T.C.
SAKARYA UYGULAMALI BİLİMLER ÜNİVERSİTESİ
TEKNOLOJİ FAKÜLTESİ
BİLGİSAYAR MÜHENDİSLİĞİ

SİSTEM PROGRAMLAMA
Proje Ödevi

Adı Soyadı	: Ömer Umur, Sudenur KAZKONDU Alikan KÖSE, Enes DURAN
Bölümü/Programı	: Bilgisayar Mühendisliği
No	: B200109027, 22010903115, 23010903133, B200109040

1. MSP430 Mikrodenetleyicisi ve Assembly Programlama Hakkında Genel Bilgi

1.1. MSP430 Mimarisi

MSP430, Texas Instruments tarafından geliştirilen, düşük güç tüketimi ve yüksek verimlilikle öne çıkan 16-bitlik bir mikrodenetleyici ailesidir.

MSP430'un bazı temel mimari özellikleri şunlardır:

- **16-bit RISC mimarisi:** Basit ve hızlı komut yürütme süreci sunar.
- **Geniş adresleme modları desteği:** Farklı veri erişim biçimleri sağlar.
- **Flash tabanlı program hafızası:** Kullanıcı tarafından tekrar programlanabilir yapıdadır.

1.2. MSP430 Assembly Dili ve Adresleme Modları

Assembly dili, bir mikrodenetleyiciye en düşük seviyede doğrudan talimat vermemizi sağlayan programlama dilidir. MSP430'un RISC tabanlı yapısı sayesinde assembly komutları kısa, öz ve genellikle sabit uzunluktadır. Her komutun binary formatı, işlem kodu (opcode), adresleme modları ve operandlara göre belirlenir.

MSP430'da Kullanılan Adresleme Modları:

Adresleme Modu	**Sözdizimi**	**Ek Boyut (byte)**	**Toplam Komut Boyutu**
Immediate	#1234	+2 byte	4 byte
Register	R5	Yok	2 byte
Indirect	@R5	Yok	2 byte
Indirect Autoincrement	@R5+	Yok	2 byte
Indexed	offset(R5)	+2 byte	4 byte
Absolute	&1234	+2 byte	4 byte

Her adresleme modunun kullanımı, performans, bellek kullanımı ve kod okunabilirliği açısından farklı avantajlar sağlar. Örneğin:

- #value ile doğrudan değer yüklenir (immediate),
- R5 gibi doğrudan register erişimi daha hızlıdır ve kod boyutunu küçültür,
- @R5 dolaylı adresleme, bellekten veri okumayı sağlar,
- offset(R5) ile belirli bir uzaklıktaki adrese erişim yapılabilir.
- &1234 sözdizimin de "&" Talimatı takip eden kelime mutlak adresi içerir.

1.3. MSP430 ve Assembler İlişkisi

Assembler, assembly dilinde yazılmış komutları makine diline çeviren yazılımdır. MSP430 mimarisine özgü olarak tasarlanan bir assembler:

- Komut uzunluklarını ve adresleme modlarını göz önünde bulundurur,
- Her komutun opcode'unu ve operand yapısını MSP430 mimarisine uygun şekilde işler,
- Etiket (label) tanımlamalarını bellek adreslerine dönüştürerek program akışını sağlar.

Assembler'ların genellikle iki aşamalı (two-pass) çalıştığı görülür:

1. **Pass 1:** Tüm sembollerin (etiketlerin) adresleri belirlenir ve sembol tablosu (SYMTAB) oluşturulur.

2. **Pass 2:** Assembly komutları opcode'lara çevrilir, operand adresleri hesaplanır ve makine kodu (binary) üretilir.

2. MSP430 Assembler Programının Amacı

Bu proje kapsamında geliştirilen Python uygulamasının temel amacı, MSP430 mikrodenetleyicisi için assembly dilinde yazılmış komutları analiz ederek, bu komutların karşılık geldiği ikilik (binary) makine kodlarını üretmektir. Böylece düşük seviyeli yazılım geliştirme sürecinde önemli bir adım olan derleme süreci yazılım temelli olarak simüle edilmiş olur.

2.1. Problem Tanımı

MSP430 için yazılmış assembly kodları, doğrudan donanımın anlayabileceği formatta değildir. Bu kodların:

- Doğru bir şekilde çözümlenmesi (parsing),
- Etiketlerin doğru adreslerle eşleştirilmesi (label resolution),
- Komutların opcode ve operandlarına ayrıştırılması,
- Her komutun binary formatının üretilmesi

2.2. Çözüm Yaklaşımı

Hazırlanan Python uygulaması, bu sorunu çözmek amacıyla özel olarak geliştirilmiş olup, iki temel aşamada çalışır:

- **Pass 1 (Etiket Tanımlama ve Adresleme):** Assembly kodları satır satır işlenerek etiketler (labels) tespit edilir ve her etikete karşılık gelen bellek adresi kaydedilir.
- **Pass 2 (Makine Koduna Dönüştürme):** Belirlenen etiket adresleriyle birlikte komutlar ayrıştırılır, operand türleri analiz edilir ve her komutun ikilik (binary) karşılığı hesaplanarak çıktı üretilir.

2.3. Uygulamanın Sağladığı Faydalar

Bu yazılım sayesinde:

- Assembly kodları grafik arayüz (GUI) üzerinden girilebilir, düzenlenebilir ve test edilebilir,
- MSP430 işlemcisine özel olarak komut formatları göz önünde bulundurularak doğru binary çıktı elde edilebilir.

Uygulama, kullanıcı dostu bir arayüz (Tkinter GUI) ile donatılmıştır ve MOV, ADD, SUB, CMP, JMP, JEQ gibi temel MSP430 komutlarını desteklemektedir. Aynı zamanda etiketlerin tekrar tanımlanması gibi hataları da raporlar.

3. Kodun Çalışma Algoritması

Bu Python uygulaması, MSP430 assembly komutlarını makine koduna dönüştürmek amacıyla iki aşamalı (two-pass) bir algoritma kullanır. Bu yöntem, etiketlerin (label) bellek adresleri ile eşleştirilmesi ve komutların operand yapılarının analiz edilerek doğru binary çıktının üretilmesini sağlar.

3.1 .ASM → .OBJ → .COF Süreci

Uygulama temel olarak üç aşamada çalışır:

3.1.1 Derleme (Assembly)

`.asm` uzantılı kaynak dosyalar okunur ve her satır:

- Etiket (label) içeriyorsa etiket adresi hesaplanır,
- Komutlar opcode ve operandlarına ayrıştırılır,
- Doğru adresleme moduna göre komutun binary karşılığı üretilir.

Bu işlem sonunda her dosya için:

- `.text` (komutlar),
- `.data` (veriler),
- `EXPORTS` (diğer dosyalara sunulan semboller),
- `RELOCATIONS` (dış sembollerin adreslenmesi gereken yerler)

gibi alanlar içeren `.obj` uzantılı bir nesne dosyası oluşturulur.

3.1.2 Bağlama (Linking)

Birden fazla `.obj` dosyası alınıp:

- Her modüle bir base address atanır,
- Tüm `EXPORTS` ve dış referanslar (`relocs`) çözülür,
- Adresler yeniden hesaplanır ve binary içerik (global text/data) oluşturulur.

3.1.3 Çıktı Üretimi

Son olarak:

- Relocation işlemleri tamamlanmış,
- Global semboller birleştirilmiş,
- `.text` ve `.data` segmentleri birleşmiş bir şekilde

COFF_LINKED EXECUTABLE FILE başlığıyla .cof formatında çıktı yazılır.

3.2. Akış Şeması

1. **Kullanıcıdan Kod Alımı:**
   - GUI üzerinden kullanıcıdan assembly kodları alınır.
   - Yorum satırları temizlenir ve boş satırlar ayıklanır.

2. **Pass 1 – Etiket Tanıma ve Adres Atama:**
   - ORG direktifi varsa başlangıç adresi belirlenir.
   - Her satır analiz edilerek:
     - Etiket içeriyorsa label adı ve adresi kaydedilir.
     - Komutun operand yapısına göre adres artışı belirlenir:
       - Immediate (#1234), Absolute (&1234) veya Indexed (offset(Rx)) modlarında komut 4 byte yer kaplar.
       - Diğer modlarda komut 2 byte uzunluktadır.
   - Etiketler self.labels sözlüğüne kaydedilir.
   - Aynı etiketin birden fazla kez tanımlanması halinde kullanıcıya hata mesajı döner.

3. **Pass 2 – Binary Kod Üretimi:**
   - Tüm satırlar tekrar işlenir.

   - Varsa etiket isimleri, Pass 1'de hesaplanan adreslerle değiştirilir.
   - Her satırdaki komut ve operand yapısı analiz edilir:
     - Komut adı (MOV, ADD, JMP, vb.) tanımlı mı?
     - Operandlar register mı, immediate mi?
   - get_operand_binary_dual_operand() fonksiyonu ile operandlar ayrıştırılır ve uygun addressing mode'a göre binary formatı hazırlanır.
   - Komutun opcode değeri, register değerleri ve addressing mode bilgileri birleştirilerek tek bir binary string oluşturulur.
   - Her binary komut çıktısı bir listeye eklenir.
   - Hatalı ya da desteklenmeyen komut durumlarında kullanıcıya hata mesajı gösterilir.

4. **Sonuçların Görüntülenmesi:**
   - Pass 2 başarıyla tamamlandıysa, oluşan binary kod çıktısı GUI üzerinde kullanıcıya gösterilir.
   - Etiketlerin bulunduğu adres listesi GUI'nin durum

çubuğunda gösterilir.

