SAKARYA
UYGULAMALI BİLİMLER
ÜNİVERSİTESİ

MSP430 Assembler Tasarımı
Proje Raporu

Takım Üyeleri:

Adem COŞKUN – B200109003
Emine KAYIT – B210109372
Sedanur PEKER - 22010903060
Zeliha POLAT - 22010903069

1. Giriş

1.1 Amaç

Bu projenin temel amacı, MSP430 mikrodenetleyicisi için özel olarak tasarlanmış bir assembler (derleyici) aracının geliştirilmesidir. Geliştirilen bu araç, assembly dilinde yazılan kodları makine diline çevirerek mikrodenetleyici tarafından çalıştırılabilir hale getirir. Proje, düşük seviyeli programlama mantığını pekiştirmek ve donanım-yazılım etkileşimini daha iyi kavramak adına önemli bir mühendislik problemi olan assembler tasarımı üzerinden ele alınmıştır.

1.2 Kapsam

Bu çalışma, MSP430 assembly dilinde yazılmış örnek kodları analiz ederek, söz konusu kodları makine koduna çevirebilen iki geçişli (two-pass) bir derleyici mimarisi üzerine kurulmuştur. Projede aşağıdaki işlemler başarıyla gerçekleştirilmiştir:

- Assembly komutlarının ayrıştırılması (parsing)
- Etiketlerin ve sembollerin tanımlanması
- .word gibi direktiflerin işlenmesi
- Sembol tablosunun oluşturulması
- Makine kodlarının üretilmesi

1.3 Kullanılan Teknolojiler

Proje kapsamında kullanılan başlıca teknolojiler ve araçlar şunlardır:

- Python: Derleyicinin ana geliştirme dili olarak kullanılmıştır. Dosya okuma/yazma, metin ayrıştırma, veri yapıları oluşturma ve hata kontrol mekanizmaları gibi işlemler Python diliyle gerçekleştirilmiştir.
- MSP430 Assembly Dili: Kodların girdi olarak alındığı düşük seviyeli programlama dili.
- Opcode ve Sembol Tabloları: Assembly komutlarının karşılık gelen makine kodlarına dönüştürülmesi amacıyla özel veri yapıları tasarlanmıştır.
- Terminal/CLI Arayüzü: Kullanıcıdan assembly dosyasını alıp çıktı dosyasını oluşturmak için komut satırı kullanılmıştır.

2. Sistem Tasarımı

Geliştirilen assembler aracı, MSP430 mikrodenetleyicisinin assembly dilinde yazılmış programlarını okuyarak bu programları makine koduna dönüştürmeyi hedefleyen bir yazılım sistemidir. Bu süreç, genellikle iki aşamalı (two-pass) bir mimari ile gerçekleştirilir. Bu mimari, önce sembollerin analiz edilmesini, ardından gerçek adreslerin ve makine kodlarının üretilmesini sağlar.

Assembler tasarımı, aşağıdaki temel bileşenlerden oluşmaktadır:

2.1 İki Geçişli Derleyici Mimarisi (Two-Pass Assembler)

- **Pass 1 – Sembol Çözümleme ve Etiket Toplama**
  - İlk geçişte, program satır satır okunur ve aşağıdaki işlemler gerçekleştirilir:
  - Etiketler ve semboller tanımlanır.
  - Her satır için adres bilgisi (program counter) hesaplanır.
  - .word gibi direktifler işlenir.
  - Bir sembol tablosu (symbol table) oluşturulur.

- Bu geçişte makine kodu üretilmez; asıl amaç, tüm sembollerin yerini doğru biçimde belirleyip ikinci geçişte kullanılacak sağlam bir temel oluşturmaktır.

- **Pass 2 – Makine Kodu Üretimi**

İkinci geçişte, oluşturulan sembol tablosu kullanılarak:

- Komutlar çözülür ve opcode değerleri ile eşleştirilir.
- Operandlar analiz edilir, gerekiyorsa sembol tablosundan adresleri alınır.
- Sonuç olarak, her satır için uygun makine kodu üretilir.
- Üretilen makine kodları bir çıktı dosyasına yazılır.

2.2 Sembol Tablosu

Sembol tablosu, assembly kodundaki etiketlerin karşılık geldiği bellek adreslerini saklayan bir veri yapısıdır.
Bu yapı sayesinde, ikinci geçişte operand olarak geçen sembollerin adreslerine hızlıca erişilebilir.

2.3 Opcode Tablosu

Her komutun MSP430 mimarisine uygun opcode karşılığı vardır. Assembler, bir komutla karşılaştığında, bu tabloyu kullanarak komutun makine kodundaki karşılığını belirler.

2.4 Operand Ayırma ve Adresleme Modları

Assembler, komutların operandlarını ayrıştırırken çeşitli adresleme modlarını da göz önünde bulundurur:

- **Register mode** (örneğin: R5)
- **Immediate mode** (örneğin: #10)
- **Symbolic/Label addressing** (örneğin: LOOP)
- **Indexed mode** (örneğin: 10(R4))

Bu ayrım operand analizinde özel bir mantıkla yapılır ve her bir adresleme türüne karşılık gelen opcode bit düzeni hesaplanır.

2.5 Hata Kontrolü ve Uyarılar

Sistem tasarımı sırasında kullanıcıyı yönlendirmek için bazı hata kontrol mekanizmaları da yerleştirilmiştir:

- Tanımlanmamış sembol uyarısı
- Geçersiz komut veya operand
- Desteklenmeyen direktif/komut bildirimi

Bu kontroller sayesinde geliştirilen assembler aracı yalnızca doğru yazılmış kodları işler, hata içeren satırları kullanıcıya bildirir.

