalıştırılabilir hale getirir. Ayrıca, makine kodunun optimize edilmesi ve hata kontrolü gibi avantajlar sağlar.

# Uygulama

Bu uygulama, C++ programlama dili kullanılarak geliştirilmiş bir assembler’dir. Assembler, düşük seviye bir programlama dilidir ve genellikle belirli bir donanım mimarisi için makine diline çevrilir. Bu uygulama, assembly kodlarınızı alır ve onları bir bilgisayarın anlayabileceği makine diline çevirir.

**Girdi İşlemleri**: Assembler programı, assembly dilindeki 8085 mikroişlemci komutlarını bir dosyadan okur. Bu dosya genellikle kullanıcı tarafından hazırlanmış bir metin dosyası olabilir. Dosyadaki komutlar, program tarafından sırayla okunarak işlenir. Bu işlem, assembly dilinde yazılmış kodların doğru bir şekilde yorumlanmasını sağlar.

**Çıktı İşlemleri**: Assembler, okunan komutları opcode’lara çevirerek doğru makine kodunu üretir. Bu makine kodu çıktısı, hexadecimal formatında başka bir dosyaya yazılır. Hexadecimal format, makine kodunun insanlar tarafından daha okunabilir bir şekilde temsil edilmesini sağlar. Bu çıktı dosyası, başka bir uygulamada veya donanımda kullanılabilir.

**İlk Geçiş** sırasında, assembler programı sembol tablosunu oluşturur. Bu tablo, sembolik isimleri (etiketleri) bellek adresleriyle ilişkilendirir. Örneğin, “LABEL1” ismi “1000h” bellek adresiyle ilişkilendirilebilir. Bu işlem, assembly dilindeki sembolik isimlerin doğru bir şekilde çözümlenmesini sağlar.

Ayrıca, ilk geçiş sırasında etiketlerin çözümlemesi yapılır. Yani, program içindeki her etiketin hangi bellek adresiyle ilişkilendirildiği belirlenir. Bu işlem, ileri referansları yönetmek için kullanılır.

**İleri Referansların Yönetilmesi ve Güncellenmesi**: İleri referanslar, henüz tanımlanmamış veya çözümlenmemiş sembollerdir. İlk geçişte bu referanslar yönetilir ve ikinci geçişte doğru bir şekilde çözülerek güncellenir. Bu, sembolik isimler arasındaki bağlantıları sağlar.

**İkinci Geçiş** sırasında, assembler programı sembol tablosundan gelen bilgileri kullanarak 8085 mikroişlemci komutlarının opcode’larına çevirir. Her sembolik komut, karşılık gelen bir opcode’ya dönüştürülür. Bu işlem, assembly dilindeki komutların doğru bir şekilde çevrilmesini sağlar.

Daha sonra, opcode’lar çevrilir ve doğru makine kodu oluşturulur. Bu makine kodu, assembly dilindeki komutların doğru bir şekilde çalıştırılabilmesi için gereklidir.

**Tablolar**: Assembler programı, sembol tablosu ve ileri referans tablosu oluşturur. Sembol tablosu, sembolik isimleri (etiketleri) bellek adresleriyle ilişkilendirir. Bu tablo, sembolik isimlerin doğru bellek adreslerine bağlanmasını sağlar. İleri referans tablosu ise, henüz tanımlanmamış sembollerin çözümlemesini yapar. Bu tablo, sembolik isimler arasındaki ilişkileri güncelleyerek doğru bağlantıları sağlar.

**Hata Yönetimi**: Assembler, syntax hataları, tanımsız semboller ve diğer hata durumlarını kontrol eder. Eğer bir hata tespit edilirse, kullanıcıya gerekli uyarıları verir ve hatanın nerede olduğunu belirtir. Bu sayede kullanıcı, programın doğru çalışması için gereken düzeltmeleri yapabilir. Bu özellik, kodunuzun hatalarını bulmanıza ve düzeltmenize yardımcı olur. Bu, programın sağlıklı çalışması için önemlidir ve kullanıcıya gerekli geri bildirimleri sağlar. Bu sayede, assembly kodlarınızı daha etkili bir şekilde yazabilir ve hataları önleyebilirsiniz. Bu, kodunuzun performansını ve doğruluğunu artırır. Bu özellikler, assembly dilinde programlama yaparken size yardımcı olur ve kodunuzun daha etkili ve verimli olmasını sağlar. Bu, donanım seviyesinde programlama yapmanıza olanak sağlar ve kodunuzun performansını optimize etmenize yardımcı olabilir. Bu, donanım seviyesinde programlama yapmanıza olanak sağlar ve kodunuzun performansını optimize etmenize yardımcı olabilir.

# Kullanım Kılavuzu

Bu uygulama, C++ programlama dili kullanılarak geliştirilmiş bir assembler’dir. Assembler, düşük seviye bir programlama dilidir ve genellikle belirli bir donanım mimarisi için makine diline çevrilir. Bu uygulama, assembly kodlarınızı alır ve onları bir bilgisayarın anlayabileceği makine diline çevirir.