3.3. Önemli Fonksiyonlar ve Görevleri

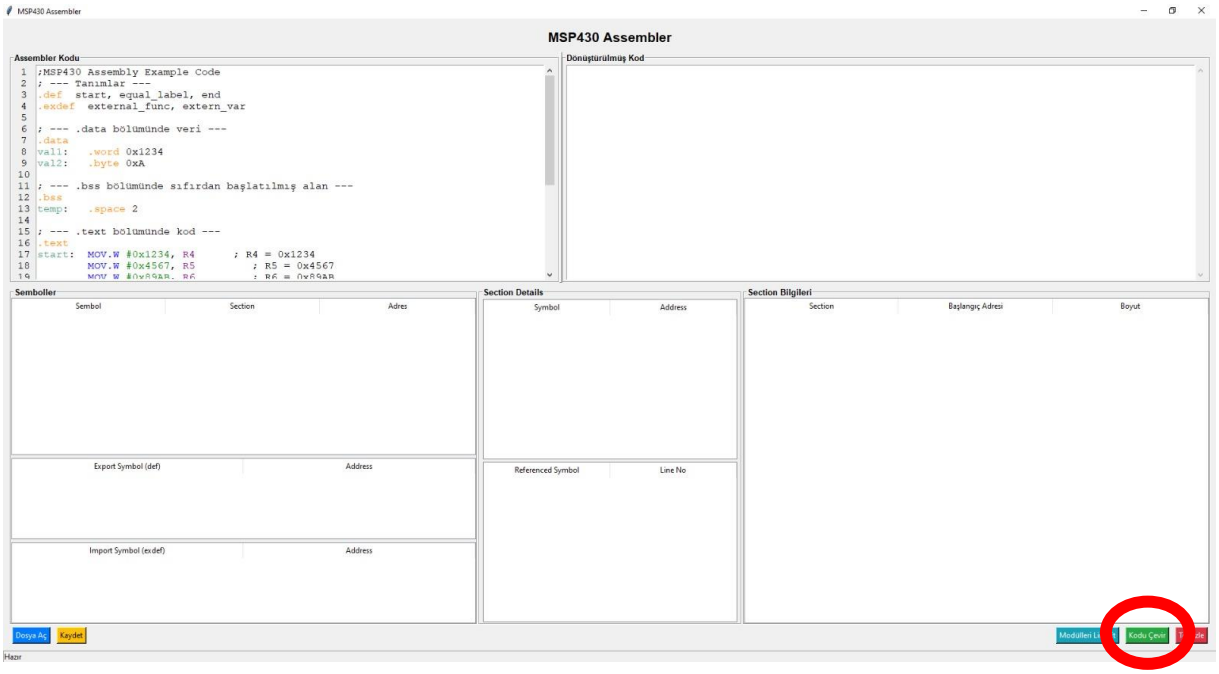
Fonksiyon	Açıklama
hexadec_to_binary(hex)	Hexadecimal sayıyı 16 bit binary'ye dönüştürür.
msp430_hex_addition(hex1, hex2)	İki hexadecimal sayıyı toplar, adres hesaplamada kullanılır.
pass1(lines)	Etiketleri belirler, adres hesaplar ve hatalı tanımlamaları kontrol eder.
pass2(lines)	Komutları binary formatına çevirir.
get_operand_binary_dual_operand()	İki operandlı komutların addressing mode'una göre binary karşılığını üretir.

4. Programa Ait Fotoğraflar

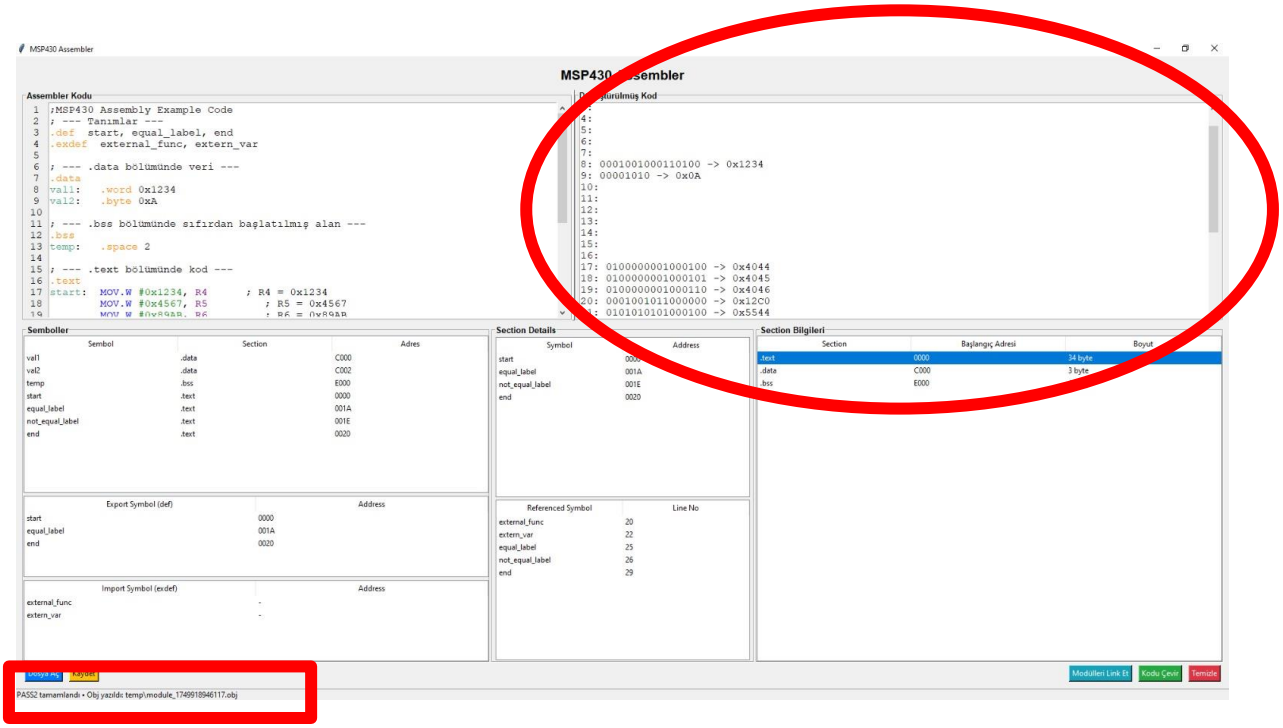
Geliştirilen Python tabanlı bu uygulama, MSP430 mikrodenetleyicisi için yazılan assembly komutlarını iki aşamada işleyerek makine diline dönüştürmektedir.

Kodun çekirdeğinde yer alan MSP430Assembler sınıfı, komutları opcode'lara ayırma, operand analizleri yapma ve her komutun ikilik karşılığını üretme görevlerini üstlenir. Etiketlerin yönetimi, adres hesaplamaları ve hata kontrolü bu sınıf içerisinde bütünleşik olarak sağlanır.

Arayüz tarafında ise Tkinter kütüphanesi ile hazırlanan kullanıcı dostu bir panel yer alır. Kullanıcılar buradan assembly kodlarını girip, çıktılarını anında görebilir, dosya açabilir veya kaydedebilirler. Uygulama, öğrenme amacıyla geliştirilmiş olup temel komutları (MOV, ADD, SUB, CMP, JMP, JEQ) destekler.

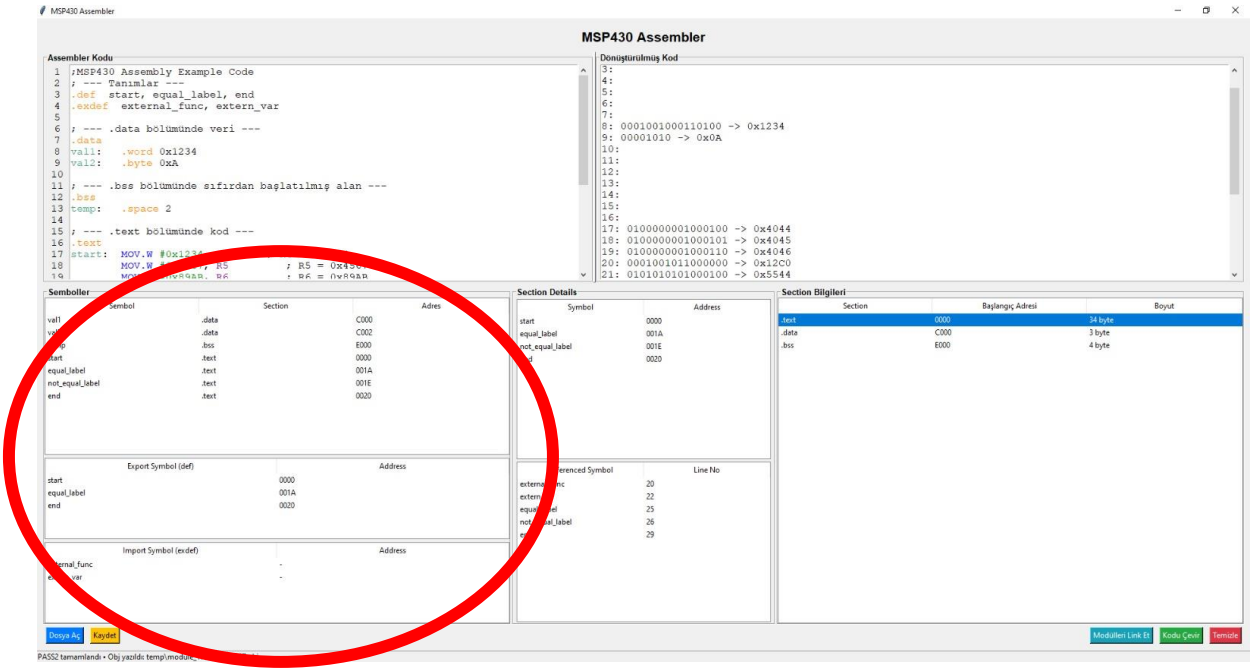


1. Bu fotoğrafta assembler'ımızın sol alt köşesinde hazır durumda olduğunu görüyoruz; bu, kodlarımızı eklemek ve işlem yapmak için sistemin hazır olduğunu gösterir. Programımızı sol üstte yer alan "Assembler Kodu" bölümüne yazıyoruz ve ardından "Kodu Çevir" butonuna tıklayarak derleme işlemini başlatıyoruz. Bu işlem sonucunda assembler, yazılan komutları analiz ederek dönüştürülmüş makine kodunu üretir ve aynı anda geçici klasör (temp) içine bir .obj dosyası oluşturur. Bu dosya, ileri aşamalarda bağlayıcı (linker) işlemleri için kullanılmak üzere kaydedilir.