3. Kodun Açıklanması

3.1.Kütüphaneler

```
import re
import tkinter as tk
from tkinter import ttk, scrolledtext, messagebox
```

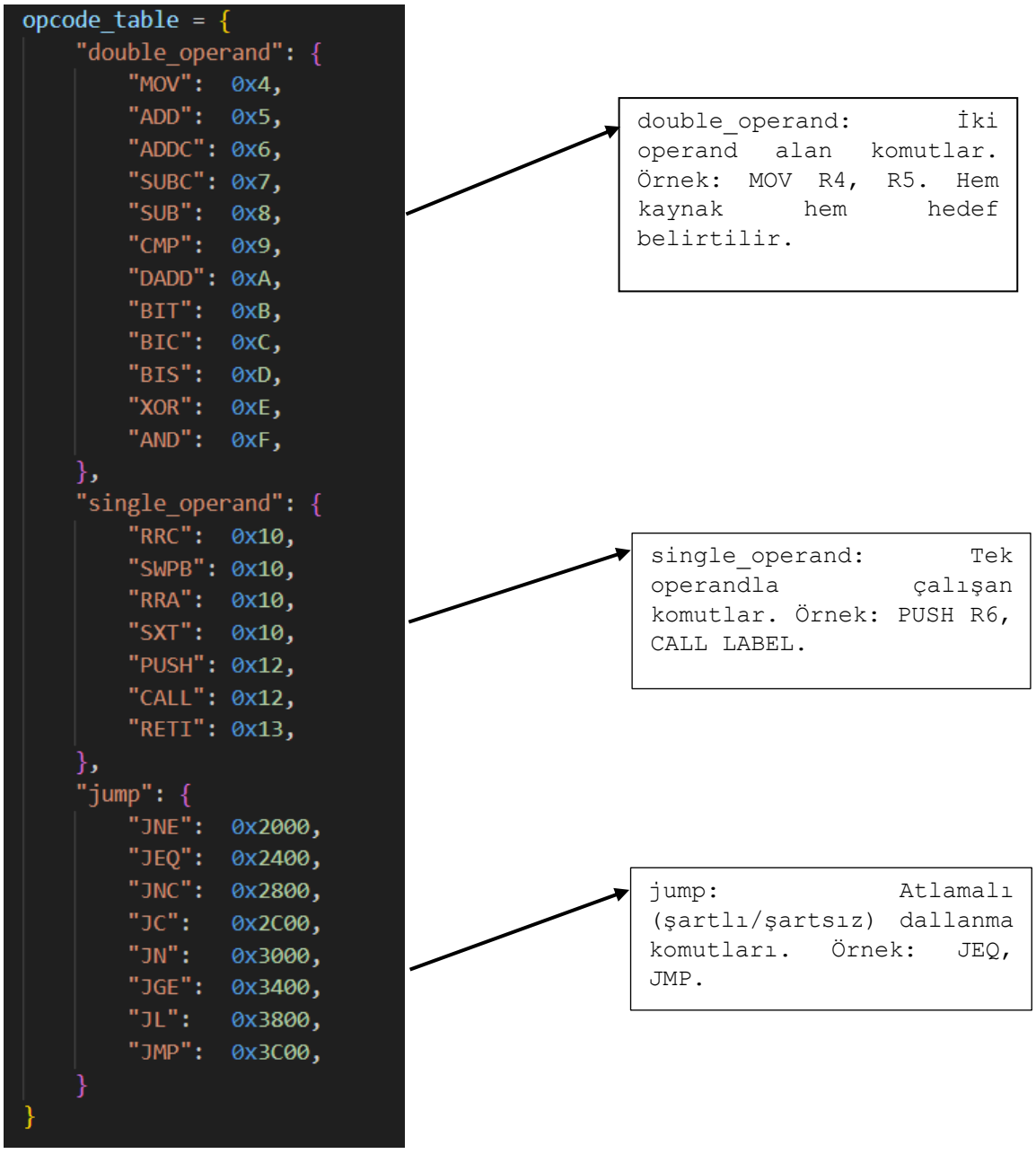
re: Düzenli ifadeler (regex) ile metin işleme işlemleri yapmak için kullanılır.
tkinter: Python'un yerleşik GUI kütüphanesidir. tk takma adıyla kısaltılarak kullanımı kolaylaştırılmış.

- **ttk:** Daha modern ve şık arayüz bileşenleri (buton, çerçeve vs.) sunar.

- **scrolledtext:** Kaydırılabilir metin kutusu.
- **messagebox:** Uyarı ve bilgi pencereleri göstermek için kullanılır.

3.2. Msp430 Opcode Tablosu

MSP430 mimarisi için desteklenen temel komutların opcode (işlem kodu) değerlerini kategorilere ayırarak saklar. Bu yapı sayesinde assembler, karşılaştığı assembly komutunu kolayca tanıyıp uygun makine koduna çevirebilir.



Bu yapı sayesinde assembler, komutun türünü ve hangi opcode kategorisine ait olduğunu kolayca ayırt eder ve buna göre işleme alır.

3.3. Parse Operand Fonksiyonu

```python
# parse_operand fonksiyonu
def parse_operand(operand, symbol_table=None, bw_bit=0):
    operand = operand.strip()
    if operand.startswith('#'):
        value_str = operand[1:].strip()

        if re.match(r"^'.'$", value_str):
            char = value_str[1]
            return {'mode': 'immediate', 'value': ord(char), 'As': 0x3, 'register': 'R0'}

        elif value_str == "''":
            raise ValueError("Bos karakter literali gecersiz: #''")

        elif re.match(r'^0b[01]+$', value_str) or re.match(r'^[01]+b$', value_str):
            bin_val = value_str[2:] if value_str.startswith('0b') else value_str[:-1]
            return {'mode': 'immediate', 'value': int(bin_val, 2), 'As': 0x3, 'register': 'R0'}

        elif re.match(r'^0x[0-9a-fA-F]+$', value_str) or re.match(r'^[0-9a-fA-F]+h$', value_str):
            hex_val = value_str[2:] if value_str.startswith('0x') else value_str[:-1]
            return {'mode': 'immediate', 'value': int(hex_val, 16), 'As': 0x3, 'register': 'R0'}

        elif value_str.isdigit():
            return {'mode': 'immediate', 'value': int(value_str), 'As': 0x3, 'register': 'R0'}

        else:
            if symbol_table and value_str in symbol_table:
                if isinstance(symbol_table[value_str], dict) and 'value' in symbol_table[value_str]:
                    return {'mode': 'immediate', 'value': int(symbol_table[value_str]['value']), 'As': 0x3, 'register': 'R0'}
                else:
                    return {'mode': 'immediate', 'value': int(symbol_table[value_str]), 'As': 0x3, 'register': 'R0'}

    elif operand.startswith('&'):
        label = operand[1:]
        if symbol_table and label in symbol_table:
            if isinstance(symbol_table[label], dict) and 'value' in symbol_table[label]:
                return {'mode': 'absolute', 'value': int(symbol_table[label]['value']), 'As': 0x1, 'register': 'R2'}
            else:
                return {'mode': 'absolute', 'value': int(symbol_table[label]), 'As': 0x1, 'register': 'R2'}
        try:
            addr = int(label, 0)
            return {'mode': 'absolute', 'value': addr, 'As': 0x1, 'register': 'R2'}
        except ValueError:
            return {'mode': 'absolute', 'label': label, 'As': 0x1, 'register': 'R2'}

    elif re.match(r'^-?\d+\(R\d+\)$', operand):
        offset, reg = operand[:-1].split('(')
        return {
```

parse_operand fonksiyonu, MSP430 assembler tasarımında bir komutun operandının hangi adresleme moduna ait olduğunu belirlemek amacıyla kullanılır. Fonksiyon, kendisine verilen operandı analiz ederek immediate, mutlak, indeksli, dolaylı, otomatik artan, register veya sembolik adresleme modlarından hangisine ait olduğunu tespit eder. Bu belirlemeye göre operandın adresleme modu (mode), değeri (value), hangi register'ı kullandığı (register) ve adresleme biti (As) gibi bilgiler içeren bir sözlük yapısı oluşturur. Bu bilgiler, assembler'ın ikinci geçişinde (pass2) komutun doğru şekilde makine koduna dönüştürülmesi için kullanılır.

