SAKARYA
UYGULAMALI BİLİMLER
ÜNİVERSİTESİ

MSP430 Assembler Tasarımı

Proje Raporu

Takım Üyeleri:

Adem COŞKUN – B200109003

Emine KAYIT – B210109372

Sedanur PEKER - 22010903060

Zeliha POLAT - 22010903069

1.Linker'ın MSP430 Assembler Rolü

MSP430 mikrodenetleyici mimarisinde, linker, bir veya birden fazla derlenmiş nesne dosyasını alarak bunları çalıştırılabilir bir program haline getiren yazılımsal bileşendir. Bu işlem sırasında linker, her modülde bulunan sembolleri çözümler, relocation (adres düzeltme) işlemlerini uygular ve farklı modüllerin içeriklerini bellek üzerinde uygun adreslere yerleştirme planını yapar.

Linker, hangi elf dosyasındaki hangi bölümün, mikrodenetleyicinin hangi fiziksel belleğine yerleşeceğini belirleyerek bellek çakışmalarını engeller, sembolleri gerçek adreslere bağlar ve relocation işlemlerini tamamlar.

Linker, MSP430 assembler çıktılarında semboller ve etiketler üzerinde kritik bir rol oynar; farklı .elf dosyalarındaki sembolleri (fonksiyon adları, değişkenler, etiketler gibi) merkezi bir sembol tablosunda birleştirir, her sembolün ilgili program bölümüne (örneğin .text veya .data) göre gerçek çalıştırma adresini hesaplar ve bu adresleri tüm program boyunca tutarlılık sağlayacak şekilde günceller. Eğer aynı sembol farklı dosyalarda birden fazla kez tanımlanmışsa, çakışmaları tespit eder ve hata verir. Ayrıca tanımsız sembolleri, başka dosyalardan çözerek referansları tamamlar. Böylece tüm semboller, doğru adreslere yerleştirilmiş, tutarlı ve çalışmaya hazır hale gelir.

Bir assembler, yalnızca bireysel .elf dosyaları üretirken, linker bu dosyaları birleştirerek aşağıdaki işlemleri gerçekleştirir:

- .text, .data, .bss, .usect gibi standart bölümleri tanır ve bu bölümleri bellek haritasına uygun şekilde yerleştirir.
- Tüm etiketlerin (label) nihai adreslerini hesaplar.
- Bir komut içerisinde başka bir sembole (etikete) başvuran adresler için relocation tablolarını kullanarak, bu adreslerin doğru yerleri göstermesini sağlar.
- Sonuç olarak bir çalıştırılabilir dosya (ELF formatında) oluşturur.

2.Linker.py Modülü

linker.py modülü, MSP430 assembler sisteminde birden fazla .elf nesne dosyasını alarak bunların .text, .data, sembol tablosu ve relocation bilgilerini birleştirir; sembollerin adreslerini doğru şekilde günceller, relocation (adres çözümleme) işlemlerini gerçekleştirir ve tüm bu bölümleri tek bir bütün halinde birleştirerek çalıştırılabilir bir çıktı dosyası (program.elf) oluşturur. Bu sayede, çok dosyalı projelerde tanımlı ve referans edilen semboller arasında tutarlılık sağlanır, belleğe yerleşim çakışmaları önlenir ve program çalıştırılabilir hale gelir.

read_elf(filename) Fonksiyonu:

Assembler tarafından üretilen .elf benzeri metin tabanlı nesne dosyalarını okuyarak bu dosyalardaki .text, .data, .symtab ve .relocation gibi bölümleri ayrıştırır. Fonksiyon, dosya içeriğini satır satır tarar ve her bölüm başlangıcını özel başlık satırlarından (örneğin .text Section, .data Section) algılayarak modunu (mode) belirler. Her modda, bölüme özgü veriler uygun veri yapılarına dönüştürülerek saklanır: .text ve .data bölümleri adres-kod/değer çiftleri olarak listelere, .symtab sembol adı ve özellikleriyle birlikte sözlüğe, .relocation bölümü ise offset, sembol ve tür bilgisi içeren listelere kaydedilir. Böylece, her .elf dosyasının içeriği yapısal olarak belleğe alınmış olur. Fonksiyon, dönüş değeri olarak sözlük biçiminde bu dört temel bölümün verilerini içeren bir yapı döndürür ve böylece link() fonksiyonunun birden fazla ELF dosyasını birleştirmesi için hazır veri sağlar.

```
def read_elf(filename):
    text_section = []
    data_section = []
    symbol_table = {}
    relocation_entries = []

    mode = None
    for i, line in enumerate(lines):
        line = line.strip()

        if line.startswith('.text Section') or (line.startswith('.text') and 'Section' in line):
            mode = 'text'
            print(f"  Text section başladı (satır {i+1})")
            continue
        elif line.startswith('.data Section') or (line.startswith('.data') and 'Section' in line):
            mode = 'data'
            print(f"  Data section başladı (satır {i+1})")
            continue
        elif line.startswith('.symtab Section') or (line.startswith('.symtab') and 'Section' in line):
            mode = 'symtab'
            print(f"  Symbol table başladı (satır {i+1})")
            continue
        elif line.startswith('.rel') and 'Section' in line:
            mode = 'relocation'
            print(f"  Relocation section başladı (satır {i+1})")
            continue
```

link Fonksiyonu:

Bir veya birden fazla .elf formatındaki nesne dosyasını bir araya getirerek tek bir çalıştırılabilir dosya oluşturur. Bu işlem sırasında her dosya, read_elf() fonksiyonu aracılığıyla ayrıştırılır ve içerdiği .text ve .data bölümleri belirli bir adres ofseti ile hedef bellekte uygun yerlere yerleştirilir. Tüm semboller birleştirilerek global_symbol_table adlı küresel sembol tablosuna kaydedilir. Aynı isimde bir sembol birden fazla dosyada tanımlanmışsa çakışma kontrolü yapılır. Tanımlı olmayan semboller relocation işlemleri sırasında çözümlenmek üzere bekletilir. link() fonksiyonu daha sonra relocation işlemlerini gerçekleştirir; sembol adreslerini belirleyerek linked_text içindeki talimatlara yerleştirir. Son adımda, birleştirilmiş .text, .data, sembol tablosu ve çözülmüş relocation kayıtları bir metin dosyasına yazılır. Bu dosya, linked_output.elf adıyla kaydedilir ve hem insan tarafından okunabilir hem de yükleyici tarafından işlenebilir biçimdedir. Fonksiyon sonunda başarılı birleştirme mesajı verilerek işlem tamamlanır.

```
#birden fazla .elf dosyasını birleştirerek tek bir çalıştırılabilir dosya oluşturur, bölümleri düzenler,
#sembolleri çözümler ve adresleri günceller.
def link(elf_files, output_file='linked_output.elf'):
    linked_text = []
    linked_data = []
    global_symbol_table = {}
    all_relocations = []

    current_text_offset = 0x0000
    current_data_offset = 0x0200

    for filename in elf_files:
        obj = read_elf(filename)
        file_text_start = current_text_offset
        file_data_start = current_data_offset

        for sym, info in obj['symbols'].items():
            updated_info = info.copy()
            if info['section'] == 'text':
                updated_info['value'] += file_text_start
            elif info['section'] == 'data':
                updated_info['value'] += file_data_start
            updated_info['source_file'] = filename
```

3. generate_test_elfs.py Modülünün Amacı

generate_test_elfs.py modülü, MSP430 assembler projesinde birden fazla .asm dosyasını otomatik olarak işleyip her biri için bağımsız .elf (nesne) dosyaları oluşturan bir test ve üretim aracıdır. Her dosya için pass1 ve pass2 aşamalarını çalıştırarak sembol tablosu, makine kodu, literal ve relocation bilgilerini oluşturur. Ardından bu verileri ELF formatında kaydeder. Bu modül sayesinde çoklu dosya desteği test edilir ve linker aşaması için uygun nesne dosyaları hazırlanır.