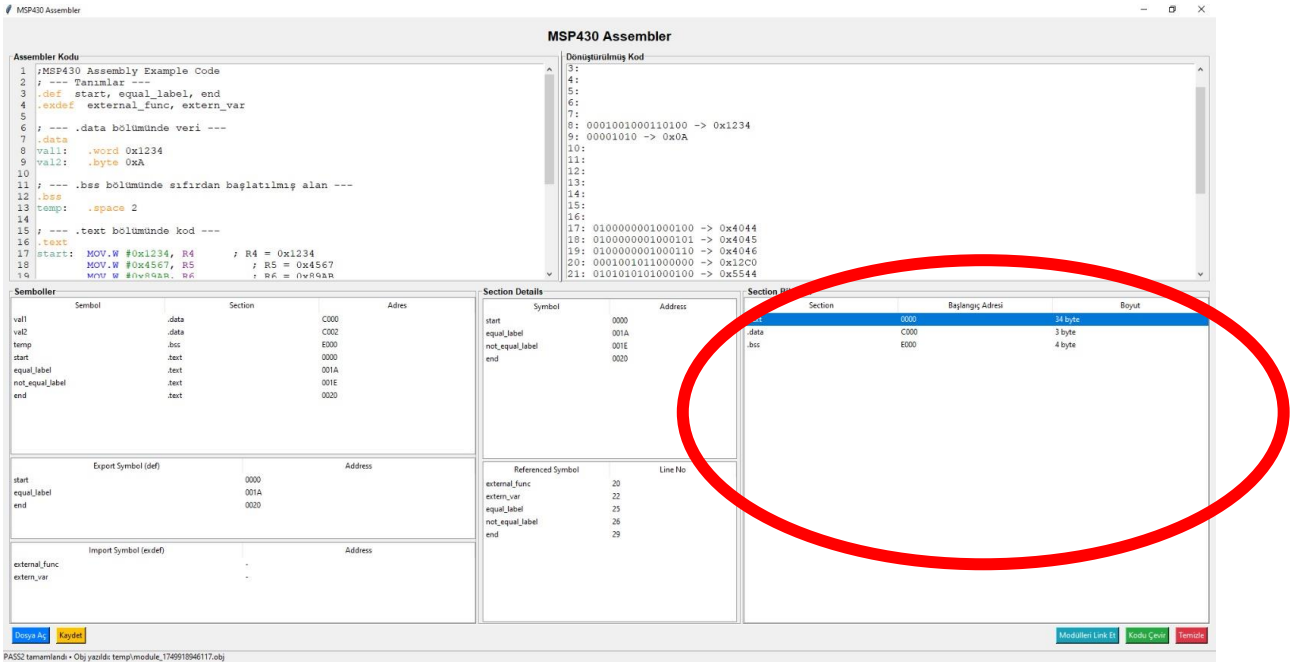


2. Kod çevrim işlemimiz gerçekleşiyor ve bize binary çıktısını

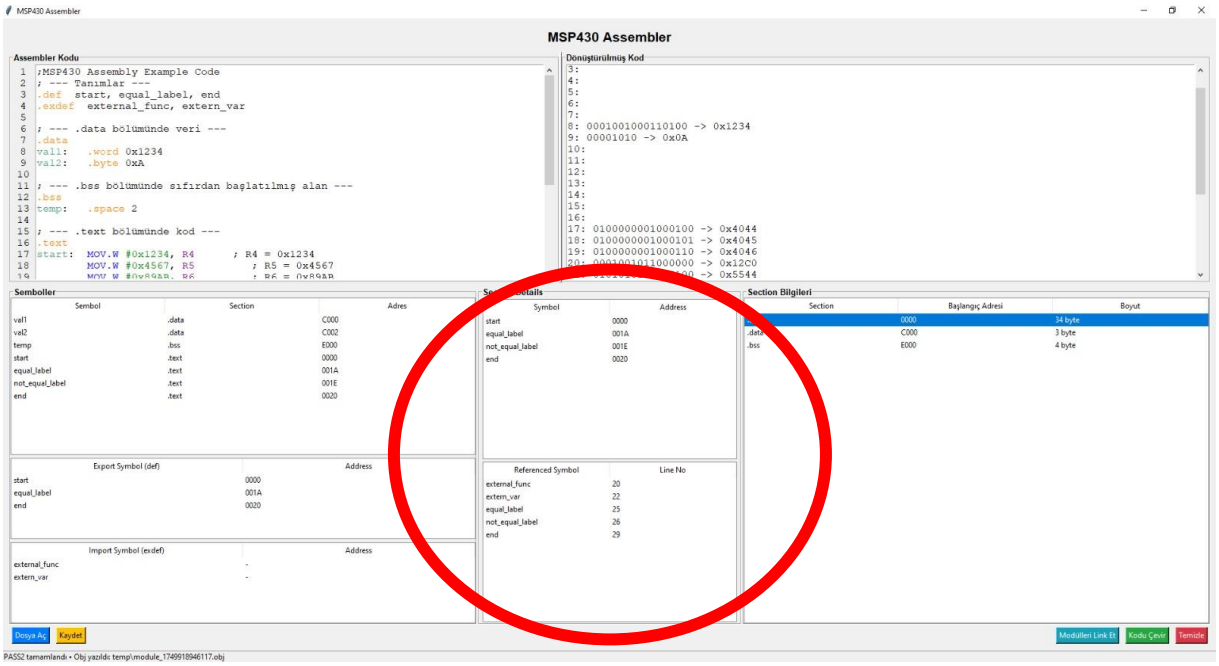
ekranın sağ bölümünde veriyor. Ekstra olarak assembler'ın sol alt bölümünde "PASS2 tamamlandı" yazısını görebiliyoruz.



3. Bu bölümde, derleyici tarafından tanımlanan sembollerin bilgileri yer almaktadır. Arayüzün ilgili kısmında "Sembol Adı", "Section Bilgisi" ve "Adres Bilgisi" başlıklarıyla listelenmiş şekilde görüntülenir. Sembol adı, kod içinde tanımlanan etiketleri (label) ifade eder. Section bilgisi, bu sembolün hangi bölümde (.text, .data vs.) yer aldığını gösterir. Adres bilgisi ise sembolün bellek içindeki konumunu belirtir. Ayrıca, güncellenen arayüz ile birlikte bu alana iki yeni tablo daha eklenmiştir: Export Symbol (def): .def direktifi ile tanımlanan ve başka modüller tarafından kullanılmak üzere dışa aktarılan sembolleri gösterir. Bu tabloda, dışa açık hale getirilen sembol adları ve bellek adresleri listelenir. Import Symbol (exdef): .exdef direktifi ile tanımlanan ve başka modüllerde tanımlı olan sembollere yapılan referansları gösterir. Bu tabloda dış modüllerden çağrılan semboller ve adres bilgileri yer alır. Henüz bu semboller birleştirilmediği için adres bilgileri boş olabilir.

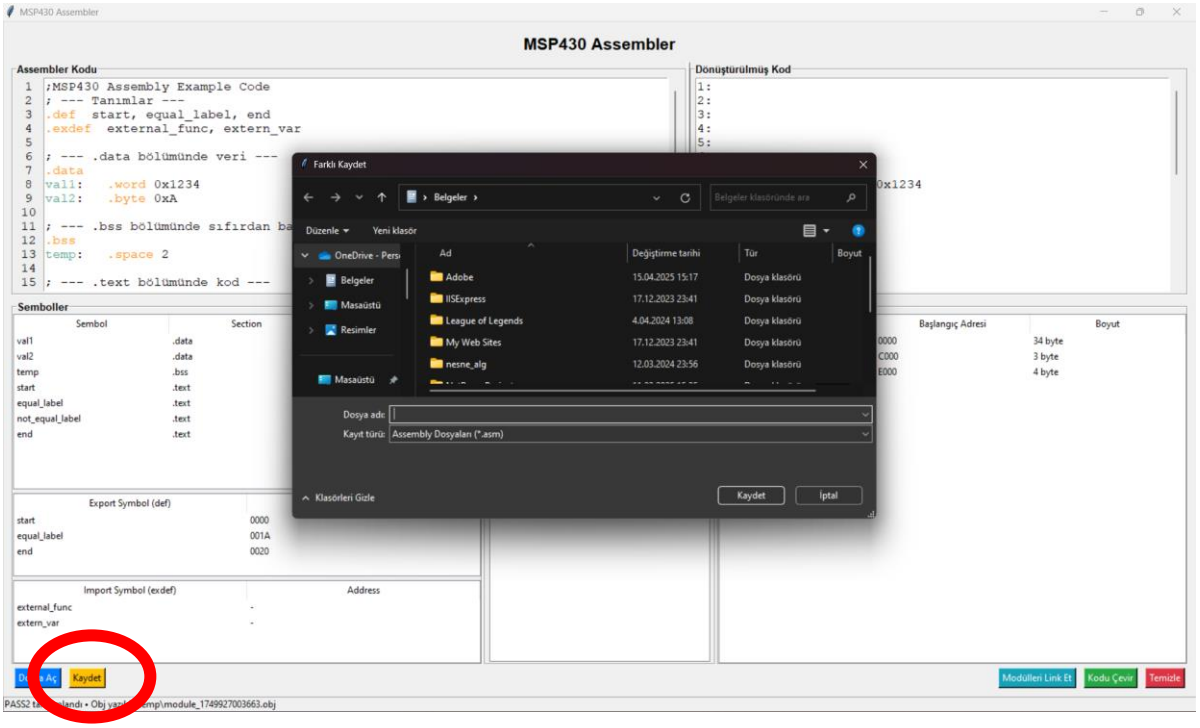


4. Bu bölümde, assembler tarafından oluşturulan her bir section'a (bölüme) ait temel bilgiler yer alır. Arayüzde **"Section Adı"**, **"Başlangıç Adresi" ve "Boyut"** başlıkları altında gösterilir. Section adı, örneğin .text veya .data gibi kod ve veri bölümlerini belirtir. Başlangıç adresi, section'ın hafızada başladığı adresi gösterir. Boyut bilgisi ise o section'a yazılan toplam veri miktarını (byte cinsinden) ifade eder.