MSP430 ASSEMBLY DE LİTERALLERİN KULLANILMASI

Msp430 assembly dilinde literallerin (sabit değerlerin) başına # işareti konur.
Bu işaret immediate adresleme modu anlamına gelir. Yani bir registera sabit bir değer yüklemek veya bir işlemde doğrudan sabit kullanmak istediğimizde # kullanılır. Aksi halde, derleyici bu değeri bir bellek adresi olarak yorumlar ve hatalara yol açar. # işareti, hem literaller hem de immediate adresleme modu için kullanılır. Literal ve immediate değer aslında aynı şeyi ifade eder. # işareti bir değerin literal olduğunu ve immediate modda kullanıldığını belirtir.

→ Assembly derleyicisi şu türde literalleri destekler;
a) İkilik (Binary) Tam Sayılar:
0b veya ….B ile tanımlanır. .byte, .word, .long ile boyut belirtir.

b) Sekizlik (Oktal) Tam Sayılar:
Sondan Q veya q ile yazılır. Alternatif olarak sadece başında 0 varsa ve sadece 0-7 rakamları içeriyorsa da geçerlidir.

c) Onluk (Decimal) Tam Sayılar:
Negatif veya pozitif olabilir. Direktif sayı yazılır.

d) Onaltılık (Hexadecimal) Tam Sayılar:
Ya sonda H veya h olur, ya da başta 0xkullanılır. H/h kullanıldığında ilk karakter 0-9 olmalıdır.

e) Karakter Sabitleri :
Tek tırnak içinde ('a') , ('c') yazılır. ASCII değeri saklanır. Boşkarakterler için ' ' kullanılır. Değeri 0'dır. Karakter sabiti ile karakter dizisi farklıdır. Bu sadece 1 karakteri temsil eder.

f) Karakter Dizisi (String) Sabitleri:
Çift tırnak içinde "Merhaba" string direktifi null-terminated dize oluşturur. ASCII olarak saklanır. Uzunluk kullanıldığı komutla sınırlıdır. "sample program" ; 14 karakterlik bir dize.

Kullanım Verileri

1) .copy "dosya_adı"
2) .sect "bölüm_ismi"
3) .byte "metin"

g) Ondalıklı (floating-point) Sayılar:
MSP430'da doğrudan destek yok ; $strtod() dönüşüm yapılır.

3.4. resolve_forward_references Fonksiyonu

```python
# resolve_forward_references fonksiyonu eklenmiştir
def resolve_forward_references(symbol_table):
    """
    İleriye dönük referansları sembol tablosunda çözer (pass1 için)
    """
    for label in list(symbol_table.keys()):
        entry = symbol_table[label]
        if not entry.get('defined', False) and 'forward_references' in entry:
            for ref_label in entry['forward_references']:
                if ref_label in symbol_table and symbol_table[ref_label].get('is_constant', False):
                    try:
                        # Eğer referans bir .equ/.set sabitiyse değeri güncelle
                        symbol_table[label]['value'] = symbol_table[ref_label]['value']
                        symbol_table[label]['defined'] = True
                    except KeyError:
                        pass
```

Bu fonksiyon, pass1 sonunda çağrılarak ileriye dönük referansları çözer.

Temel İşleyiş:

- **Tüm Sembolleri Tarar:** symbol_table'daki her sembol için:
- **Tanımsız & Forward Referans Varsa:** Sembol henüz tanımlanmamışsa ve başka sembollere referans veriyorsa (forward_references), bu referansların çözülüp çözülemediği kontrol edilir.
- **Referans Çözümleme:** Referans verilen sembol (ör. COUNT) sabitse (.equ/.set), değeri güncellenir ve defined işaretlenir.

3.5.Pass1 Fonksiyonu

```
    elif line.lower().startswith('.sect'):
        match = re.match(r'\.sect\s+\"([^\"]+)\"', line, re.IGNORECASE)
        if match:
            sect_name = match.group(1)
            current_section = sect_name
            if sect_name not in section_addresses:
                section_addresses[sect_name] = 0
            location_counter = section_addresses[sect_name]
        continue

    elif line.lower().startswith('.text'):
        current_section = 'text'
        location_counter = section_addresses.get(current_section, 0)
        continue
    elif line.lower().startswith('.data'):
        current_section = 'data'
        location_counter = section_addresses.get(current_section, 0x0200)
        continue
    elif line.lower().startswith('.bss'):
        current_section = 'bss'
        location_counter = section_addresses.get(current_section, 0x0400)
        continue
```

Geliştirilmiş pass1 fonksiyonu ile önceki sürüm arasındaki en temel farklardan biri, yeni direktiflerin tanınması ve bunlara bağlı olarak location_counter değerinin daha doğru ve mimariye uygun şekilde güncellenmesidir. Eski versiyonda yalnızca START komutu ile başlangıç adresi belirlenirken, yeni versiyonda .org, .text, .data, .bss, .sect, .usect gibi blok yönetimi direktifleri desteklenmiş ve her biri location_counter'ı ilgili segmentin başlangıç adresine veya kullanıcı tarafından verilen adrese göre güncelleyecek şekilde tasarlanmıştır.