Örnek Test Programında Linker İşlemi

main.elf

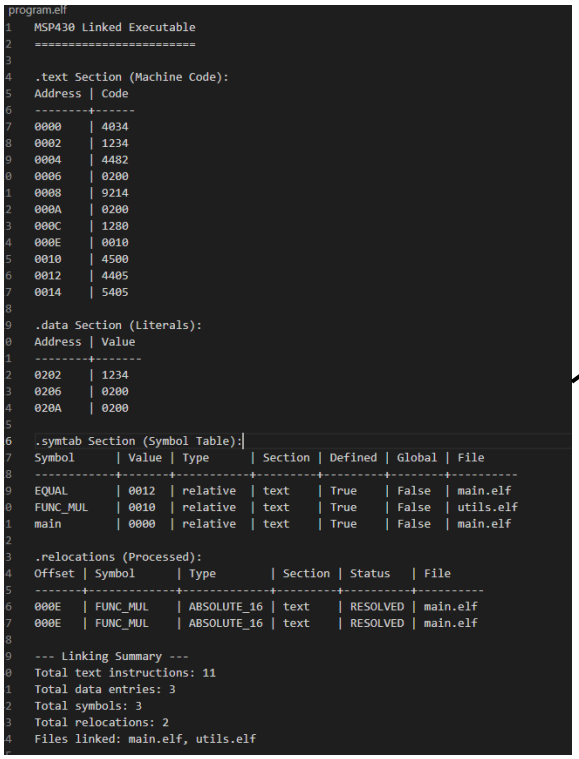
```
ELF Object File
================

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 000010
  [ 2] .data      PROGBITS        020000 000006
  [ 3] .symtab    SYMTAB          000000 000030
  [ 4] .shstrtab  STRTAB          000000 000100
  [ 5] .rel.text  REL             000000 000008

.text Section (Machine Code):
Address | Code
--------------
0000    | 4034
0002    | 1234
0004    | 4482
0006    | 0200
0008    | 9214
000A    | 0200
000C    | 1280
000E    | 0000

.data Section (Literals):
Address | Value   | Type
----------------------
0002    | 1234 | src
0006    | 0200 | dst
000A    | 0200 | src

.symtab Section (Symbol Table):
Symbol    | Value | Type      | Section | Defined | Global
-----------------------------------------------------------
FUNC_MUL  | 0000 | external  | none    | False   | False
main      | 0000 | relative  | text    | True    | False
EQUAL     | 0012 | relative  | text    | True    | False

.relocation Section:
```

utils.elf

```
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 000006
  [ 2] .data      PROGBITS        020000 000000
  [ 3] .symtab    SYMTAB          000000 000010
  [ 4] .shstrtab  STRTAB          000000 000100

.text Section (Machine Code):
Address | Code
---------------
0000    | 4500
0002    | 4405
0004    | 5405

.data Section (Literals):
Address | Value   | Type
----------------------

.symtab Section (Symbol Table):
Symbol    | Value | Type      | Section | Defined | Global
-----------------------------------------------------------
FUNC_MUL  | 0000 | relative  | text    | True    | False

Section Information:
Section | Start  | Size
------------------
text    | 0000 | 0006
data    | 0200 | 0000
bss     | 0400 | 0000
```

generate_test_elfs.py → main.elf ve utils.elf oluşturur.
linker.py → Bu ELF'leri alır, sembolleri ve adresleri birleştirir, tüm referansları çözer.
Sonuç: Tek bir çalıştırılabilir ELF dosyası → program.elf

main.elf ve utils.elf dosyaları, linker.py tarafından işlenerek tek bir çalıştırılabilir dosya olan program.elf dosyasına dönüştürülmüştür. Öncelikle linker.py, her .elf dosyasının .text, .data, .symtab ve .relocation bölümlerini satır satır okuyarak ayrıştırır. main.elf dosyasındaki .text bölümü 0x0000 adresinden başlatılırken, utils.elf'in .text kodları bunun hemen arkasına 0x0010 adresinden yerleştirilmiştir. Aynı şekilde .data bölümleri de bellekte ardışık şekilde 0x0200'den itibaren konumlandırılmıştır. Tüm semboller global bir sembol tablosunda toplanmış ve adresleri yeni konumlarına göre güncellenmiştir. main.elf dosyasındaki CALL #FUNC_MUL komutundaki FUNC_MUL sembolü tanımsız olduğu için linker, bu sembolü utils.elf dosyasında aramış ve 0x0010 adresinde tanımlandığını tespit ederek relocation işlemini gerçekleştirmiştir. Böylece



```
program.elf
MSP430 Linked Executable
========================

.text Section (Machine Code):
Address | Code
--------+------
0000    | 4034
0002    | 1234
0004    | 4482
0006    | 0200
0008    | 9214
000A    | 0200
000C    | 1280
000E    | 0010
0010    | 4500
0012    | 4405
0014    | 5405

.data Section (Literals):
Address | Value
--------+-------
0202    | 1234
0206    | 0200
020A    | 0200

.symtab Section (Symbol Table):
Symbol      | Value | Type      | Section | Defined | Global | File
------------+-------+-----------+---------+---------+--------+----------
EQUAL       | 0012  | relative  | text    | True    | False  | main.elf
FUNC_MUL    | 0010  | relative  | text    | True    | False  | utils.elf
main        | 0000  | relative  | text    | True    | False  | main.elf

.relocations (Processed):
Offset | Symbol      | Type        | Section | Status   | File
-------+-------------+-------------+---------+----------+----------
000E   | FUNC_MUL    | ABSOLUTE_16 | text    | RESOLVED | main.elf
000E   | FUNC_MUL    | ABSOLUTE_16 | text    | RESOLVED | main.elf

--- Linking Summary ---
Total text instructions: 11
Total data entries: 3
Total symbols: 3
Total relocations: 2
Files linked: main.elf, utils.elf
```

Program.elf dosyası, main.elf ve utils.elf dosyalarının linker tarafından birleştirilmesiyle oluşmuştur. Tüm semboller tek bir tabloya aktarılmış ve adresleri güncellenmiştir.Tanımsız semboller çözülmüş , relocation işlemleri tamamlandıktan sonra sectionlarla birlikte çalıştırılabilir bir çıktı dosyası olan program.elf oluşturulmuştur.

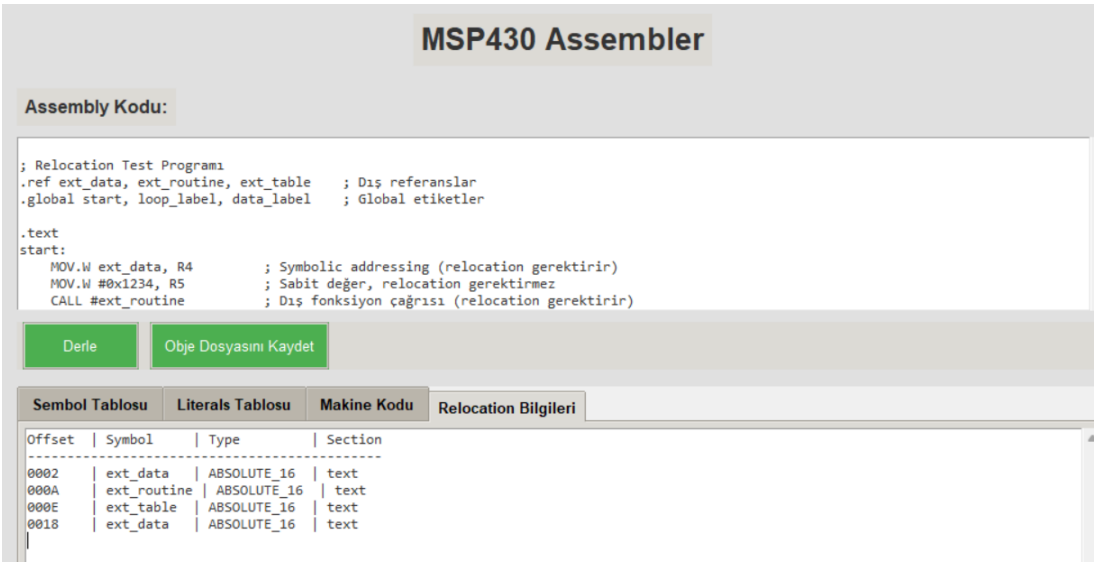
.relocation bölümündeki tüm semboller çözümlenmiş, sembol çakışması veya tanımsızlık olmadan tüm kod ve veri birleştirilmiş, tek bir yürütülebilir program.elf dosyası başarıyla oluşturulmuştur.

4.MSP430'da Relocation (Yer Değiştirme) İşlemi

Relocation işlemi, MSP430 assembler çıktılarında sembollerin (örneğin değişkenler veya etiketler) gerçek adresleri derleme anında henüz bilinmediği durumlarda devreye girer. Bu işlem, programın farklı parçaları (örneğin .text ve .data bölümleri) ayrı nesne dosyalarında derlendikten sonra, bu parçaların bellek üzerindeki gerçek konumlarının bağlayıcı (linker) tarafından belirlendiği aşamada gerçekleştirilir. MSP430 için relocation, özellikle belirsiz sembol adreslerinin çözülmesi ve kod içerisindeki bu sembollere yapılan referansların doğru adreslerle güncellenmesini içerir. Linker, her relocation girdisindeki sembol adını kullanarak global sembol tablosunda tanımlı adresi bulur ve bu adresi, ilgili .text bölümündeki makine komutlarına veya veri referanslarına yerleştirir. Böylece farklı obj dosyalarından gelen parçalar, tek bir bütün halinde doğru adreslere sahip olacak şekilde birleştirilir. MSP430 mimarisinde bu işlem, özellikle çağrı adresleri, sabit veri konumları ve atlama hedefleri gibi bellek bağımlı komutlar için kritik öneme sahiptir.

```python
def needs_relocation(operand_info, symbol_table):
    if 'label' in operand_info:
        label = operand_info['label']
        if label in symbol_table:
            if isinstance(symbol_table[label], dict):
                return not symbol_table[label].get('defined', False) or symbol_table[label].get('type') == 'external'
            else:
                return True
        else:
            return True
    return False
```

Örnek Test Programında Relocation Tablosu



.ref ile Tanımlanan Harici Semboller: MSP430 assembler'da bir sembolün başka bir dosyada tanımlı olduğunu belirtmek için .ref direktifi kullanılır. Bu direktif, derleyiciye sembolün tanımsız (undefined) olduğunu ve linkleme aşamasında başka bir kaynaktan geleceğini bildirir. Örneğin, CALL #FUNC_MUL komutu assembler tarafından işlenirken FUNC_MUL tanımsız ise ve .ref FUNC_MUL satırı varsa, bu sembol "external" olarak değerlendirilir ve daha sonra çözülmek üzere relocation tablosuna bir kayıt eklenir. Bu sayede birden fazla dosyada tanımlı semboller arasında ilişki kurulabilir.

5.MSP430 Loader

MSP430 için loader; derlenmiş programın makine kodlarını alıp, işlemcinin hafıza yapısına uygun şekilde Flash ve RAM bellek bölgelerine yerleştiren, gerekli adres dönüşümlerini (relocation) uygulayan, dış sembolleri (linking) çözen ve çalıştırılmaya hazır hale getirerek programın başlangıç noktasından çalışmasını sağlayan bir sistem yazılımıdır. MSP430 mimarisinde program kodları ve sabit veriler genellikle Flash belleğe yazılırken, çalışma sırasında değişecek veriler ve geçici bilgiler RAM bellek üzerinde tutulur. Loader bu ayrımı dikkate alarak .text (kod), .data (başlangıç değeri olan veri) ve .bss (başlangıçsız veri) gibi bölümleri uygun adreslere yükler, .bss alanını sıfırlar, yürütülecek ilk komutun adresini belirleyip kontrolü bu adrese aktararak programın çalışmasını başlatır.

```python
class MSP430ELFLoader:
    def __init__(self, memory: MSP430VirtualMemory):
        self.memory = memory

    def load_linked_elf(self, filename: str, text_base: int = 0x4400, data_base: int = 0x1C00) -> bool:
        if not os.path.exists(filename):
            print(f"HATA: '{filename}' dosyası bulunamadı.")
            return False

        print(f"ELF dosyası yükleniyor: {filename}")
```

6.Sanal Bellek

MSP430 için geliştirilen loader/assembler sisteminde sanal bellek, gerçek donanım belleğinin yapısını yazılım içinde modellemek amacıyla oluşturulan, bölümlere ayrılmış bytearray yapılarıdır. Bu yapı; kod, veri segmentlerini doğru yerlere yüklemeye, adresleme hatalarını tespit etmeye, relocation ve linking işlemlerini gerçekleştirmeye olanak sağlar. Gerçek donanıma ihtiyaç duymadan, yazılımın bellekle nasıl etkileşim kurduğunu detaylı bir şekilde simüle etmek için kullanılmıştır.

```
class MSP430VirtualMemory:
    def __init__(self):
        self.memory = {
            'SFR': bytearray(0x200),     # 0x0000-0x01FF
            'PERIPH': bytearray(0x1A00), # 0x0200-0x1BFF
            'RAM': bytearray(0x800),     # 0x1C00-0x23FF
            'FLASH': bytearray(0xBBC0), # 0x4400-0xFFBF
            'VECTORS': bytearray(0x40)  # 0xFFC0-0xFFFF
        }
```

6.1.MSP430 Bellek Bölgeleri (Örn: MSP430G2553)

a)Flash Bellek

Boyut: 16 KB Adres Aralığı: 0xC000 – 0xFFFF

Açıklama: Program kodları ve sabit veriler bu alana yerleştirilir. Elektrik kesilse bile veri silinmez.

b)RAM Bellek

Boyut: 512 Byte Adres Aralığı: 0x0200 – 0x03FF

Açıklama: Değişkenler, yığın (stack), çalışma alanları burada tutulur. Geçici bellek olarak kullanılır.

1. **Peripheral Registers (Çevresel Donanım Kayıtları)**

Boyut: Yaklaşık 256 Byte Adres Aralığı: 0x0100 – 0x01FF

2. **Special Function Registers (SFR)**

Boyut: Yaklaşık 256 Byte Adres Aralığı: 0x0000 – 0x00FF

3. **Interrupt Vector Table (Kesme Vektör Tablosu)**

Boyut: Yaklaşık 64 Byte Adres Aralığı: 0xFF80 – 0xFFFF (Flash belleğin son kısmı)

Açıklama: Tüm kesme servis rutinleri için başlangıç adresleri burada tutulur. Her kesme için 2 bayt yer ayrılır.

Örnek Test Programının Sanal Belleğe Yerleşimi

MSP430 Bellek Haritası

Adres	Değer	Bölge
0x4400	0x4034	FLASH
0x4402	0x1234	FLASH
0x4404	0x4482	FLASH
0x4406	0x0200	FLASH
0x4408	0x9214	FLASH
0x440A	0x0200	FLASH
0x440C	0x1280	FLASH
0x440E	0x0010	FLASH
0x4410	0x4500	FLASH
0x4412	0x4405	FLASH
0x4414	0x5405	FLASH
0x1E02	0x1234	RAM
0x1E06	0x0200	RAM
0x1E0A	0x0200	RAM

7) .macro ve .endm Direktifi

.macro direktifi, assembler içinde tekrar eden kod bloklarını tanımlamak için kullanılır ve bir makro tanımının başlangıcını belirtir. .macro'dan sonra gelen ilk ifade makronun adı, onu takip eden ifadeler ise parametrelerdir. Bu parametreler, makro her çağrıldığında farklı argümanlarla değiştirilerek yeniden kullanılabilir bir yapı sağlar. Makro tanımı, .endm (end macro) direktifiyle sona erer.
Bu iki direktif arasında yazılan satırlar makronun gövdesini oluşturur. Genişletme sırasında, bu gövde içinde geçen parametre isimleri makro çağrısında verilen argümanlarla yer değiştirir. Bu yapı sayesinde kod tekrarı azaltılır, okunabilirlik artar ve kodun bakım maliyeti düşer.

parse_macros() Fonksiyonu:

```
macro_table = {}
macro_expansion_counter = 0

class Macro:
    def __init__(self, name, params, body):
        self.name = name
        self.params = params
        self.body = body

def parse_macros(lines):
    """Makro tanımlarını parse eder ve macro_table'a ekler"""
    global macro_table
    i = 0
    new_lines = []  # Makro tanımları olmayan satırları sakla

    while i < len(lines):
        line = lines[i].strip()
        if line.startswith(".macro"):
            # Makro satırını virgüllerle ayır ve temizle
            macro_line = line[6:].strip()  # ".macro" kısmını çıkar
            parts = [p.strip() for p in macro_line.replace(',', ' ').split()]
```

MSP430 assembler'da tanımlanmış makroları algılamak ve bellekte saklamak için kullanılır. Makro tanımları .macro direktifiyle başlar ve .endm ile sona erer. Bu fonksiyon, satır satır assembly kodunu tarar ve .macro ile başlayan blokları tanımlayarak makro adını, parametrelerini ve gövdesini (body) alır. Her makro bir Macro nesnesi olarak saklanır ve macro_table isimli global sözlükte kayıt altına alınır. Böylece daha sonra bu makro çağrıldığında genişletilmek üzere hazır hale getirilmiş olur. Makro parametreleri opsiyonel olup, varsa sırasıyla kaydedilir. Gövdedeki satırlar temizlenmiş (strip()) haliyle kaydedilerek ilerideki eşleşme işlemlerinin sağlıklı çalışması sağlanır.

expand_macros() Fonksiyonu:

```
def expand_macros(lines):
    """Makro çağrılarını genişletir"""
    global macro_table, macro_expansion_counter
    expanded_lines = []

    for line_num, line in enumerate(lines):
        original_line = line.strip()

        # Yorum satırlarını kontrol et
        comment_pos = original_line.find('//')
        if comment_pos != -1:
            code_part = original_line[:comment_pos].strip()
            comment_part = original_line[comment_pos:]
        else:
            code_part = original_line
            comment_part = ""
```

Kaynak kodda tanımlanmış makroların çağrılarını algılar ve bunları daha önceden tanımlanan gövdelere göre genişletir. Kod satırları arasında, ilk kelimesi macro_table içinde tanımlı bir makro adına eşit olan satırlar makro çağrısı olarak değerlendirilir. Fonksiyon, bu satırdaki argümanları ilgili makro parametreleriyle eşleştirir ve makro gövdesindeki her satırı, argümanlarla parametreleri bire bir eşleyerek genişletilmiş yeni satırlar haline getirir. Bu işlem sırasında #param gibi immediate (literal) parametreler, :param: biçimindeki zorunlu yer değiştirmeler ve normal parametre isimleri sırayla doğru değerlerle değiştirilir. Ayrıca her makro çağrısı için benzersiz etiket isimleri üretmek amacıyla etiket? gibi özel semboller, etiket.1, etiket.2 gibi numaralandırılır. Sonuç olarak genişletilmiş satırlar, orijinal kodun bir parçasıymış gibi derleyiciye geri verilir.

Makro Çağrısının Genişletilmesi

```
if __name__ == "__main__":
    test_code = """
    ; === Makro Tanımı ===
    .macro LOADIMM reg, val
        MOV #val, reg
    .endm

    ; === Makro Çağrısı ile Kod ===
    .text
    start:
        LOADIMM R5, 0x1234
        LOADIMM R6, 0xABCD
        RET

    .end

    """

    run_test_program(test_code)
```

```
==== Genişletilmiş Kod ====
    ; === Makro Tanımı ===

    ; === Makro Çağrısı ile Kod ===
    .text
    start:
    MOV #0x1234, R5
    MOV #0xABCD, R6
        RET

    .end
==========================
```

Görseller de, LOADIMM makrosunun tanımı ve çağrısı sonrası genişletilmiş hali gösterilmektedir. LOADIMM R5, 0x1234 çağrısı MOV #0x1234, R5 şeklinde genişletilmiştir. Bu örnek, makro sisteminin doğru çalıştığını ve parametrelerin başarılı şekilde yerine yerleştirildiğini göstermektedir.

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*.