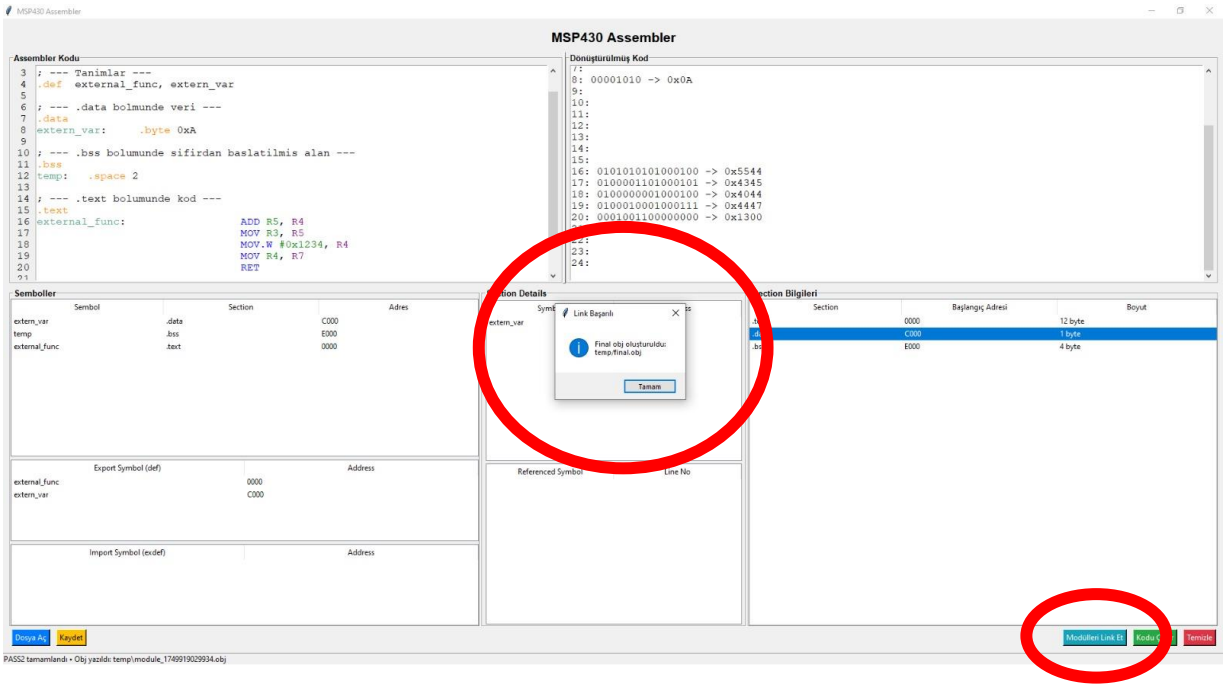


5. Bu kısımda, program içerisinde kullanılan ancak tanımları başka modüllerde yer alan semboller ile her bölümdeki tanımlı

etiketler bir arada gösterilmektedir. Referanslı semboller alanında, örneğin external_func gibi dış modüllerden çağrılan sembollerin programın hangi satırlarında kullanıldığı bilgisi yer alır; bu da çoklu modül yapılarında sembol takibini kolaylaştırır. Hemen yanında bulunan section detayları kısmında ise .text, .data ve .bss gibi farklı bölümlere ait sembollerin başlangıç adresleriyle birlikte hangi etikete karşılık geldiği listelenmektedir. Bu bilgiler sayesinde kod yapısı daha net anlaşılır ve sembollerin bellek içindeki konumları açıkça izlenebilir.



6. Assembler'ın sol alt köşesinde bulunan kaydet butonuna basarak da yazmış olduğumuz programı .asm uzantısı ile bilgisayarımıza kayıt edebiliyoruz.



7. Bu görselde, assembler arayüzünde yer alan "Modülleri Link Et" butonunu görüyoruz. Bu buton, daha önce oluşturulan .obj dosyalarının birleştirilerek çalıştırılabilir hale getirilmesini sağlar. Çok modüllü programlama yapılarında, birden fazla objenin bağlanması gerekebilir; bu işlem sayesinde .def ve .ref sembolleri eşleştirilir, dış referanslar çözülür ve bütünleşik bir bellek haritası oluşturulur. Kullanıcı bu butona tıklayarak, parçalı olarak derlenmiş modülleri tek bir programda birleştirebilir.

Kodlar:

main2.py

C: > Users > umuro > Downloads > omer_u > main2.py > LineNumberedText > __init__

```python
import tkinter as tk
from tkinter import scrolledtext, filedialog, messagebox

class LineNumberedText(tk.Frame):
    def __init__(self, parent, *args, **kwargs):
        tk.Frame.__init__(self, parent)
        self.text = scrolledtext.ScrolledText(self, *args, **kwargs)
        self.linenumbers = tk.Canvas(self, width=30, bg='#f0f0f0')

        self.linenumbers.pack(side=tk.LEFT, fill=tk.Y)
        self.text.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

        self.text.bind('<KeyRelease>', self.on_text_change)
        self.text.bind('<MouseWheel>', self.on_scroll)
        self.text.bind('<ButtonRelease-1>', self.on_text_change)

        self.text.config(yscrollcommand=self.on_text_scroll)
        self.textscroll = self.text.vbar  # scrolledtext'in scrollbar'ını al

        self.update_line_numbers()

    def on_text_scroll(self, *args):
        self.textscroll.set(*args)
        self.update_line_numbers()

    def on_scroll(self, event=None):
        self.update_line_numbers()

    def on_text_change(self, event=None):
        self.update_line_numbers()

    def update_line_numbers(self):
        self.linenumbers.delete("all")

        i = self.text.index("@0,0")
        while True:
            dline = self.text.dlineinfo(i)
            if dline is None:
                break
            y = dline[1]
            linenum = str(i).split(".")[0]
            self.linenumbers.create_text(15, y, anchor="n", text=linenum, font=self.text.cget("font"))
            i = self.text.index(f"{i}+1line")

class MSP430Assembler:
    def __init__(self):
        self.instructions = {
            "MOV":  "0100",
            "MOV.W":  "0100",
            "ADD":  "0101",
            "SUB":  "1000",
            "CMP":  "1001",
            "JMP":  "001111",
            "JEQ":  "001001",
        }
        self.registers = {
            "R0": "0000",
            "R1": "0001",
            "R2": "0010",
            "R3": "0011",
            "R4": "0100",
            "R5": "0101",
            "R6": "0110",
            "R7": "0111",
            "R8": "1000",
            "R9": "1001",
            "R10": "1010",
            "R11": "1011",
            "R12": "1100",
            "R13": "1101",
            "R14": "1110",
            "R15": "1111"
        }
        self.labels = {}

    def hexadec_to_binary(self, hexadec):
        """Hexadecimal sayıyı 16 bitlik bir binary sayıya dönüştürür."""
        binary = bin(int(hexadec, 16))[2:].zfill(16)
        return binary

    def binary_to_hex(self, binary):
        """Binary sayıyı hexadecimal formatına dönüştürür."""
        if len(binary) % 4 != 0:
            binary = binary.zfill((len(binary) // 4 + 1) * 4)
        hex_value = hex(int(binary, 2))[2:].upper().zfill(len(binary) // 4)
        return hex_value

    def msp430_hex_addition(self, hex1, hex2):
        dec1 = int(hex1, 16)
        dec2 = int(hex2, 16)
        total_dec = (dec1 + dec2) & 0xFFFF
        total_hex = format(total_dec, '04x')
```

Kullanıcı arayüzü oluşturmak için tkinter kütüphanesi kullanılır.

ScrolledText editörü oluşturulur. Sol tarafa bir Canvas yerleştirilir (satır numaraları burada gösterilir).Yazı değiştikçe veya kaydırıldıkça update_line_numbers() çağrılır.

Metnin her satırı için satır numarası hesaplanır. Her satır için Canvas üzerine metin çizilir (örneğin 1, 2, 3...).

MSP430 mimarisi için yazılmış assembly kodlarını binary makine koduna dönüştürür. self.instructions = Komutlar (MOV, ADD vb.) opcode karşılıklarıyla tanımlanır (örneğin MOV = 0100). self.registers = Register isimleri (R0 – R15) binary karşılıkları ile eşlenir.

Bu bölümde hexadecimal sayıyı binary'e ve binary sayıyı hexadecimal'e çevirme işlemleri için fonksiyonlar yer almaktadır.

```
        return total_hex

    def pass1(self, lines):
        address = "0000"
        duplicate_labels = set()
        if lines[0].startswith("ORG"):
            address = lines[0].split(" ")[1]
            lines = lines[1:]

        for line in lines:
            if ":" in line:
                label = line.split(":",1)[0]
                if label in self.labels:
                    duplicate_labels.add(label)
                self.labels[label] = address

            """
                Immediate mode (#1234): +2 byte ek (toplam 4 byte)
                Register mode (R5): Ek artış yok (toplam 2 byte)
                Indirect mode (@R5): Ek artış yok (toplam 2 byte)
                Indirect autoincrement (@R5+): Ek artış yok (toplam 2 byte)
                Indexed mode (offset(R5)): +2 byte ek (toplam 4 byte)
                Absolute mode (&1234): +2 byte ek (toplam 4 byte)
            """

            increment = "0002"  #varsayılan artış 1 word (2 byte)
            if any(x in line for x in ['#', '&', '(']):
                increment = "0004"  # 2 word (4 byte)
            address = self.msp430_hex_addition(address, increment)

        if duplicate_labels:
            error_msg = f"Error: Duplicate label definitions found: {', '.join(duplicate_labels)}"
            return error_msg

        return self.labels

    def get_operand_binary_dual_operand(self, operand1, operand2):
        """Operand1 analiz eder ve uygun binary kodunu döndürür"""
        try:

            # register mode: Rn
            if operand1 in self.registers and operand2 in self.registers:
                return self.registers[operand1], self.registers[operand2], "00" #2 byte

            # register mode (immediate): #1234
            if operand1.startswith("#"):
                value = operand1[1:]
                return self.registers["R3"], self.registers[operand2],"11",self.hexadec_to_binary(value) # 4 byte

            return "Desteklenmeyen Operand"
        except:
            return "Desteklenmeyen Operand"

    def pass2(self, lines):
        machine_code = []

        for line in lines:
            line = line.strip()
            if not line or line.startswith(";"):  # boş satır veya yorumları atla
                continue

            # etiketleri çıkart
            if ":" in line:
                line = line.split(":", 1)[1].strip()

            # etiketleri değiştir
            for label in self.labels:
                if label in line:
                    line = line.replace(label, self.labels[label])

            parts = line.split()
            if not parts:
                continue

            instruction = parts[0].upper()

            # çift operandlı komutlar
            if instruction in ["MOV", "MOV.W", "ADD", "SUB", "CMP"]:
                if len(parts) < 3:
                    return f"Error: Unsupported Command -> {line}"

                src, dst = parts[1].strip(","), parts[2].strip(",")

                operand_info = self.get_operand_binary_dual_operand(src,dst)
                if isinstance(operand_info, str): return operand_info
```

pass1, assembler kodunun ilk tarama aşamasıdır. Temel amacı, kod içinde tanımlanan tüm etiketleri (label:) bulmak ve her bir etikete karşılık gelen bellek adresini hesaplamaktır. Eğer ilk satır ORG ile başlıyorsa, programın başlangıç adresi bu değerden alınır. Satır satır ilerlenir, her satırda bir etiket varsa adresle birlikte labels sözlüğüne eklenir. Aynı etiket iki kez tanımlanmışsa hata verir. Ardından, satırın operand tipine göre komutun bellek boyutu (2 veya 4 byte) belirlenir ve adres buna göre artırılır. Bu işlem sonunda her etiketin bellek içindeki tam konumu belirlenmiş olur ve pass2 fonksiyonunda kullanılmak üzere geri döndürülür.

Assembly komutlarını binary formata çevirir. İkinci geçiştir (Pass 2).