MSP430 Assembler'ında Program Blokları

MSP430 assembler'ında program blokları, yazılımı donanıma uygun biçimde kod ve veri bölümlerine ayırmak amacıyla kullanılır ve her blok, assembler'ın komutları ya da verileri belleğe doğru yerleştirmesini sağlar. Bu amaçla kullanılan .text, .data, .bss, .sect ve .usect gibi direktifler, assembler'ın hangi bölümde çalıştığını belirler. **.text** bölümü yürütülebilir komutların bulunduğu kod segmentidir ve genellikle ROM'da saklanır. **.data**, başlangıç değeri verilmiş değişkenlerin tutulduğu veri segmentidir; RAM'e yüklenir ve çalıştırma sırasında erişilir. **.bss** bölümü ise başlangıç değeri olmayan, yalnızca yer ayrılması gereken değişkenler içindir; bellekte sıfırla başlatılır ancak fiziksel içerik içermez. Kullanıcı tanımlı özel bölümler oluşturmak için kullanılan **.sect**, yeni bir kod veya veri segmenti başlatır ve ilgili sembolleri bu segmentle ilişkilendirir. **.usect** ise özel ve başlatılmamış veri blokları tanımlamak için kullanılır; genellikle tamponlar ya da donanıma ayrılmış alanlar gibi içerik üretmeyen ama adres tutulması gereken bölümler içindir. Assembler her bölüm için ayrı bir adres sayacı (SPC – Section Program Counter) kullanır, sembolleri bulunduğu bölüme göre konumlandırır ve bu sayede doğru bellek haritalaması sağlar. Program blokları Pass 1 aşamasında algılanır ve tanımlanır, Pass 2 aşamasında ise bu bilgiler kullanılarak kod ve veri içerikleri doğru segmentlere yerleştirilir. Bu ayrım, hem yazılımın yapısal düzenliliğini sağlar hem de fiziksel bellekteki yerleşimin donanıma uygun şekilde organize edilmesine olanak tanır. Ayrıca sembollerin segment takibi, relocation işlemleri ve obje dosyası üretimi gibi derleme sonrası işlemlerde de blok yapısı kritik rol oynar. MSP430 gibi kod

ve veri belleği ayrık olan mimarilerde bu ayrımın sağlanması, sistemin hatasız, kararlı ve donanım uyumlu şekilde çalışabilmesi için zorunludur.

```
if '.equ' in line.lower() or '.set' in line.lower():
    parts = re.split(r'\s+', line, maxsplit=2)
    if len(parts) >= 3 and parts[1].lower() in ['.equ', '.set']:
        label = parts[0].strip()
        value_expr = parts[2].strip()

        symbols_in_expr = re.findall(r'\b[A-Za-z_]\w*\b', value_expr)
        operand_types = []
        unresolved = False

        for sym in symbols_in_expr:
            if sym not in symbol_table or not symbol_table[sym].get("def
                if sym not in symbol_table:
```

.set Direktifi

MSP430'da .set direktifi, derleme zamanında bir sembole sabit bir değer atamak için kullanılır. Bu sembol, program içinde sabit değerlerin (sayılar, adresler vb.) anlamlı isimlerle temsil edilmesini sağlar. Örneğin, belirli bir kaydedicinin adresini veya sabit bir sayıyı tanımlamak için kullanılabilir.

Sözdizimi:

```
SAYI .set 0x20     ; SAYI sembolüne 0x20 değeri atanır
MOV #SAYI, R10     ; R10'a 0x20 değeri yüklenir
```

- Sembol, kodun herhangi bir yerinde kullanılabilir.
- Değer, önceden tanımlanmış olmalıdır (örneğin, sayılar veya daha önce tanımlanmış semboller).

.equ Direktifi

`.equ` direktifi, `.set` ile tamamen aynı işlevi görür ve MSP430'da benzer şekilde kullanılır. Tek fark, yazım tercihine bağlıdır. Özellikle sabitleri tanımlarken okunabilirliği artırmak için tercih edilebilir.

Söz Dizimi:

```
PORT1_ADDR .equ 0x0200      ;PORT1'in bellek adresi tanımlanır
MOV &PORT1_ADDR, R11        ;PORT1'in değeri R11'e okunur
```

- .equ ile tanımlanan semboller, .global veya .def ile harici modüllere açılabilir.
- Her iki direktif de derleyici tarafından işlenir ve gerçek makine koduna dönüşmez, yalnızca derleme sırasında sembolik değerlerin yerini alır.

Not: MSP430'da genellikle adres sabitleri, maske değerleri veya sık kullanılan sayılar için bu direktifler tercih edilir.İkisi de birbirinin yerine kullanılabilir; seçim kişisel kod stilize etme alışkanlığına bağlıdır.

MSP430'da .global, .def ve .ref Kullanımı

MSP430 assembly dilinde .global, .def ve .ref direktifleri, farklı modüller arasında sembollerin (değişkenler, fonksiyonlar, sabitler) paylaşılmasını

sağlamak için kullanılır. Bu direktifler, bağlayıcının (linker) sembolleri doğru şekilde çözümlemesine yardımcı olur.

1. .global:

Hem .def hem de .ref gibi davranır. Eğer sembol mevcut dosyada tanımlıysa (LABEL:, .set, .equ, vs.), .def gibi davranır (dışa aktarır). Eğer sembol başka bir dosyada tanımlıysa, .ref gibi davranır (harici referans olarak işaretler).

```
Söz Dizimi:
.global INIT      ; INIT bu dosyada tanımlıysa dışa aktarılır, değilse harici
                  ; referans olarak işaretlenir
INIT:             ;INIT tanımı (fonksiyon veya veri etiketi)
      MOV #0xFF, R5
      RET
```

2. .def:

Sembolün mevcut dosyada tanımlandığını ve diğer dosyalar tarafından kullanılabileceğini belirtir. Linker, bu sembolü diğer modüllere "export" eder.

```
Söz Dizimi
.def INIT         ;INIT bu dosyada tanımlı ve diğer dosyalar tarafından
                  ;kullanılabilir
INIT:
     CLR R6
```

3. .ref:

Sembolün başka bir dosyada tanımlandığını belirtir (harici referans). Linker, sembolün tanımını diğer dosyalarda arar. Eğer tanım bulunamazsa, "undefined reference" hatası verir.

```
Söz Dizimi
.ref DELAY        ;DELAY başka bir dosyada tanımlı ve bu dosyada kullanılacak
main:
     CALL DELAY
```

Önemli Kurallar

1. Bir sembol birden fazla yerde .def veya .global ile tanımlanırsa, linker "çoklu tanım" hatası verir.

2. .ref ile işaretlenen bir sembolün mutlaka bir yerde tanımlanması gerekir, yoksa linker hatası olur.

3. .global daha esnektir, ancak .def ve .ref kullanımı daha açık ve kontrollüdür.

Sonuç

- .global → Hem tanımlamak hem de referans vermek için kullanılır.

- .def → Sadece tanımlanan sembolleri dışa aktarır.
- .ref → Sadece harici referansları belirtir.