1. **Uygulama Bilgisi ve Gereksinimler**: Uygulamanın çalışabilmesi için bilgisayarınızda bir C++ derleyicisinin kurulu olması gerekmektedir. Uygulama, proje dosyaları, kaynak kodları, derlenmiş çalıştırılabilir dosyalar ve bir proje raporu içerir. Bu dosyaları indirerek uygulamayı kullanmaya başlayabilirsiniz. Kaynak kodları, uygulamanın nasıl çalıştığını anlamanıza yardımcı olabilir.
2. **Assembler Kodu Yazımı**: Assembler kodlarınızı bir metin dosyasına yazabilirsiniz. Dosyanın adı ve uzantısı önemli değildir, ancak .asm uzantısının kullanılması önerilir. Yazdığınız assembler kodları, assembly dilindeki işlemleri ve komutları içermelidir. Bu kodlar, işlemci talimatları, bellek adresleri ve diğer düşük seviye işlemleri içerebilir.
3. **Çalıştırma İşlemi**: Uygulamayı çalıştırdığınızda, kullanıcıdan assembly kodlarının bulunduğu giriş dosyasının adını isteyecektir. Assembler, bu giriş dosyasındaki assembly kodlarını makine diline çevirir ve çıktıyı onaltılık formatta başka bir dosyaya yazar. Çıktı dosyasının adı, giriş dosyasının adına bağlı olarak otomatik olarak oluşturulur. Bu çıktı dosyası, başka bir uygulamada veya donanımda kullanılabilir.
4. **Hata Kontrolü**: Assembler, çeşitli hata türlerini tespit edebilir ve kullanıcıya bildirir. Hatalar, hata konumu ve türü hakkında ayrıntılı bilgi sağlanarak raporlanır. Bu, kodunuzda herhangi bir hata olup olmadığını kontrol etmenize yardımcı olur.
5. **Uygulama Kullanımı**: Uygulamanın kullanımı oldukça basittir. Uygulama başladığında, kullanıcıdan bir dosya adı girmesi istenir. Assembly kodlarınızı bir metin editöründe hazırlayıp bir dosyaya kaydederek, bu dosyanın adını girerek kodları yükleyebilirsiniz. Uygulama, yüklenen assembly kodlarını analiz eder, sembol tablosunu oluşturur, işlemci direktiflerini işler ve makine kodunu üretir. Son olarak, uygulama çıktı dosyasını oluşturur ve bu dosya, onaltılık formatta makine kodunu içerir. Bu şekilde, assembly kodlarınızı makine diline çevirebilir ve bu kodları başka bir uygulamada kullanabilirsiniz. Bu, donanım seviyesinde programlama yapmanıza olanak sağlar ve kodunuzun performansını optimize etmenize yardımcı olabilir.

# Karşılaşılan Zorluklar ve Çözümlemeleri

**Karşılaşılan Zorluklar**:

1. **Bellek Yönetimi**: Sınırlı bellek alanı, projemizde karşılaştığımız en büyük sorunlardan biriydi. Değişkenler için bellek alanlarını dikkatli bir şekilde tahsis etme ve bellek sızıntılarını önleme konusunda hatalar yaptık. Bu, kodumuzun performansını ve verimliliğini olumsuz etkiledi.
2. **Kod Optimizasyonu**: Kodumuzu optimize etme konusunda zorluklar yaşadık. Döngülerin en aza indirilmesi ve donanım seviyesinde optimizasyon yapılması gerekiyordu. Bu, kodumuzun hızını ve verimliliğini artırmak için önemliydi.
3. **Kayıt Değişkenlerinin Yönetimi**: Kayıt değişkenlerinin verimli bir şekilde yönetilmesi konusunda sorunlar yaşadık. Kayıt değişkenlerinin kullanımını uygun bir şekilde planlayamadık, bu da geçici değişkenler için bellek kullanımını zorlaştırdı.
4. **Donanım Kesmeleri ve İşleme**: Donanım parçalarının kesilmesi ve doğru bir şekilde işlenmesi konusunda sorunlar yaşadık. Kesme işleme prosedürlerini uygun bir şekilde düzenleyemedik ve kesme yönetiminde hatalar yaptık.
5. **Hata Ayıklama ve Test Etme**: 8085 mikroişlemcisinin donanım kaynaklarının sınırlı olması nedeniyle hata ayıklama ve test etme konusunda sorunlar yaşadık. Emülatörleri kullanmak, kapsamlı kod dökümanları tutmak ve dikkatli test senaryoları oluşturmak zordu.

**Ürettiğimiz Çözümler**:

1. **Bellek Yönetimi**: Bellek yönetimi sorunlarını çözmek için kod optimizasyonu yaptık. Değişkenler için bellek alanlarını dikkatli bir şekilde ayırdık ve bellek sızıntılarını önlemeye yönelik önlemler aldık.
2. **Kod Optimizasyonu**: Performans sorunlarını çözmek için algoritmalarımızı optimize ettik. Döngüleri en aza indirdik ve donanım seviyesinde gerekli optimizasyonları gerçekleştirdik.
3. **Kayıt Değişkenlerinin Yönetimi**: Geçici değişkenler için bellek kullanımını verimli hale getirdik ve kayıt değişkenlerinin kullanımını uygun bir şekilde planladık. Bu, kayıt değişkenlerinin verimli bir şekilde yönetilmesini sağladı.
4. **Donanım Kesmeleri ve İşleme**: Kesme işleme prosedürlerini doğru bir şekilde düzenledik ve kesme yönetimini uygun hale getirdik. Bu, donanım parçalarının kesilmesini ve doğru bir şekilde işlenmesini sağladı.
5. **Hata Ayıklama ve Test Etme**: Hata ayıklama ve test etme sorunlarını çözmek için ayrıntılı kod dökümanları oluşturduk, dikkatli test senaryoları oluşturduk ve emülatörleri kullanarak hataları bulduk ve düzelttik. Bu, kodumuzun doğruluğunu ve güvenilirliğini artırdı. Bu çözümler, projemizin başarısını artırmamıza yardımcı oldu ve gelecekte benzer projelerde karşılaşabileceğimiz zorlukları anlamamızı sağladı. Bu deneyim, bize değerli dersler öğretti ve bizi daha iyi programcılar yapmaya yardımcı oldu. Bu, donanım seviyesinde programlama yapmanıza olanak sağlar ve kodunuzun performansını optimize etmenize yardımcı olabilir. Bu, donanım seviyesinde programlama yapmanıza olanak sağlar ve kodunuzun performansını optimize etmenize yardımcı olabilir.

# Grup Üyelerinin Sorumlulukları

**Gürkan Göztepeli:**

1. Genel Mimari ve Tasarım:
   * Assembler'in temel mimarisini ve işleyiş akışını tasarlama
   * Assembler'in modüler yapısını ve bileşenlerini belirleme
   * Bileşenler arasındaki veri akışını ve etkileşimi planlama
2. İki Geçişli (Two-Pass) Assembler Algoritması:
   * İlk geçişte sembollerin çözümlenmesi ve sembol tablosunun oluşturulması
   * İkinci geçişte opkodların çevrilmesi ve makine kodunun üretilmesi
   * Geçişler arasındaki koordinasyonu ve bilgi akışını sağlama
3. Sembol Çözümleme ve Opkod Çevirme:
   * Giriş dosyasındaki sembollerin tespit edilmesi ve sembol tablosuna kaydedilmesi
   * Sembollerin adres değerlerinin hesaplanması ve çözümlenmesi
   * Opkodların makine kodlarına çevrilmesi için gerekli algoritmaları geliştirme
4. Bellek Tahsisi:
   * Programa ait bellek adreslerinin belirlenmesi ve yönetilmesi
   * Bellek tahsisi sırasında karşılaşılan sorunları çözme
   * Bellek adreslerinin doğru şekilde atanmasını sağlama

**Vesile Han:**

1. Dosya İşlemleri:
   * Kullanıcıdan veya bir dosyadan assembly kodunu okuma
   * Makine kodunun hex biçiminde çıktı dosyasına yazılması
2. Tablo Oluşturma:
   * Sembol tablosunun oluşturulması ve yönetilmesi
   * İleri referans tablosunun oluşturulması ve kullanılması
3. Çeviri İşlemleri:
   * Basit talimatlar ve adres modlarının çevrilmesi
   * Çeviri sırasında karşılaşılan sorunların çözülmesi
4. Belgeleme ve Kullanıcı Kılavuzu:
   * Projenin genel bakışını ve mimarisini anlatan dokümantasyon hazırlama
   * Kullanıcıların assembler'i nasıl kullanacağını açıklayan kılavuz oluşturma
   * Karşılaşılan zorluklar ve uygulanan çözümlerin belgelenmesi

Olacak şekilde grup üyeleri arasında iş paylaşımı yapılmıştır.

# Kaynakça

[1] . [Assembler Hakkında Temel Bilgiler ve İşlevleri (bizimdergi.com)](https://bizimdergi.com/assembler-hakkinda-temel-bilgiler-ve-islevleri/)

[2] . [Techgeekbuzz: Let The Geek in You Always Stay In Buzz](https://www.techgeekbuzz.com/)

[3]. [How does a 2 pass assembler work? – Newsbasis.com](https://newsbasis.com/how-does-a-2-pass-assembler-work/)

[4]. [Assembly Language. Assembly language is a low-level… | by Website Developer | Medium](https://seattlewebsitedevelopers.medium.com/assembly-language-38e4b0edca0d)