Adımlar:

1. Satır boşsa ya da yorum ise geçilir.
2. Etiket varsa ayrıştırılır.
3. Etiket kullanımı varsa adresleri ile değiştirilir.
4. Komutun opcode değeri alınır.
5. Operandlar analiz edilir, binary kodu üretilir.

```
                opcode = self.instructions[instruction]
                ad = "0"
                bw = "1"
                if len(operand_info) == 3:
                    source, dest, ass = operand_info
                    binary_instruction = f"{opcode}{source}{ad}{bw}{ass}{dest}"
                if len(operand_info) == 4:
                    r3, dest, ass, immedite_value = operand_info
                    binary_instruction = f"{opcode}{r3}{ad}{bw}{ass}{dest}{immedite_value}"

                machine_code.append(binary_instruction)

            # tek operandlı komutlar (JMP, JEQ, NOP)
            elif instruction in ["JMP", "JEQ"]:
                print("...")

            else:
                raise ValueError(f"Error: Unknown instruction {instruction} in line -> {line}")

        return machine_code


class MSP430AssemblerUI:
    def __init__(self, root):
        self.root = root
        self.root.title("MSP430 Assembler")
        self.root.geometry("1200x700")

        self.main_frame = tk.Frame(root)
        self.main_frame.pack(fill=tk.BOTH, expand=True, padx=10, pady=10)

        self.title_label = tk.Label(self.main_frame, text="MSP430 Assembler", font=("Arial", 16, "bold"))
        self.title_label.pack(pady=5)

        self.split_frame = tk.PanedWindow(self.main_frame, orient=tk.HORIZONTAL)
        self.split_frame.pack(fill=tk.BOTH, expand=True, pady=5)

        self.left_frame = tk.LabelFrame(self.split_frame, text="Assembler Kodu", font=("Arial", 10, "bold"))

        self.right_frame = tk.LabelFrame(self.split_frame, text="Dönüştürülmüş Kod", font=("Arial", 10, "bold"))

        self.split_frame.add(self.left_frame)
        self.split_frame.add(self.right_frame)

        self.code_text = LineNumberedText(self.left_frame, wrap=tk.WORD, font=("Courier New", 12))
        self.code_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)
        self.add_example_code()

        self.result_text = scrolledtext.ScrolledText(self.right_frame, wrap=tk.WORD, font=("Courier New", 12))
        self.result_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        self.button_frame = tk.Frame(self.main_frame)
        self.button_frame.pack(fill=tk.X, pady=10)

        self.file_frame = tk.Frame(self.button_frame)
        self.file_frame.pack(side=tk.LEFT)

        self.load_button = tk.Button(self.file_frame, text="Dosya Aç", command=self.load_file,
                                     bg="#007bff", fg="white", font=("Arial", 10))
        self.load_button.pack(side=tk.LEFT, padx=5)

        self.save_button = tk.Button(self.file_frame, text="Kaydet", command=self.save_file,
                                     bg="#17a2b8", fg="white", font=("Arial", 10))
        self.save_button.pack(side=tk.LEFT, padx=5)

        self.action_frame = tk.Frame(self.button_frame)
        self.action_frame.pack(side=tk.RIGHT)

        self.clear_button = tk.Button(self.action_frame, text="Temizle", command=self.clear_code,
                                      bg="#dc3545", fg="white", font=("Arial", 10))
        self.clear_button.pack(side=tk.RIGHT, padx=5)

        self.convert_button = tk.Button(self.action_frame, text="Kodu Çevir",
                                        command=self.convert_code,
                                        bg="#28a745", fg="white", font=("Arial", 10, "bold"),
                                        padx=10, pady=2)
        self.convert_button.pack(side=tk.RIGHT, padx=5)

        self.status_bar = tk.Label(root, text="Hazır", bd=1, relief=tk.SUNKEN, anchor=tk.W)
        self.status_bar.pack(side=tk.BOTTOM, fill=tk.X)

    def add_example_code(self):
        example_code = """MOV #0x1234, R4   ; R4 = 0x1234
MOV.W #0x56, R5    ; R5 = 0x0056
ADD #1, R4         ; R4 = R4 + 1
SUB #2, R5         ; R5 = R5 - 2
CMP R4, R5         ; R4 ve R5'i karşılaştır
end: MOV R4, R5               ;"""
        self.code_text.text.insert(tk.END, example_code)
```

Kullanıcının kod yazabileceği, derleyebileceği ve sonucu görebileceği bir grafik arayüz sağlar. self.root.title ve tk.Frame bölümlerinde uygulama başlığı ve temel yerleşim ayarlanır. self.code_text ve self.result_text bölümleri kullanıcının kod yazdığı bölüm ve derlenmiş binary ve hex çıktısının gösterildiği bölümleri içerir. self.load_button, self.save_button, self.convert_button, self.clear_button bu bölümlerde dosya aç, kaydet, kodu çevir ve temizle gibi işlemleri yapılır.

Program başlarken test edilebilecek hazır assembly kodları ekler.

```
    def load_file(self):
        file_path = filedialog.askopenfilename(
            title="Assembler Dosyasını Aç",
            filetypes=[("Assembly Dosyaları", "*.asm"), ("Tüm Dosyalar", "*.*")]
        )
        if file_path:
            try:
                with open(file_path, 'r') as file:
                    content = file.read()
                    self.code_text.text.delete(1.0, tk.END)
                    self.code_text.text.insert(tk.END, content)
                self.status_bar.config(text=f"Dosya yüklendi: {file_path}")
            except Exception as e:
                messagebox.showerror("Hata", f"Dosya açılırken hata oluştu: {str(e)}")

    def save_file(self):
        file_path = filedialog.asksaveasfilename(
            title="Assembler Dosyasını Kaydet",
            filetypes=[("Assembly Dosyaları", "*.asm"), ("Tüm Dosyalar", "*.*")],
            defaultextension=".asm"
        )
        if file_path:
            try:
                with open(file_path, 'w') as file:
                    content = self.code_text.text.get(1.0, tk.END)
                    file.write(content)
                self.status_bar.config(text=f"Dosya kaydedildi: {file_path}")
            except Exception as e:
                messagebox.showerror("Hata", f"Dosya kaydedilirken hata oluştu: {str(e)}")

    def clear_code(self):
        self.code_text.text.delete(1.0, tk.END)
        self.result_text.delete(1.0, tk.END)
        self.status_bar.config(text="Hazır")

    def convert_code(self):
        code = self.code_text.text.get(1.0, tk.END).strip()
        if not code:
            messagebox.showwarning("Uyarı", "Lütfen çevrilecek assembler kodunu giriniz.")
            return

        assembler = MSP430Assembler()
        lines = code.split('\n')
        cleaned_lines = []
        for line in lines:
            if ';' in line:
                line = line.split(';')[0]
            line = line.strip()
            if line:
                cleaned_lines.append(line)

        pass1_result = assembler.pass1(cleaned_lines)
        if isinstance(pass1_result, dict):
            result = pass1_result
            self.status_bar.config(text=f"PASS1 Başarılı : {len(result)} etiket bulundu. -> {', '.join(result.keys())}")
            try:
                machine_code = assembler.pass2(cleaned_lines)
                if isinstance(machine_code, list):
                    self.result_text.delete(1.0, tk.END)

                    for i, binary_code in enumerate(machine_code):
                        hex_code = assembler.binary_to_hex(binary_code)
                        self.result_text.insert(tk.END, f"{binary_code} -> 0x{hex_code}\n")

                    self.status_bar.config(text="PASS2 Başarılı: Assembler kodu başarıyla çevrildi. Etiketler -> " + ',
                else:
                    messagebox.showerror("Hata", machine_code)
            except Exception as e:
                messagebox.showerror("Hata", f"{str(e)}")
        else:
            messagebox.showerror("Hata", pass1_result)


if __name__ == "__main__":
    root = tk.Tk()
    app = MSP430AssemblerUI(root)
    root.mainloop()
```

Assembler'a .asm uzantılı dosya yükleme fonksiyonudur.

Assembler'da yazılan assembly kodlarını .asm uzantılı dosya olarak bilgisayara kaydetme fonksiyonu.

Assembler'daki kodu temizleme fonksiyonudur.

Bu bölümde Assembler'a yazılmış olan kodun boş olup olmadığı kontrol edilir. Eğer boş değilse yazılmış olan kod satırlar halinde bölünür ve bir listeye temiz kod halinde eklenir. Sonrasında sırasıyla pass1 ve pass2 olarak işlemlere tabi tutulur ve hex koda çevrim işlemi gerçekleştirilir.

Ana Tkinter döngüsünü başlatır ve uygulama penceresini gösterir.

Jump (Sıçrama) Komutları Nasıl Çalışıyor?

Assembler'da bazı komutlar vardır ki programın sıradaki komutu çalıştırmasını engelleyip başka bir yere sıçramasını" (jump) sağlar. Bu, genelde koşullu dallanmalarda (if gibi) ya da döngülerde (loop gibi) kullanılır.

Bu komutlar farklı türde sıçramaları temsil eder:

Komut	Açıklama
JMP	Her zaman sıçrar (koşulsuz).
JEQ	Eğer eşitse sıçrar.
JNE	Eğer eşit değilse sıçrar.
JC	Eğer "carry" bayrağı varsa sıçrar.
JN	Eğer negatifse sıçrar.
JNC	Eğer "carry" yoksa sıçrar.
JGE	Eğer büyük ya da eşitse sıçrar.
JL	Eğer küçükse sıçrar.

Bu blok, JMP, JEQ, JNE, JC, JN, JNC, JGE, JL gibi koşullu ve koşulsuz sıçrama (jump) komutlarını işler.

```
# Jump komutları
elif instruction in ["JMP", "JEQ", "JNE", "JC", "JN", "JNC", "JGE", "JL"]:
    if len(parts) < 2:
        raise Exception(f"Error: Missing jump target in line -> {line}")
    opcode = self.instructions[instruction]
    target_label = parts[1]
    if target_label not in self.labels:
        raise Exception(f"Error: Undefined label {target_label} in line -> {line}")
    current_address = int(line_addresses[text_code_line_index], 16)
    dest_address = int(self.labels[target_label][1], 16)
    next_address = current_address + 2
    offset = (dest_address - next_address) // 2
    offset = offset & 0x3FF
    offset_bin = bin(offset)[2:].zfill(10)
    binary_instruction = f"{opcode}{offset_bin}"
    section_codes[".text"].append(binary_instruction)
    text_code_line_index += 1
```

Offset, sıçrama yapılacak komutun şu anki komuta göre uzaklığını (komut cinsinden) belirtir. Offset değeri, 10-bit genişliğinde 2'nin tümleyenine göre hesaplanır. Bu, bazı mimarilerde -512 ile +511 arasında bir sıçrama aralığı sağlar.
& 0x3FF ile 10 bit'lik bir maskeleme uygulanır. Offset, ikilik (binary) biçime çevrilip 10-bit olacak şekilde doldurulur.