MSP430 gömülü sistemlerde, özellikle büyük projelerde modüler programlama için bu direktifler önemlidir. Doğru kullanılmazsa linker hataları (undefined reference, multiple definition) ortaya çıkabilir.

$ Sembolünün Kullanımı ve İşlenmesi

MSP430 assembler'ında $ sembolü, mevcut bölümün program sayacını (SPC) temsil eder ve .equ, .set gibi direktiflerde kullanıldığında otomatik olarak o anki adres değerine dönüştürülür.

Temel Anlamı: $, bulunduğu bölümdeki (text, data, vs.) mevcut konumu (SPC) gösterir. Örneğin .text bölümünde $ kullanılırsa, o anda .text'in SPC değerini verir.

3.6.eval_value_expression Fonksiyonu

eval_value_expression fonksiyonu, bir assembly ifadesinin sayısal değerini hesaplamak için kullanılır. Özellikle .equ veya .set gibi direktiflerle tanımlanan sabitlerin değerini çözümlemek amacıyla tasarlanmıştır. Fonksiyonun amacı verilen expr ifadesini ve symbol_table'ı kullanarak bu ifadenin tam sayı karşılığını (int) döndürmektir.

```
def eval_value_expression(expr, symbol_table):
    expr = expr.strip()

    if expr.startswith('0x'):
        return int(expr[2:], 16)
    elif expr.startswith('0b'):
        return int(expr[2:], 2)
    elif expr.isdigit():
        return int(expr)
```

Başındaki boşlukları temizler ve basit sayısal sabitleri doğrudan tanır.

```
if expr in symbol_table:
    if isinstance(symbol_table[expr], dict) and 'value' in symbol_table[expr]:
        if symbol_table[expr]['is_constant']:
            return int(symbol_table[expr]['value'])
        else:
            raise ValueError(f"Sabit olmayan sembol kullanilamaz: {expr}")
    else:
        return int(symbol_table[expr])
```

Eğer ifade sadece bir sembol adıysa (örneğin VAL1), bu sembolün sabit olup olmadığı kontrol edilir. is_constant == True ise değeri döndürülür. Aksi halde hata fırlatılır.

3.7. Pass2 Fonksiyonu

Bu fonksiyon, assembly kodunu makine koduna çeviren ikinci aşamadır. İlk aşamada belirlenen bölüm adreslerini (section_addresses) ve sembol tablosunu (symbol_table) kullanarak, kodun gerçek bellek adreslerine yerleştirilmesini sağlar. Ayrıca .word, .text, .data gibi direktifleri ve atlama komutlarını işler.

```python
def pass2(lines, symbol_table, opcode_table):
    machine_code = []
    literals_table = []
    location_counter = 0
    start_address = None
    current_section = 'text'
    section_addresses = {
        'text': 0,
        'data': 0x0200,
        'bss': 0x0400
    }
    location_counter = section_addresses[current_section]

    for line in lines:
        line = line.split(';', 1)[0].strip()
        if not line:
            continue
```

Girdiler:

- lines: Assembly kod satırları.
- symbol_table: Pass1'de oluşturulan sembol tablosu (etiketler ve adresleri).
- opcode_table: Komutların makine kodu karşılıklarını içeren tablo.

Çıktılar:

- machine_code: Üretilen makine kodu (adres, değer çiftleri).
- literals_table: Kodda kullanılan sabit değerlerin listesi.

Başlangıç Değerleri:

- section_addresses: Bölümlerin varsayılan adresleri (.text: 0x0000, .data: 0x0200, .bss: 0x0400).
- location_counter: Kodun yerleştirileceği anlık bellek adresi.

```python
# Handle section directives
if line.lower().startswith('.org'):
    addr = line[len('.org'):].strip()
    location_counter = int(addr, 16) if addr.lower().startswith('0x') else int(addr, 0)
    continue

if line.lower().startswith('.text'):
    current_section = 'text'
    location_counter = section_addresses.get(current_section, 0)
    continue
elif line.lower().startswith('.data'):
    current_section = 'data'
    location_counter = section_addresses.get(current_section, 0x0200)
    continue
elif line.lower().startswith('.bss'):
    current_section = 'bss'
    location_counter = section_addresses.get(current_section, 0x0400)
    continue
```

.org ADRES:

location_counter'ı belirtilen adrese atar. Örneğin, .org 0x100 sonrası tüm kodlar 0x100 adresinden başlar.

.word Direktifinin İşlenmesi:

```
# Handle .word directives in pass2
if line.startswith('.word'):
    if ':' in line:
        label_part, rest = line.split(':', 1)
        label = label_part.strip()
        values = [v.strip() for v in re.split(r'\s*,\s*', rest.strip())]
    else:
        label = None
        values = [v.strip() for v in re.split(r'\s*,\s*', line[len('.word'):].strip())]
    for value in values:
        int_value = int(value, 16) if value.lower().startswith('0x') else int(value, 0)
        if label:
            symbol_table[label] = {'value': location_counter, 'type': 'data', 'defined':
        machine_code.append((location_counter, int_value))
        location_counter += 2
    continue
```

Belleğe 16-bit (2 byte) değerler yerleştirir. Etiket varsa (örneğin buffer: .word 0x1234), sembol tablosuna bu etiketin adresini kaydeder.
Örnek Kullanım: .word 0x1234, 5678

Etiketlerin (Labels) İşlenmesi

```
# Handle labels
label = None
if ':' in line:
    label_part, rest = line.split(':', 1)
    label = label_part.strip()
    line = rest.strip()

parts = re.split(r'\s+', line, maxsplit=1)
mnemonic = parts[0].upper() if parts else ""
operands = parts[1] if len(parts) > 1 else ""
```

Örnek Kullanım: loop: MOV R1, R2
loop etiketi, bulunduğu adresi (location_counter) sembol tablosuna kaydeder.

Komutların Makine Koduna Çevrilmesi

```
if mnemonic_clean in opcode_table["double_operand"]:
    ops = [op.strip() for op in operands.split(',')]
    src_info = parse_operand(ops[0], symbol_table, bw_bit)
    dst_info = parse_operand(ops[1], symbol_table, bw_bit)

    src_reg = int(src_info['register'][1:]) if 'register' in src_info else 0
    dst_reg = int(dst_info['register'][1:]) if 'register' in dst_info else 0

    As = src_info['As']
    Ad = 1 if dst_info['mode'] in ['indexed', 'symbolic', 'absolute'] else 0

    opcode = opcode_table["double_operand"][mnemonic_clean]
    word = (opcode << 12) | (src_reg << 8) | (Ad << 7) | (bw_bit << 6) | (As << 4) | dst_reg
    machine_code.append((location_counter, int(word)))
    location_counter += 2
```

Çift Operandlı Komutlar (Örnek: MOV R1, R2):
Kaynak (src) ve hedef (dst) operandları ayrıştırılır.Opcode tablosundan komutun makine kodu karşılığı alınır.Adresleme modlarına göre ekstra kelimeler (örneğin 0x1234) eklenir.

```
elif mnemonic_clean in opcode_table["single_operand"]:
    if mnemonic_clean == "RETI":
        word = 0x1300
        machine_code.append((location_counter, int(word)))
        location_counter += 2
    else:
        operand_info = parse_operand(operands, symbol_table, bw_bit)
        reg = int(operand_info['register'][1:]) if 'register' in operand_info else 0
        As = operand_info['As']
```

Tek Operandlı Komutlar (Örnek: PUSH R5, CALL delay):
RETI gibi özel komutlar doğrudan sabit makine koduna çevrilir.
CALL için hedef adres sembol tablosundan alınır.

```
elif mnemonic_clean in opcode_table["jump"]:
    offset_label = operands.strip()
    if offset_label not in symbol_table:
        raise ValueError(f"Etiket bulunamadi: {offset_label}")

    if isinstance(symbol_table[offset_label], dict) and 'value' in symbol_table[offset_label]:
        target_address = int(symbol_table[offset_label]['value'])
    else:
        target_address = int(symbol_table[offset_label])
```

Atlama Komutları (Örnek: JMP loop):
Hedef etiketin adresi sembol tablosundan alınır.
offset = (hedef_adres - mevcut_adres) / 2 hesaplanır.

3.8. Assemble Fonksiyonu

```
def assemble(assembly_code):
    lines = assembly_code.strip().split('\n')
    symbol_table = pass1(lines)
    machine_code, literals = pass2(lines, symbol_table, opcode_table)

    # Keep machine code as integers for now
    formatted_machine_code = []
    for addr, code in machine_code:
        formatted_machine_code.append((addr, code))

    print(f"Assemble Literals: {literals}, types: {[type(lit['value']) for lit in literals]}")
    return formatted_machine_code, symbol_table, literals
```

Assemble fonksiyonu, MSP430 assembler tasarımında tüm derleme sürecini başlatan ve koordine eden ana işlemi yürütür. Görevi, kullanıcıdan alınan assembly dilindeki kaynak kodu alıp, önce pass1 fonksiyonu ile sembol tablosunu oluşturmak; ardından pass2 fonksiyonu ile bu sembol tablosunu kullanarak gerçek makine kodu üretmektir. Fonksiyonun çalışması üç aşamalıdır:

1. **Satırları Ayrıştırma:** assembly_code parametresi, çok satırlı kaynak kod olarak gelir. Bu metin satır satır bölünerek lines listesi elde edilir.

2. **Pass1 Uygulaması:** pass1(lines) çağrısı ile tüm etiketler, sabitler, program bölümleri (.text, .data, .bss) ve konum sayaçları analiz edilerek bir sembol tablosu (symbol_table) oluşturulur.
3. **Pass2 Uygulaması:** pass2(lines, symbol_table, opcode_table) çağrısı ile daha önce oluşturulan sembol tablosu kullanılarak her komutun karşılığı olan makine kodları (machine_code) ve literal değerleri (literals) hesaplanır.

Sonuç olarak, assemble fonksiyonu formatted_machine_code, symbol_table ve literals olmak üzere üç önemli veri yapısını döndürür. Bu çıktılar, daha sonra nesne dosyası üretimi, görselleştirme, hata ayıklama veya simülasyon gibi işlemler için kullanılabilir. Bu fonksiyon MSP430 assembler'ın derleme motorunun merkezinde yer alır ve tüm süreci yönetir.

3.9. create_object_file Fonksiyonu

```
f.write("Section Headers:\n")
f.write("  [Nr] Name       Type            Addr   Size\n")
f.write("  [ 0]            NULL            000000 000000\n")
f.write("  [ 1] .text      PROGBITS        000000 %06X\n" % (len(machine_code) * 2))
f.write("  [ 2] .data      PROGBITS        020000 %06X\n" % (len(literals) * 2))
f.write("  [ 3] .symtab    SYMTAB          000000 %06X\n" % (len(symbol_table) * 16))
f.write("  [ 4] .shstrtab  STRTAB          000000 000100\n")
f.write("\n")
```

create_object_file fonksiyonu, MSP430 assembler tarafından üretilen makine kodu, literal değerler ve sembol tablosunu alarak ELF benzeri yapıda bir nesne (object) dosyası oluşturur. Bu dosya; .text bölümünde makine kodlarını, .data bölümünde literal verileri, .symtab bölümünde ise sembollerin adres, tür ve bölüm bilgilerini içerir. Fonksiyon, bu verileri düzgün biçimde formatlayarak dosyaya yazar ve derlenen programın dışa aktarılabilir, bağlantıya hazır bir sürümünü sağlar.

```
f.write(".text Section (Machine Code):\n")
f.write("Address | Code\n")
f.write("---------------\n")
for addr, code in machine_code:
    f.write(f"{addr:04X}    | {code:04X}\n")
f.write("\n")

f.write(".data Section (Literals):\n")
f.write("Address | Value   | Type\n")
f.write("----------------------\n")
for lit in literals:
    addr = lit['address']
    val = lit['value']
    f.write(f"{addr:04X}    | {val:04X} | {lit['type']}\n")
f.write("\n")
```

3.10. AssemblerGUI Sınıfı

```
self.assemble_button = ttk.Button(self.button_frame, text="Derle", command=self.assemble_code, style='TButton')
self.assemble_button.pack(side=tk.LEFT, padx=5)

self.save_object_button = ttk.Button(self.button_frame, text="Obje Dosyasını Kaydet", command=self.save_object_file, style='TButton')
self.save_object_button.pack(side=tk.LEFT, padx=5)

self.output_notebook = ttk.Notebook(self.main_frame)
self.output_notebook.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

self.symbol_frame = ttk.Frame(self.output_notebook)
self.output_notebook.add(self.symbol_frame, text="Sembol Tablosu")

self.symbol_text = scrolledtext.ScrolledText(self.symbol_frame, height=15, width=100, font=("Consolas", 10), bg='#ffffff', fg='#333333')
self.symbol_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

self.literals_frame = ttk.Frame(self.output_notebook)
self.output_notebook.add(self.literals_frame, text="Literals Tablosu")

self.literals_text = scrolledtext.ScrolledText(self.literals_frame, height=15, width=100, font=("Consolas", 10), bg='#ffffff', fg='#333333')
self.literals_text.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

self.machine_frame = ttk.Frame(self.output_notebook)
self.output_notebook.add(self.machine_frame, text="Makine Kodu")
```

MSP430 assembler uygulamasının tüm görsel arayüzünü yönetir ve kullanıcıdan assembly kodu girişi alarak bu kodun derlenmesini, sembol tablosunun, literal tablosunun ve üretilen makine kodlarının ayrı sekmelerde görsel olarak sunulmasını sağlar.