Son olarak komutun opcode'u ile hesaplanan offset birleştirilerek 16 bitlik makine komutu oluşturulur. Bu komut, .text segmentine eklenir.

MSP430'da Section ve Kontrol Yapısı

MSP430 mimarisinde bir programın bellek yerleşimi, assembler seviyesinde tanımlanan **section** yapıları üzerinden düzenlenir. Bu yapılar sayesinde, kod ve veriler fiziksel bellekte organize bir şekilde yer alır. MSP430, özellikle gömülü sistemlerde düşük seviyeli kontrol sunması nedeniyle, bu yapıları

oldukça açık bir biçimde destekler.

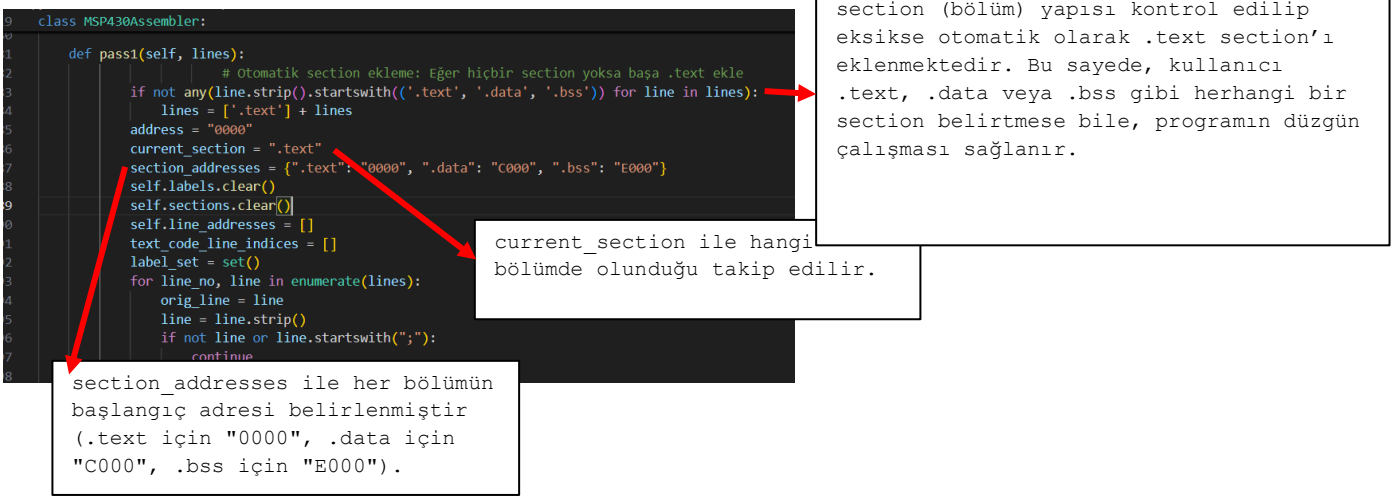
Program yazarken en sık karşılaşılan section'lar şunlardır:

Section Adı	Açıklama
.text	Programın yürütülebilir komutlarının tutulduğu bölümdür. Derleyici veya assembler tarafından oluşturulan kod, bu section'a yerleştirilir.
.data	Başlangıç değeri atanmış global veya static değişkenler için kullanılır. Programın çalışması sırasında RAM'de yer alır ve başlangıçta belirli değerlerle doldurulur.
.bss	Başlangıç değeri olmayan global veya static değişkenler burada yer alır. Program başlatılırken bu değişkenler sıfırla başlatılır. RAM'de yer alır ancak program dosyasına fiziksel olarak yazılmaz.
.const	Değiştirilemeyen sabit veriler burada tutulur. ROM gibi salt-okunur bellek alanına yerleştirilir. Örneğin sabit diziler veya const anahtar kelimesiyle tanımlanmış değişkenler bu section'a girer.
.stack	Programın çalışma zamanı sırasında fonksiyon çağrıları, yerel değişkenler ve geri dönüş adresleri için kullanılan yığın (stack) alanıdır. Bu section, özel olarak yığın işlemleri için ayrılmıştır ve genellikle bellek haritasının sonuna yakın konumlandırılır.
.usect	Kullanıcı tarafından tanımlanmış, başlatılmamış verileri içeren section'dır. .bss section'ına benzer ancak geliştiriciye daha fazla kontrol sağlar. Özellikle, bellek haritasının özel bir yerine konumlandırılmak istenen başlatılmamış veriler için kullanılır.
.sect"isim"	Geliştirici tarafından istenen bir isimle özel bir section oluşturulabilir. Bu şekilde belirli veri veya kod bölümleri bellekte ayrılmış özel alanlarda tutulabilir. Örneğin sect ".myCode" gibi bir tanımlama ile bu koda özel bir yer ayrılabilir.

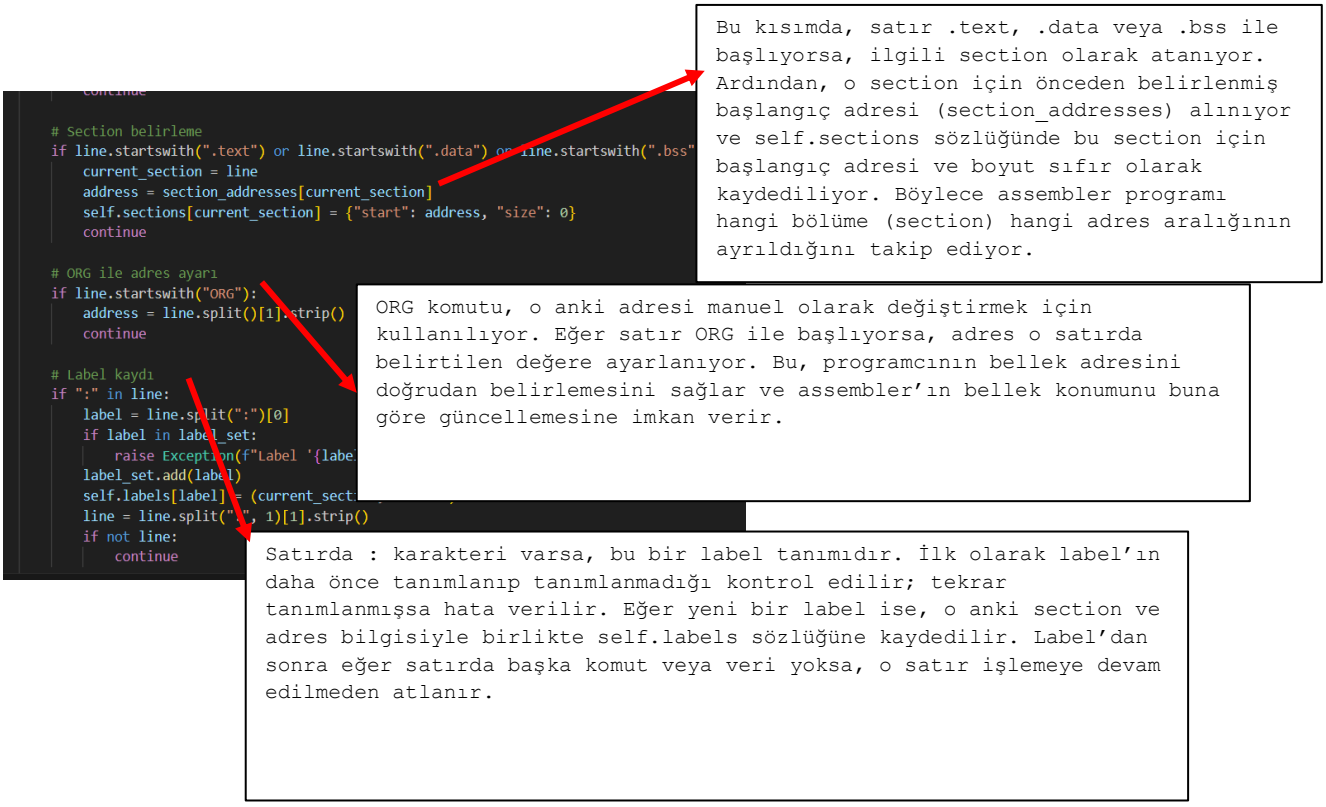
MSP430'da doğrudan bir **control section** (CSECT) kavramı yer almaz. Ancak assembler üzerinden farklı section'lar tanımlanarak, programın belli bölümleri ayrı tutulabilir. Örneğin, birden fazla **.sect** kullanılarak kod modülleri ayrıştırılabilir ve bellekte istenen yerlere yerleştirilebilir.

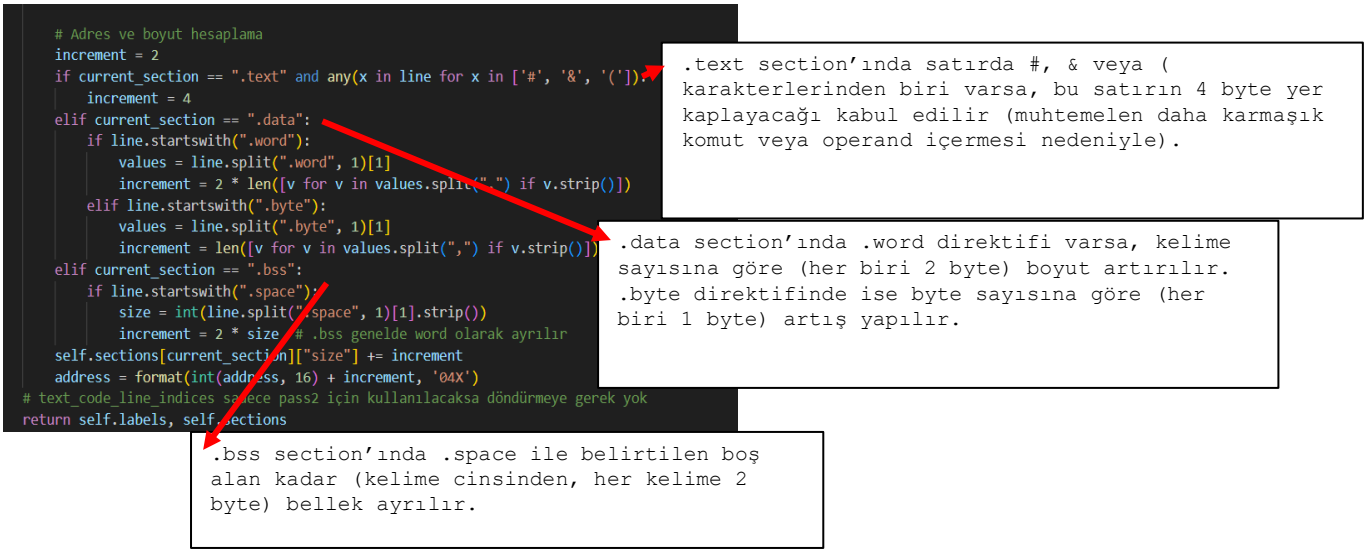
Bu yapı, ilk bakışta basit bir section yerleşimi gibi görünse de, aslında daha önce çalışılmış olan SIC mimarisiyle karşılaştırıldığında ilginç bir benzerlik içerir. SIC'te açıkça tanımlanan blok yapısı —her biri bağımsız işlenebilen ve bağlanabilen control section'lar— MSP430'da doğrudan bir blok kavramı olarak yer almaz. Fakat **section yapıları aracılığıyla bu blok mantığının işlevsel karşılığı** elde edilir. Yani MSP430, SIC'te ayrı bloklar hâlinde organize edilen kod yapısını, kendi içinde section'lara bölerek dolaylı şekilde uygular. Bu yönüyle, MSP430'da **blok ve section kavramları harmanlanmış bir biçimde** ortaya çıkar.