Tkinter ve ttk kullanılarak oluşturulan arayüzde üst kısımda başlık, kullanıcıdan assembly kodunun girildiği bir ScrolledText kutusu, derleme ve obje dosyası kaydetme işlevlerini gerçekleştiren butonlar yer alır.

Ortadaki Notebook sekmeleri ise üç farklı çıktı bölümü içerir: sembol tablosu, literal tablosu ve makine kodu.

Alt kısımda ise işlemlerin durumunu anlık olarak bildiren bir durum çubuğu (status_bar) bulunur. Bu sınıf içinde assemble_code fonksiyonu derleme işlemini başlatırken, save_object_file fonksiyonu oluşan ELF nesne dosyasını kaydetmek için kullanılır.

3.11. assemble_code Fonksiyonu

```
def assemble_code(self):
    try:
        self.status_var.set("Derleniyor...")
        self.status_bar.configure(background='#FF9800', foreground='white')
        assembly_code = self.input_text.get("1.0", tk.END)
        machine_code, symbol_table, literals = assemble(assembly_code)

        print(f"GUI Literals Before Display: {literals}, types: {[type(lit['value']) for lit in literals]}")  # Debug
        print(f"Machine Code Before Display: {machine_code}, types: {[(type(addr), type(code)) for addr, code in machine_c

        self.symbol_text.delete("1.0", tk.END)
        self.literals_text.delete("1.0", tk.END)
        self.machine_text.delete("1.0", tk.END)

        self.symbol_text.insert(tk.END, "Sembol    | Deger | Tur       | Bolum   | Tanimli | Global\n")
        self.symbol_text.insert(tk.END, "-------------------------------------------------------\n")
        for symbol, info in symbol_table.items():
            if isinstance(info, dict) and 'value' in info:
                value = info['value']
                print(f"Symbol Table Entry: {symbol}, value={value}, type={type(value)}")  # Debug
                self.symbol_text.insert(tk.END, f"{symbol:<10} | {value:04X} | {info['type']:<9} | {info['section']:<7} |
            else:
                value = int(info) if isinstance(info, (int, str)) else 0
                self.symbol_text.insert(tk.END, f"{symbol:<10} | {value:04X} | external  | none    | False   | False\n")
```

Assemble_code fonksiyonu, kullanıcı tarafından arayüze girilen MSP430 assembly kodunu alır, bu kodu satır satır assemble fonksiyonuna göndererek sembol tablosu, literal tablosu ve makine kodu üretir. Derleme işlemi başarılı olursa, GUI'deki ilgili bölmelere (sembol tablosu, literal tablosu, makine kodları) bu veriler yazdırılır. Aynı zamanda create_object_file fonksiyonu çağrılarak bu bilgiler .o uzantılı ELF benzeri bir nesne dosyasına kaydedilir. Kullanıcıya durum çubuğunda derlemenin başarılı olup olmadığı görsel olarak bildirilir. Hatalı bir durumla karşılaşılırsa, kullanıcıya bir hata mesajı gösterilir ve durum çubuğu uygun şekilde kırmızı renkle güncellenir. Bu yapı, hem kullanıcı geri bildirimi hem de iç hata ayıklama için debug çıktıları ile desteklenmiştir.

3.12. save_object_file Fonksiyonu

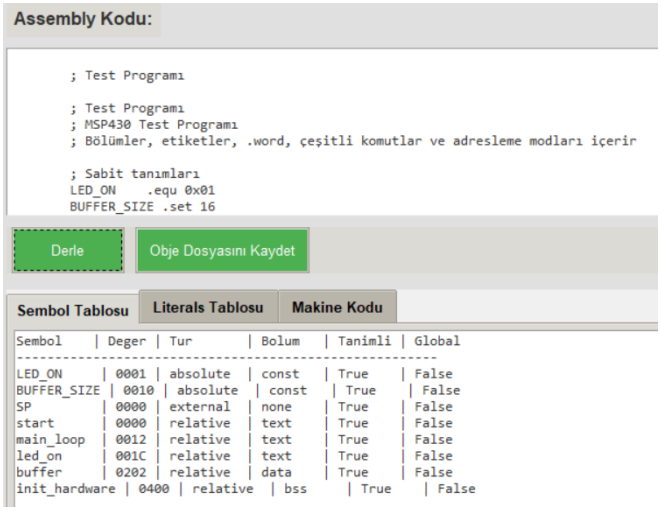
```
def save_object_file(self):
    if not hasattr(self, 'object_filename'):
        messagebox.showwarning("Uyari", "Lutfen once kodu derleyin.")
        self.status_var.set("Once derleme yapin")
        self.status_bar.configure(background='#FF9800', foreground='white')
        return

    try:
        with open(self.object_filename, 'r') as f:
            content = f.read()
        with open(self.object_filename, 'w') as f:
            f.write(content)
        self.status_var.set(f"ELF Obje dosyasi kaydedildi: {self.object_filename}")
        self.status_bar.configure(background='#4CAF50', foreground='white')
    except Exception as e:
        messagebox.showerror("Hata", f"ELF Obje dosyasi kaydedilemedi: {str(e)}")
        self.status_var.set("ELF Obje dosyasi kaydetme hatasi")
        self.status_bar.configure(background='#F44336', foreground='white')
```

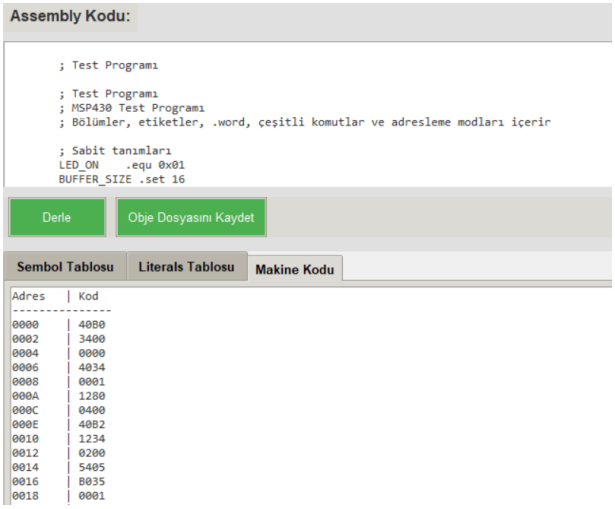
save_object_file fonksiyonu, assembler arayüzünde daha önce derlenmiş olan ELF biçimindeki nesne dosyasını kaydetmek için kullanılır. Eğer kullanıcı derleme işlemini gerçekleştirmemişse (self.object_filename henüz tanımlanmamışsa), bir uyarı mesajı ile önce derleme yapılması gerektiği belirtilir. Derleme yapılmışsa, dosya açılır, içeriği okunur ve ardından aynı adla tekrar yazılarak kaydetme işlemi tamamlanır. Başarılı bir kayıttan sonra durum çubuğu yeşile dönerek kullanıcı bilgilendirilir. Kaydetme esnasında bir hata oluşursa, bu hata kullanıcıya mesaj kutusu aracılığıyla bildirilir ve durum çubuğu kırmızıya döner. Bu yapı, kullanıcı etkileşimini yöneten güvenli bir dosya kaydetme mekanizması sunar.

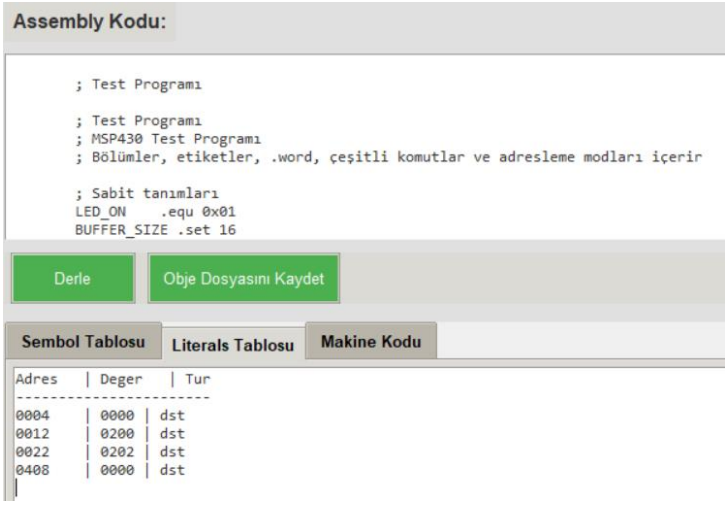
3.13. Test Programı

Bu test programı, MSP430 işlemcisi için yazılmış örnek bir assembly kodudur ve aşağıdaki amaçları taşır:

- Sabit tanımlar (.equ, .set) kullanmayı,
- Kodun farklı bölümlere (.text, .data, .bss) ayrılmasını,
- Tüm **adresleme modlarını** kullanmayı,
- Basit bir donanım başlatma ve döngüsel kontrol yapısı oluşturmayı göstermektedir.







4.ELF (Executable and Linkable Format)

ELF (Executable and Linkable Format), modern işletim sistemlerinde kullanılan standart bir nesne dosyası formatıdır. Derleyici çıktılarının (örneğin .o dosyalarının), bağlantı öncesi aşamada makine kodunu, sembol tablolarını, sabit verileri ve gerekli meta verileri düzenli bir şekilde saklamasını sağlar.

ELF dosyaları genellikle:

- .text bölümü: Makine kodlarını (talimatları),

- .data bölümü: Başlangıçta değeri belirli olan verileri,
- .bss bölümü: Başlangıçta değeri olmayan (sıfır olacak) değişkenleri,
- symbol table ve relocation table gibi ek yapıları içerir.
-

Bu yapı sayesinde: Derleme sonrası çıktılar kolayca bağlayıcı (linker) tarafından okunabilir, yükleyici (loader) bu dosyayı doğrudan çalıştırılabilir programa dönüştürebilir,insanlar ya da araçlar bu bölümleri analiz ederek daha kolay hata ayıklama ve işlem yapabilir.

```
ELF Object File
================

ELF Header:
  Magic:   7F 45 4C 46 (ELF)
  Class:   ELF32
  Data:    2's complement, little endian
  Version: 1 (current)
  OS/ABI:  System V ABI
  Type:    REL (Relocatable file)
  Machine: MSP430
  Entry:   0x0000

Section Headers:
  [Nr] Name       Type            Addr   Size
  [ 0]            NULL            000000 000000
  [ 1] .text      PROGBITS        000000 000038
  [ 2] .data      PROGBITS        020000 000008
  [ 3] .symtab    SYMTAB          000000 000080
  [ 4] .shstrtab  STRTAB          000000 000100

.text Section (Machine Code):
Address | Code
--------------
0000    | 40B0
0002    | 3400
0004    | 0000
0006    | 4034
0008    | 0001
000A    | 1280
000C    | 0400
000E    | 40B2
0010    | 1234
0012    | 0200
0014    | 5405
```

```

  .data Section (Literals):
  Address | Value   | Type
  ----------------------
  0004    | 0000 | dst
  0012    | 0200 | dst
  0022    | 0202 | dst
  0408    | 0000 | dst

  .symtab Section (Symbol Table):
  Symbol     | Value | Type      | Section | Defined | Global
  ------------------------------------------------------------
  LED_ON     | 0001 | absolute  | const   | True    | False
  BUFFER_SIZE | 0010 | absolute  | const   | True    | False
  SP         | 0000 | external  | none    | True    | False
  start      | 0000 | relative  | text    | True    | False
  main_loop  | 0012 | relative  | text    | True    | False
  led_on     | 001C | relative  | text    | True    | False
  buffer     | 0202 | relative  | data    | True    | False
  init_hardware | 0400 | relative  | bss     | True    | False

```

KAYNAKÇA

[1] Texas Instruments. *MSP430 Assembly Language Tools v4.4 User's Guide* (Rev. J). Literature Number: SLAU131J, November 2014. [Online]. Available: https://www.ti.com/lit/pdf/slau131

[2] Texas Instruments. *MSP430 Assembly Language Tools v21.6.0.LTS User's Guide* (Rev. Y). Literature Number: SLAU131Y, October 2004 – Revised June 2021. [Online]. Available: https://www.ti.com/lit/pdf/slau131

[3] Ryan's Edit. *MSP430 Reference – Instruction Set and Architecture Summary*. Internal teaching document derived from MSP430 documentation and educational resources.

[4] Texas Instruments. (n.d.). *MSP430 Family Instruction Set Summary*.