Sonuç olarak, MSP430'un assembler düzeyindeki section tanımı, SIC'teki modüler blok yapısının daha düşük seviyeli ve donanım-yakın bir karşılığı olarak değerlendirilebilir.



İşlenen her satırın boyutuna göre adres güncellenir ve section'un toplam boyutu hesaplanır; bu sayede ileride bellek tahsisi ve linkleme işlemleri sorunsuz gerçekleşir. Ayrıca, adresler standart dört haneli büyük harfli hexadecimal formatında tutulur. Section değişimi olduğunda adres, ilgili section'ın başlangıç adresi olarak yeniden ayarlanır ve label'lar hangi section içinde olduklarıyla birlikte kaydedilir. Bu işlemler, kodun doğru parçalanması, adreslenmesi ve sonrasında hatasız çalışması için vazgeçilmezdir.





Literal Kullanımı

MSP430 mimarisinde *literal* ya da sabit değer kullanımı, işlemcinin doğrudan komut içerisinde sabit bir sayıyla işlem yapmasına olanak tanır. Bu değerler, **#** sembolüyle tanımlanır ve **immediate addressing** yöntemiyle doğrudan işlenir. Kodun okunabilirliğini artırır ve veriye hızlı erişim sağlar.

assembly

```
MOV #25, R6      ; R6 register'ına 25 sayısı atanır
ADD #1, R5       ; R5'e 1 eklenir
CMP #0xFF, R7    ; R7 içeriği 0xFF ile karşılaştırılır
```

Bu tür kullanımda sabit değer, bellekte bir konumdan çağrılmaz; işlemci komutu çözümlerken literal değeri doğrudan komutun bir parçası olarak işler. Bu yaklaşım, bazı mimarilerde görülen *literal pool* gibi özel yapıların kullanımına duyulan gereksinimi ortadan kaldırır. Yani sabit değer için ayrı bir bellek alanına gitmeden işlem yapılabilir. Böylece hem bellek yönetimi sadeleşir hem de yürütme süresi kısalır.

Bazı durumlarda, kullanılan sabit değer çok büyükse veya tekrar eden sabitler varsa, derleyici bu verileri arka planda belirli sabit veri bölümlerine yerleştirir. Örneğin **.const** section'ları bu amaçla kullanılır. Ancak bu durum, geliştirici açısından kod yazımında ek bir yük oluşturmaz; derleyici uygun optimizasyonu yapar.

.def ve .ref Direktiflerinin MSP430 Projesindeki Kullanımı

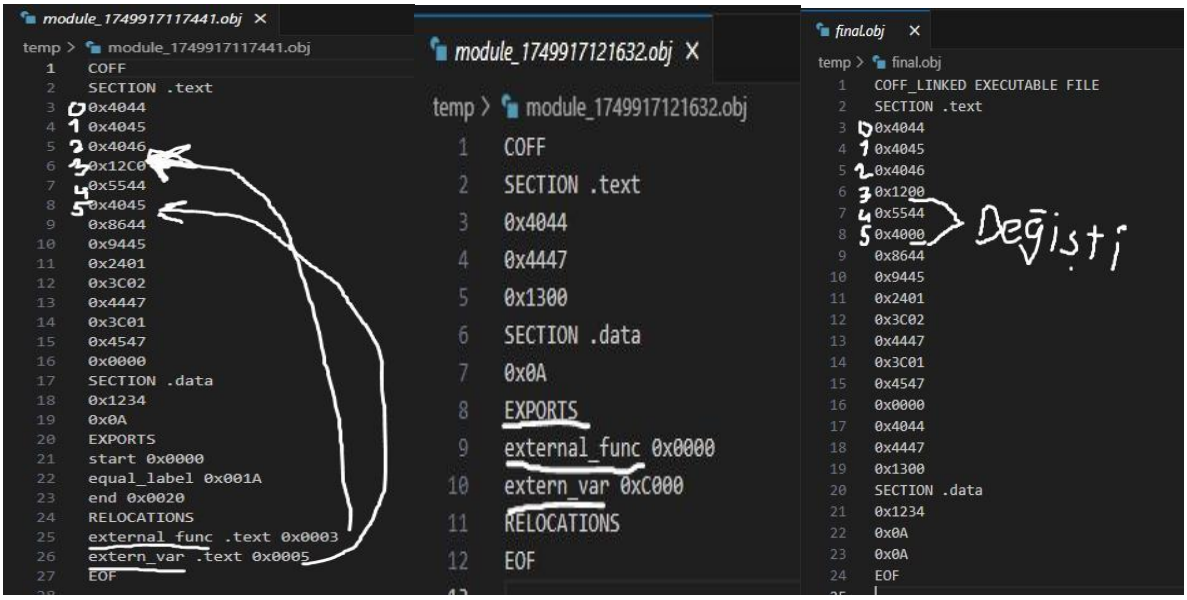
Geliştirdiğimiz MSP430 assembler projesinde, çok modüllü program yapısını desteklemek amacıyla **.def** (export definition) ve **.ref** (external reference) direktiflerine yer verilmiştir. Bu direktifler, farklı dosyalarda tanımlanmış sembollerin birbirine bağlanmasını sağlamakta ve modüler yazılım geliştirme

sürecini mümkün kılmaktadır.

.def, bir modülde tanımlanan sembolün diğer modüller tarafından erişilebilir olmasını sağlar. Örneğin, birinci modülde yazılmış bir fonksiyon ya da etiket, **.def** ile dışa açıldığında, ikinci bir modülde bu sembole doğrudan erişim sağlanabilir. Bu sayede, her sembolün tek bir yerde tanımlanması ve gerektiğinde başka kod parçaları tarafından kullanılabilmesi mümkün hale gelir.

Benzer şekilde **.ref**, modül içerisinde tanımı yapılmamış ancak başka bir modülde yer alan sembollere referans verilmesini sağlar. Bu direktif sayesinde assembler, sembolün tanımının harici olduğunu anlayarak bağlayıcı (linker) aşamasında doğru eşleştirme yapılmasına olanak tanır.

Projemizde bu yapıyı doğrudan desteklemek için hem arayüz hem de backend kısmında gerekli düzenlemeler yapılmıştır. Kullanıcılar, **.def** ve **.ref** ifadelerini kullandığında, bu semboller arayüzde ayrı bir tablo altında "Export Symbols (def)" ve "Import Symbols (ref)" başlıklarıyla listelenmektedir. Böylece kullanıcı hangi sembollerin dışa açıldığını veya dışarıdan çağrıldığını arayüzden takip edebilmektedir. Ayrıca "Modülleri Link Et" butonu kullanıldığında, obj dosyaları birleştirilerek bu semboller doğru adreslerle eşleştirilir.



Uygulama içerisinde bu yapı test edilmiştir. Örneğin bir **.asm** dosyasında **start** etiketi tanımlanıp **.def start** ile dışa açılmış, diğer bir modülde ise **.ref start** ile bu sembole çağrı yapılmıştır. Derleme sonrası obj dosyaları oluşturulmuş ve linker aracılığıyla semboller başarılı şekilde eşleştirilmiştir. Bu, assembler'ımızın çok dosyalı projelerde de kullanılabilir olduğunu ve sembol yönetimini desteklediğini göstermektedir.

Bu yapı yalnızca doğru sembol eşleşmesini sağlamakla kalmamış, aynı zamanda programın yapısını daha okunabilir ve sürdürülebilir hale getirmiştir.

MSP430 Assembler Uygulamasında Linking (Bağlama) Süreci

Geliştirdiğimiz MSP430 assembler uygulamasında, çok modüllü programların tek bir bütün haline getirilebilmesi için **linking (bağlama)** işlemi uygulanmaktadır. Linking işlemi, farklı kaynak dosyalardan derlenmiş **.obj**

dosyalarının bir araya getirilerek sembollerin çözümlenmesi ve tek bir çıktı haline dönüştürülmesini sağlar.

Uygulama mantığı şu adımlarla işlemektedir:

1. **Kodun Derlenmesi:** Kullanıcı bir modülün assembly kodunu yazıp "Kodu Çevir" butonuna tıkladığında, sistem bu kodu analiz eder ve geçici klasör (temp) içinde bir **.obj** dosyası oluşturur. Bu dosya içerisinde ilgili komutlar, semboller ve gerekli meta bilgiler (örneğin section bilgileri, relocatible adresler) yer alır.

2. **İkinci Modülün Eklenmesi:** Ardından, kullanıcı "Dosya Aç" kısmından farklı bir **.obj** dosyasını sisteme yükleyebilir. Bu dosya da aynı şekilde "Kodu Çevir" işlemine tabi tutulduğunda, temp klasöründe ikinci bir **.obj** dosyası daha oluşturulur.

3. **Linkleme İşlemi:** Temp klasörü içerisinde artık birden fazla **.obj** dosyası bulunmaktadır. Kullanıcı "Modülleri Link Et" butonuna tıkladığında, uygulama bu dosyaları bir araya getirerek final bir **.obj** dosyası oluşturur. Bu aşamada:

    a. **.ref** ile belirtilen dış semboller, karşılık gelen **.def** tanımları ile eşleştirilir.

    b. Her modülün section'ları bellekte uygun adreslere yerleştirilir.

    c. Relocation yapılması gereken tüm adresler güncellenir.

    d. Sonuç olarak, tüm kod parçaları birleşmiş, referansları çözülmüş ve çalışmaya hazır hale getirilmiş olur.

Oluşan final **.obj** dosyası, artık dışa bağımlı sembol içermeyen, tek parça halinde bir assembler çıktısıdır. Bu çıktı hem sembol bütünlüğünü sağlar hem de büyük projelerde modülerliği destekler.

Bu mimari, MSP430 assembler projemizi gerçek dünyadaki derleyici ve linker mantığına uygun hale getirmiş, kullanıcıların birden fazla modülden oluşan sistematik projeler geliştirebilmesini mümkün kılmıştır.

```python
def _load_modules(self):
    for fn in os.listdir(self.obj_dir):
        if not fn.endswith(".obj"): continue
        path = os.path.join(self.obj_dir, fn)
        self.modules.append(self._parse_obj(path))

    # önce tüm export'ları topluyoruz
    for m in self.modules:
        for sym, addr in m["exports"].items():
            if addr is None:
                raise Exception(f"Undefined exported symbol {sym}")
            if sym in self.global_exports:
                raise Exception(f"Duplicate export {sym}")
            # global_exports'de tutulacak adresi modülün kendi .text + base'e göre hesaplayacağız
            self.global_exports[sym] = (m, addr)
```

Burada, klasördeki tüm .obj dosyaları tek tek okunur ve _parse_obj() fonksiyonuyla işlenir. Daha sonra her modülde bulunan export semboller toplanarak global_exports sözlüğüne eklenir. Böylece, tüm dış erişilebilir semboller merkezi olarak saklanır.

```
            if section == "text":
                val = int(ln, 16)
                text.append(format(val, '016b'))
            elif section == "data":
                data.append(ln)
            elif section == "exports":
                # “sym addr”
                parts=ln.split()
                exports[parts[0]] = int(parts[1],16)
            elif section == "relocs":
                # “sym .text 0x0010”
                sym, sec, off = ln.split()
                relocs.append((sym, sec, int(off,16)))
    return {"text":text, "data":data, "exports":exports, "relocs":relocs}

def link(self):
    # her modülde text segmente bir base adres ata
    txt_base_idx = 0
    dat_base_idx = 0
    layout = []
    for m in self.modules:
        m["txt_base_idx"] = txt_base_idx
        m["dat_base_idx"] = dat_base_idx
        txt_base_idx += len(m["text"])
        dat_base_idx += len(m["data"])
        layout.append(m)

    # global text/data birleştir
    for m in layout:
        self.global_text.extend(m["text"])
        self.global_data.extend(m["data"])
```

Bu bölümde, .obj dosyasındaki içerikler ilgili listelere ekleniyor. .text kısmı binary'e çevrilerek text[] listesine, .data doğrudan data[] listesine, EXPORTS ve RELOCATIONS ise sözlük ve tuple olarak kayıt altına alınıyor.

Linkleme işlemi başlıyor. Her modül için .text ve .data segmentlerine birer başlangıç adresi atanıyor. Modüller arka arkaya gelecek şekilde yerleştiriliyor ve global_text ile global_data birleşik belleğe yazılıyor.

```
    # global text/data birleştir
    for m in layout:
        self.global_text.extend(m["text"])
        self.global_data.extend(m["data"])

    # relocation: her modülde kendi base’e göre patch et
    for m in layout:
        for sym, sec, inst_idx in m["relocs"]:
            if sym not in self.global_exports:
                raise Exception(f"Unresolved extern: {sym}")
            mod, sym_addr = self.global_exports[sym]
            # global_text’e gömülecek index:
            base_idx = m["txt_base_idx"]   # aşağıda def edeceğiz
            idx      = base_idx + inst_idx

            old_bin = self.global_text[idx]       # örn. "0001001011000000" == 0x12C0
            # high bits (üst 8 bit) koru:
            high = old_bin[:8]                    # "00010010" == 0x12
            # düşük baytı yeni adresin alt byte’ı ile değiştir
            low  = format(sym_addr & 0xFF, '08b') # eğer sym_addr=0x000A -> "00001010"
            self.global_text[idx] = high + low    # "00010010"+"00001010" == "0001001000001010" == 0x120A
```

Relocation işlemi bu kısımda yapılır. Komut içinde başka modülde tanımlı bir sembole erişilmek isteniyorsa, bu sembolün adresi komuta gömülür. Komutun alt baytı, sembol adresiyle değiştirilir, üst baytı ise korunur.

```python
def write(self, path):
    with open(path, "w") as f:
        f.write("COFF_LINKED EXECUTABLE FILE\n")
        f.write("SECTION .text\n")
        for b in self.global_text:
            h = hex(int(b,2))[2:].upper().zfill(len(b)//4)
            f.write(f"0x{h}\n")
        f.write("SECTION .data\n")
        for d in self.global_data:
            # eğer data'yı binary olarak parse etmediyseniz, d zaten "0x12A4" formundaysa doğrudan:
            if re.match(r'^[01]{8,16}$', d):
                h = hex(int(d,2))[2:].upper().zfill(len(d)//4)
                f.write(f"0x{h}\n")
            else:
                f.write(f"{d}\n")
        f.write("EOF\n")
```

(variable) d: Any

Bu kısımda, relocation işlemleri yapılmış olan .text ve .data bölümleri, final .obj dosyasına yazılır. Dosya sonuna EOF etiketi konur. Çıktı artık çalıştırılabilir ve birleştirilmiş, link edilmiş formdadır.

Belleksel Yerleşim

Küçük ve doğrudan kullanılan sabitler komutla birlikte taşınırken, daha büyük sabitler genellikle sabit veri alanlarında saklanır. Bu durumda sabite dolaylı erişim sağlanır:

```
assembly
MOV &TABLE_VAL, R9   ; TABLE_VAL bellekte bir sabit
```

Ancak doğrudan kullanımlar hala en yaygın ve verimli yöntemdir:

```
assembly
MOV #1000, R9
```

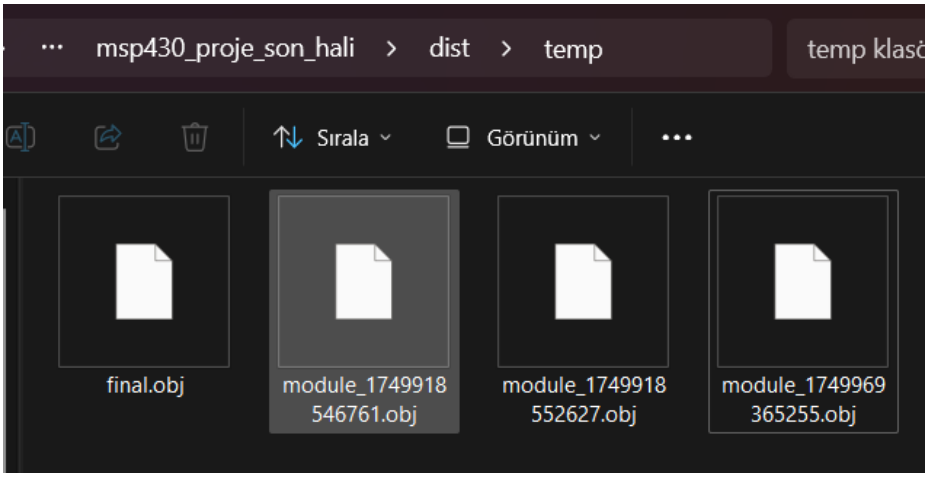
Bu yapı, literal değerlerin **doğrudan komut içinde işlenmesi** esasına dayalıdır ve sabit veri ile çalışmayı oldukça pratik hale getirir.

Programın Dosyalama Yapısı ve Üretilen Dosyalar

MSP430 için geliştirilen projelerde, `.asm` uzantılı assembly kaynak dosyaları çalıştırıldığında, derleme ve bağlama işlemleri sonucu çeşitli ara dosyalar ve çıktı dosyaları oluşur. Bu dosyaların yapısı ve anlamı aşağıda açıklanmıştır:

temp/ Klasörü

Assembly dosyaları çalıştırıldığında, proje dizininde otomatik olarak bir `temp` klasörü oluşturulur.



Bu klasör, derleme ve bağlama sürecinde oluşan geçici dosyaları içerir. Dosyaların yapısı şu şekildedir:

- **Tek bir .asm dosyası çalıştırıldığında:**
  - `temp/` klasörü içinde ilgili dosyaya ait `.obj` (object) dosyası oluşturulur.
  - Bu `.obj` dosyası, semboller ve relocation bilgileri içeren, ancak doğrudan çalıştırılamayan bir ara çıktıdır.
- **Birden fazla .asm dosyası link edildiğinde:**
  - Tüm `.obj` dosyaları `temp/` klasörü içinde tutulur.
  - Linker tarafından bu `.obj` dosyaları birleştirilerek `temp/final.obj` adlı dosya üretilir.
  - `final.obj`, içindeki tüm sembolleri çözülmüş, çalışmaya hazır nesne kodudur.

.bin Dosyası

Projede ayrıca final.obj dosyasından elde edilen bir **.bin (binary)** dosyası da yer almaktadır. Bu dosya:

- Programın **makine dili karşılığıdır**.
- Genellikle **MSP430 mikrodenetleyicisine doğrudan yüklenmek üzere** kullanılır.
- **RAM'e** yazmak için uygun ham veri formatındadır.
- İçeriğinde sadece komutlar ve veriler yer alır; herhangi bir sembol veya yorum içermez.

.bin Dosyasının Oluşturulması

`.bin` dosyası, projenin ana uygulamasından (main arayüzden) **bağımsız** olarak

klasöre eklenen bir Python scripti yardımıyla oluşturulmaktadır. Bu amaçla geliştirilen örnek bir **bin_donusturucu.py** (veya benzeri) dosyası çalıştırıldığında:

1. Kullanıcıdan bir **.obj** dosyası seçmesi istenir.
2. Seçilen **.obj** dosyası işlenerek, karşılık gelen **.bin** dosyası aynı klasöre oluşturulur.

Bu sayede kullanıcı, komut satırına ihtiyaç duymadan doğrudan arayüz üzerinden MSP430 için yüklenebilir formatta dosya üretebilir.

.

Kaynakça

Texas Instruments. (2022). *MSP430x1xx Family User's Guide (Rev. Y)*. https://www.ti.com/lit/ug/slau131y/slau131y.pdf

Texas Instruments. (2000). *MSP430 Assembly Language Tools v1.0 User's Guide*. https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf