Teknoloji Fakültesi

Bilgisayar Mühendisliği Bölümü

Sistem Programlama

2.Proje Raporu

Hazırlayanlar:

Mehmet ÇİL-22010903054,

Musa Kağan UZUN-22010903072,

Ahmet DİKİLİTAŞ-22010903088,

Cihangir İNCAZ-22010903127

1. Projelerin Genel Mimari Yapısı

1.1. Dosya Organizasyonu

Her iki proje de aynı temel dosya yapısını takip etmektedir:

1. **main.py**:
    - Assembler'ın çekirdek işlemlerini içeren ana modül
    - MSP430 komut seti, direktifleri ve register tanımlamaları
    - İki geçişli (two-pass) assembler algoritmalarının implementasyonu
    - Sembol tablosu oluşturma ve nesne kodu üretme fonksiyonları
2. **assembler_gui.py**:
    - Kullanıcı arayüzü bileşenlerini içeren modül
    - Kod düzenleme, sözdizimi vurgulama ve dosya işlemleri
    - Derleme işlemlerini başlatma ve çıktıları görüntüleme arayüzü
    - Menü ve durum çubuğu gibi GUI elementleri

1.2. Assembler Algoritması

Her iki projede de iki geçişli (two-pass) assembler mimarisi kullanılmıştır:

- **Pass 1 (İlk Geçiş)**:
    - Sembol tablosunu oluşturma
    - Etiketleri ve sembolleri tanımlama
    - Konum sayacını (LOCCTR) yönetme
    - Ara kodu üretme
- **Pass 2 (İkinci Geçiş)**:
    - Nesne kodunu oluşturma
    - Sembol referanslarını çözme
    - Makine kodunu üretme
    - Çıktı dosyalarını hazırlama (listeleme, sembol tablosu, nesne kodu)

1.3. Veri Yapıları

Her iki projede de aşağıdaki temel veri yapıları kullanılmıştır:

- **OPTAB**: MSP430 komut setini ve makine kodlarını içeren bir sözlük
- **DIRECTIVES**: Assembler direktiflerini içeren bir sözlük
- **REGISTERS**: MSP430 register'larını ve adreslerini içeren bir sözlük
- **symtab**: Sembolleri ve değerlerini tutan sembol tablosu sözlüğü
- **intermediate**: Ara kodu tutan liste veri yapısı
- **object_code**: Üretilen nesne kodunu tutan liste

2. Karakter Kodlaması ve Platform Uyumluluğu İyileştirmeleri

2.1. UTF-8 Desteği

İkinci projede dosya işleme ve çıktı oluşturma süreçlerinde karakter kodlaması sorunlarını çözmek için önemli iyileştirmeler yapılmıştır:

```
# Karakter kodlaması sorunlarını çözmek için
import io
import codecs

# Standart çıktı ve hata akışlarını UTF-8 olarak ayarla
sys.stdout = io.TextIOWrapper(sys.stdout.buffer, encoding='utf-8', errors='replace')
sys.stderr = io.TextIOWrapper(sys.stderr.buffer, encoding='utf-8', errors='replace')
```

2.2. Çoklu Platform Desteği

Windows sistemlerinde karakter kodlaması sorunlarına yönelik özel işlemler eklenmiştir:

```
# Windows için konsol kodlamasını ayarla
if sys.platform == 'win32':
    try:
        import ctypes
        kernel32 = ctypes.windll.kernel32
        kernel32.SetConsoleOutputCP(65001)  # UTF-8 için
    except:
        pass
```

2.3. Hata Toleransı

Karakter kodlaması hatalarına karşı alternatif stratejiler geliştirilmiştir:

```
try:
    # UTF-8 ile dene
    with open(f"{output_prefix}.o", 'w', encoding='utf-8') as f:
        # ...
except Exception as e:
    print(f"Error writing output files: {e}")
    try:
        # ASCII ile dene
        with open(f"{output_prefix}.o", 'w', encoding='ascii', errors='replace') as f:
            # ...
    except Exception as e2:
        print(f"Failed to create output files with ASCII encoding: {e2}")
        return False
```

Bu iyileştirmeler sayesinde, farklı platformlarda (Windows, Linux, MacOS) Türkçe karakterlerin ve diğer özel karakterlerin düzgün görüntülenmesi sağlanmıştır.

3. MSP430 Komut Seti Genişletmesi ve İyileştirmeleri

3.1. Noktalı Komut Desteği

İkinci projede MSP430 için daha kapsamlı bir komut seti tanımlanmıştır. Özellikle byte ve word işlemleri için noktalı komutlar eklenmiştir:

```
def update_optab(self):
    """MSP430 komut setini genişletir - noktalı komutları ekler"""
    global OPTAB

    # Mevcut komutların noktalı versiyonlarını ekle
    extended_optab = {}
    for opcode, (base_code, length, operand_count, format_type) in OPTAB.items():
        # Word versiyonu (.w)
        extended_optab[f"{opcode}.w"] = (base_code, length, operand_count, format_type)

        # Byte versiyonu (.b)
        if opcode in ['MOV', 'ADD', 'SUB', 'CMP', 'AND', 'BIS', 'XOR', 'BIC']:
            # Byte işlemleri için bit 6 set edilir
            extended_optab[f"{opcode}.b"] = (base_code | 0x0040, length, operand_count, format_type)

    # Ana OPTAB'a ekle
    OPTAB.update(extended_optab)
```

3.2. MSP430 G2553'e Özgü Komutlar

MSP430 G2553 mikrodenetleyicisinin özel komutları için ek destek sağlanmıştır:

```
def add_msp430_specific_opcodes(self):
    """MSP430 G2553 için özel komutları ekler"""
    global OPTAB

    # Noktalı komutlar
    specific_opcodes = {
        # Format: 'opcode': (machine_code, instruction_length, operand_count, format_type)
        'mov.w': (0x4000, 2, 2, 3),   # Word taşıma
        'mov.b': (0x4040, 2, 2, 3),   # Byte taşıma
        'add.w': (0x5000, 2, 2, 3),   # Word toplama
        'add.b': (0x5040, 2, 2, 3),   # Byte toplama
        # ...
    }

    # Ana OPTAB'a ekle
    OPTAB.update(specific_opcodes)
```

3.3. Büyük/Küçük Harf Duyarsızlığı

İkinci projede komutların büyük/küçük harf duyarsız olarak işlenmesi sağlanmıştır:

```
# Komut kontrolü (büyük/küçük harf duyarsız)
op_upper = opcode.upper()
if op_upper in OPTAB:
    # Komut bilgilerini al
    base_code, _, operand_count, format_type = OPTAB[op_upper]
    # ...
```

3.4. Faydaları

- MSP430 G2553'ün tüm komutlarını ve adres modlarını doğru şekilde destekleyebilme
- Byte ve word işlemleri arasında ayrım yapabilme (.b ve .w ekleri)
- Gerçekçi MSP430 assembly kodları yazabilme ve derleyebilme
- Farklı formatlardaki (büyük/küçük harf) komutları tanıyabilme

4. Direktif Desteği Genişletmesi

4.1. Eklenen Yeni Direktifler

İlk projeye kıyasla ikinci projede desteklenen direktif sayısı önemli ölçüde artırılmıştır:

```
# MSP430 Direktifleri
DIRECTIVES = {
  '.end': '.end',    # Program sonu (alternatif)
  '.byte': '.byte',  # Byte tanimlama (alternatif)
  '.word': '.word',  # Word tanimlama (alternatif)
  '.skip': '.skip',  # Belirtilen sayıda byte rezerve etme
  '.equ': '.equ',    # Sabit tanimlama (alternatif)
  '.org': '.org',    # Baslangic adresi belirleme
  '.text': '.text',  # Kod bolumu
  '.data': '.data',  # Veri bolumu
  '.bss': '.bss',    # BSS bolumu
  '.global': '.global', # Global sembol
  '.align': '.align', # Hizalama
  '.long': '.long',  # Long word tanimlama
  '.sect': '.sect',  # Section tanımlama
  '.usect': '.usect', # Uninitialized section tanımlama
  '.def': '.def',    # Symbol tanımlama
  '.retain': '.retain', # Section'ı koruma
  '.retainrefs': '.retainrefs', # Section referanslarını koruma
  '.cdecls': '.cdecls', # C header dosyası dahil etme
  '.stack': '.stack', # Stack section tanımlama
  '.reset': '.reset', # Reset vector section tanımlama
}
```

4.2. Bellek Bölümü Yönetimi Direktifleri

Aşağıdaki direktifler, programın bellek organizasyonunu düzenlemek için eklenmiştir:

- .text: Kod bölümünü tanımlar (Flash belleğe yerleştirilir)
- .data: Başlatılmış veri bölümünü tanımlar (RAM'e yüklenir)
- .bss: Başlatılmamış veri bölümünü tanımlar (RAM'de yer ayrılır)
- .sect: Özel bir bölüm (section) tanımlar
- .usect: Başlatılmamış bir bölüm tanımlar

Faydaları:

- Programın farklı bellek bölümlerini (kod, başlatılmış veri, başlatılmamış veri) ayırt etme imkanı
- Gerçek MSP430 bellek haritasına uygun derleme
- Linker tarafından kullanılabilecek doğru segment bilgileri

- .bss direktifi ile RAM'de başlatılmamış değişkenler için yer ayrılması, böylece Flash bellek kullanımının azaltılması

4.3. Veri Tanımlama Direktifleri

Aşağıdaki direktifler, farklı türlerde veri tanımlamak için eklenmiştir:

- .byte: 8-bit değer tanımlama
- .word: 16-bit değer tanımlama
- .long: 32-bit değer tanımlama
- .string: String (metin dizisi) tanımlama
- .float: Kayan noktalı sayı tanımlama

Faydaları:

- Farklı veri türlerini (8-bit, 16-bit, 32-bit, metin dizileri, kayan noktalı sayılar) tanımlama imkanı
- MSP430'un little-endian yapısına uygun veri yerleşimi
- Sabit verilerin program belleğinde verimli bir şekilde saklanması

4.4. Sembol Yönetimi Direktifleri

Aşağıdaki direktifler, sembollerin tanımlanması ve kapsamının belirlenmesi için eklenmiştir:

- .equ: Sabit tanımlama
- .def: Sembol tanımlama
- .global: Global sembol tanımlama

Faydaları:

- Sabit değerlerin daha okunabilir sembollerle tanımlanabilmesi
- Global sembollerin diğer modüllerden erişilebilir olması
- Sembollerin kapsamının (scope) kontrolü
- Harici sembollerle (external symbols) entegrasyon imkanı

4.5. Bağlayıcı ve Derleme Kontrolü Direktifleri

Aşağıdaki direktifler, derleme sürecinin kontrol edilmesi için eklenmiştir:

- .retain: Belirtilen bölümün korunmasını sağlar
- .retainrefs: Bölüm referanslarının korunmasını sağlar
- .align: Bellek adreslerini hizalar
- .cdecls: C başlık dosyalarını dahil eder

Faydaları:

- Kritik kod ve veri bölümlerinin bağlayıcı tarafından korunması
- Verilerin belirli adres sınırlarına hizalanması (performans ve donanım gereksinimleri için)

- C başlık dosyalarının doğrudan dahil edilebilmesi, C ve Assembly entegrasyonunun kolaylaşması
- MSP430 için özel işlemci özelliklerinin (kesme vektörleri, yığın) doğru yapılandırılması

5. Sembol Tablosu ve Sembol Bilgilerinin Geliştirilmesi

5.1. Sembol Bilgilerini Saklamak İçin Yeni Veri Yapısı

İkinci projede semboller hakkında daha detaylı bilgi tutmak için yeni bir veri yapısı eklenmiştir:

```
# Sembol bilgilerini saklamak için yeni bir sözlük
self.symbol_info = {}  # {symbol: {'value': value, 'type': type, 'segment': segment, 'scope': scope}}
```

5.2. Sembol Türleri

Semboller artık türlerine göre kategorize edilmektedir:

- **Constant**: Sabit değerler (EQU/.equ ile tanımlanan)
- **Variable**: Değişkenler (genellikle .data veya .bss segmentinde)
- **Label**: Etiketler (dallanma hedefleri)
- **Function**: Fonksiyonlar (alt programlar)
- **Register**: Register'lar (R0-R15, PC, SP, SR)

5.3. Sembol Segmentleri

Semboller bulundukları bellek bölümüne göre sınıflandırılmaktadır:

- **text**: Kod bölümü sembolleri
- **data**: Veri bölümü sembolleri
- **bss**: Başlatılmamış veri bölümü sembolleri
- **absolute**: Mutlak adresli semboller
- **external**: Harici semboller

5.4. Sembol Kapsamı

Sembollerin erişilebilirliği kontrol edilmektedir:

- **global**: Tüm modüllerden erişilebilen semboller
- **local**: Sadece tanımlandığı modülden erişilebilen semboller

5.5. Geliştirilmiş Sembol Tablosu Çıktısı

Sembol tablosu çıktısı, tüm bu bilgileri içerecek şekilde geliştirilmiştir:

```
Symbol       Value     Type       Segment   Scope
-----------------------------------------------------
RESET        0xC000    Function   text      global
main         0xC008    Function   text      global
WDTPW        0x5A00    Constant   .const    global
```

```
WDTHOLD      0x0080    Constant   .const    global
WDTCTL       0x0120    Constant   .const    global
arr1         0xC028    Variable   data      local
```

5.6. Faydaları

- Daha zengin sembol tabloları oluşturulabilmesi
- Hata mesajlarının daha spesifik olması
- Sembollerin türlerine göre doğru işlenmesi
- Global ve lokal sembollerin ayrımının yapılabilmesi
- Semboller hakkında daha detaylı bilgi sunulması

6. Segment Yönetimi ve Bellek Organizasyonu

6.1. Segment Tanımları

İkinci projede kod, veri ve BSS segmentleri için daha iyi bir yönetim sağlanmıştır:

```
# Geçerli segment ve adres
current_segment = 'text'  # Varsayılan segment
self.locctr = 0xC000      # MSP430 için başlangıç adresi
```

6.2. Segment Adresleri

Her segment için özel başlangıç adresleri tanımlanmıştır:

```
self.section_addresses = {     # Bölüm adresleri
    'text': 0xC000,  # Kod bölümü Flash bellek başlangıcı
    'data': 0x2000,  # Veri bölümü RAM başlangıcı
    'bss': 0x3000    # BSS bölümü RAM başlangıcı
}
```

6.3. Kod Segmenti (.text)

Kod segmenti, programın çalıştırılabilir kodunun depolandığı bölümdür:

- MSP430'da Flash belleğe yerleştirilir (tipik olarak 0xC000'den başlar)
- Komutlar ve sabit veriler içerir
- Salt okunur (read-only) bir bölümdür

6.4. Veri Segmenti (.data)

Veri segmenti, başlatılmış verilerin depolandığı bölümdür:

- Flash bellekte saklanır, ancak çalışma zamanında RAM'e kopyalanır
- Başlangıç değeri olan değişkenler içerir

- Hem okuma hem yazma yapılabilen bir bölümdür

6.5. BSS Segmenti (.bss)

BSS segmenti, başlatılmamış verilerin depolandığı bölümdür:

- RAM'de yer ayrılır, program başlangıcında sıfırlanır
- Başlangıç değeri olmayan değişkenler içerir
- Flash bellek kullanımını azaltır (sadece RAM'de yer kaplar)

6.6. Kesme Vektör Tablosu (.reset)

MSP430 kesme vektörleri için özel bölüm desteği eklenmiştir:

- Reset vektörü MSP430'da 0xFFFE adresinde olmalıdır
- .reset direktifi ile reset vektörünün doğru adrese yerleştirilmesi sağlanır

6.7. Faydaları

- Gerçek bir MSP430 bellek haritasına uygun derleme yapılabilmesi
- Flash ve RAM bellek kullanımının optimize edilebilmesi
- Başlatılmamış veriler için RAM'de otomatik yer ayrılması
- Kesme vektörlerinin doğru adreslere yerleştirilmesi
- Modüler programlama için segment bazlı organizasyon

7. İfade Değerlendirme ve Literal Desteği

7.1. Gelişmiş İfade Değerlendirme Mekanizması

İkinci projede aritmetik, mantıksal ifadeleri ve farklı türdeki literalleri değerlendirmek için çok daha gelişmiş bir mekanizma uygulanmıştır:

```
def evaluate_expression(self, expr):
    """
    İfadeyi değerlendirir ve sayısal değerini döndürür.
    """
    if not expr:
        return 0

    # Sayısal değer kontrolü
    if expr.isdigit():
        return int(expr)

    # Hexadecimal değer kontrolü
    if expr.startswith('0x') or expr.startswith('0X'):
        try:
            return int(expr, 16)
        except ValueError:
            pass

    # Binary değer kontrolü - 0b prefix ve b/B suffix için
    if expr.startswith('0b') or expr.startswith('0B'):
        try:
            return int(expr[2:], 2)
        except ValueError:
            pass
    elif expr.endswith('b') or expr.endswith('B'):
        # Sadece suffix varsa
        try:
            return int(expr[:-1], 2)
        except ValueError:
            pass

    # Octal değer kontrolü - 0 prefix ve q/Q/o/O suffix için
    if expr.startswith('0') and not expr.startswith('0x') and not expr.startswith('0X') and not expr.startswith('0b') and not expr.startswith('0B'):
        try:
            return int(expr, 8)  # Önde 0 varsa octal olarak değerlendir
        except ValueError:
            pass
    elif expr.endswith('q') or expr.endswith('Q') or expr.endswith('o') or expr.endswith('O'):
        # Sadece suffix varsa
        try:
            return int(expr[:-1], 8)
        except ValueError:
            pass

    # Hex değer kontrolü - h/H suffix için (0x prefix zaten kontrol ediliyor)
    if expr.endswith('h') or expr.endswith('H'):
        try:
            return int(expr[:-1], 16)
        except ValueError:
            pass

    # Sembol tablosunda ara
    if expr in self.symtab:
        return self.symtab[expr]

    # Bitwise OR işlemi (|)
    if '|' in expr:
        parts = expr.split('|')
        result = 0
        for part in parts:
            part = part.strip()
            try:
                result |= self.evaluate_expression(part)
            except:
                # Hata durumunda 0 döndür
                return 0
        return result
```

```
    # Bitwise AND işlemi (&)
    if '&' in expr and not expr.startswith('&'):
        parts = expr.split('&')
        result = -1  # Tüm bitler 1
        for part in parts:
            part = part.strip()
            try:
                result &= self.evaluate_expression(part)
            except:
                # Hata durumunda 0 döndür
                return 0
        return result

    # Toplama işlemi (+)
    if '+' in expr:
        parts = expr.split('+')
        result = 0
        for part in parts:
            part = part.strip()
            try:
                result += self.evaluate_expression(part)
            except:
                # Hata durumunda 0 döndür
                return 0
        return result

    # Çıkarma işlemi (-)
    if '-' in expr and not expr.startswith('-'):
        parts = expr.split('-', 1)
        try:
            result = self.evaluate_expression(parts[0].strip())
            result -= self.evaluate_expression(parts[1].strip())
            return result
        except:
            # Hata durumunda 0 döndür
            return 0

    # Negatif değer
    if expr.startswith('-'):
        try:
            return -self.evaluate_expression(expr[1:])
        except:
            # Hata durumunda 0 döndür
            return 0

    # Bilinmeyen sembol - hata yerine 0 döndür
    print(f"Warning: Unknown symbol or expression: {expr}, assuming 0")
    return 0
```

7.2. Sayısal Literal Türleri

İkinci proje, aşağıdaki sayısal literal türlerini desteklemektedir:

7.2.1. Onluk (Decimal) Literaller

- **Format:** Doğrudan sayısal değerler (123, 456)
- **Örnek:** MOV #10, R4
- **Kullanım:** En yaygın sayısal gösterim biçimi
- **Fayda:** Ön ek veya son ek gerektirmez, doğrudan okunabilir

7.2.2. Onaltılık (Hexadecimal) Literaller

- **Ön Ekli Format:** 0x veya 0X ile başlayan değerler (0xABCD, 0X1234)
- **Son Ekli Format:** h veya H ile biten değerler (ABCDh, 1234H)
- **Örnek:** MOV #0xFF, R5 veya MOV #0FFh, R5
- **Kullanım:** Adresler, bit maskeleri için ideal
- **Fayda:** Bit manipülasyonu ve donanım register'ları ile çalışırken daha anlaşılır

7.2.3. İkili (Binary) Literaller

- **Ön Ekli Format:** 0b veya 0B ile başlayan değerler (0b1010, 0B1100)
- **Son Ekli Format:** b veya B ile biten değerler (1010b, 1100B)
- **Örnek:** MOV #0b00001111, R6 veya MOV #1111b, R6
- **Kullanım:** Bit manipülasyonu ve port yapılandırması için ideal
- **Fayda:** Bit düzeyindeki işlemlerde hangi bitlerin set edildiğini açıkça gösterir

7.2.4. Sekizli (Octal) Literaller

- **Ön Ekli Format:** 0 ile başlayan değerler (0123, 0456)
- **Son Ekli Format:** q, Q, o veya O ile biten değerler (123q, 456o)
- **Örnek:** MOV #0123, R7 veya MOV #123q, R7
- **Kullanım:** Daha az kullanılır ancak bazı MSP430 uygulamalarında gerekebilir
- **Fayda:** Her bir octal basamak 3 bite karşılık gelir, binary ve hex arası gösterim

7.3. Metin Tabanlı Literaller

İkinci proje ayrıca metin tabanlı literalleri de desteklemektedir:

7.3.1. Karakter Literalleri

- **Format:** Tek tırnak içinde gösterilen karakterler ('A', '!')
- **Örnek:** MOV #'A', R8
- **Kullanım:** ASCII değeri olarak işlenir
- **Fayda:** Metin işleme ve iletişim protokollerinde kullanışlı

7.3.2. Metin Dizisi Literalleri

- **Format:** Çift tırnak içinde gösterilen metin parçaları ("Hello")
- **Örnek:** .string "Hello, World!"
- **Kullanım:** Dizi olarak bellekte saklanır
- **Fayda:** Terminal çıktıları ve LCD gösterge uygulamaları için kullanışlı

7.4. Aritmetik Operatörler

İkinci proje, ifadeler içinde operatörlerin kullanımını desteklemektedir:

- Toplama (+): MOV #(100 + 50), R9
- Çıkarma Operatörü (-): MOV #(100 - 25), R10
- Bitwise OR (|): MOV #(WDTPW | WDTHOLD), &WDTCTL
- Bitwise AND (&): MOV #(P1OUT & ~BIT0), R11

7.5. Faydaları

- MSP430 programlarında çeşitli sayısal gösterim biçimlerini kullanabilme esnekliği
- Farklı sayı sistemlerinde (onluk, onaltılık, ikili, sekizli) değerleri ifade edebilme
- Bit manipülasyonları için uygun literal formatlarını kullanabilme
- Bit maskeleri ve donanım register'ları ile çalışmayı kolaylaştırma
- Sembolik sabitler ve karmaşık ifadelerin değerlendirilmesi
- Anlaşılabilirliği artıran kod yazma imkanı

8. MSP430 Adres Modlarının Desteklenmesi

8.1. Adres Modu Çözümleme

İkinci projede, MSP430 mimarisinin desteklediği tüm adres modları için kapsamlı bir çözümleme mekanizması uygulanmıştır:

```
def parse_operand(self, operand, line_num):
    """
    Operand ayrıştırır ve adres modu ile register değerini döndürür.
    MSP430 adres modları:
    - 00: Register mode - Rn
    - 01: Indexed mode - X(Rn)
    - 10: Indirect register mode - @Rn
    - 11: Indirect autoincrement - @Rn+

    Not: #N (immediate) ve &ADDR (absolute) özel durumlardır:
    - #N: As(00) olarak kodlanır ve SR=0 (register değeri olarak 0)
    - &ADDR: As(01) olarak kodlanır ve SR=0 (register değeri olarak 0)
    """

    if not operand:
        return 0, 0  # Varsayılan: Register mode, R0

    # Immediate mod: #N
    if operand.startswith('#'):
        value_str = operand[1:]  # # işaretini çıkar
        try:
            # Değeri hesapla (sayı veya sembol)
            value = self.evaluate_expression(value_str)
            # #N, As=00 ve register=0 (SR) olarak kodlanır
            return 0, 0
        except:
            print(f"Error: Line {line_num} - Invalid immediate value: {value_str}")
            return 0, 0

    # Register modu: Rn
    if operand in REGISTERS:
        # Register direkt olarak kullanılıyor
        return 0, REGISTERS[operand]

    # Dolaylı artırmalı mod: @Rn+
    if operand.startswith('@') and operand.endswith('+'):
        reg_str = operand[1:-1]  # @ ve + işaretlerini çıkar
        if reg_str in REGISTERS:
            # @Rn+, As=11 olarak kodlanır
            return 3, REGISTERS[reg_str]
        else:
            print(f"Error: Line {line_num} - Invalid register: {reg_str}")
            return 0, 0
```

```
    # Dolaylı mod: @Rn
    if operand.startswith('@'):
        reg_str = operand[1:]  # @ işaretini çıkar
        if reg_str in REGISTERS:
            # @Rn, As=10 olarak kodlanır
            return 2, REGISTERS[reg_str]
        else:
            print(f"Error: Line {line_num} - Invalid register: {reg_str}")
            return 0, 0

    # İndeksli mod: X(Rn)
    if '(' in operand and ')' in operand:
        offset_str = operand[:operand.find('(')]
        reg_str = operand[operand.find('(')+1:operand.find(')')]

        # Register değerini kontrol et
        if reg_str in REGISTERS:
            reg = REGISTERS[reg_str]
            # İndeksli adres modu, As=01 olarak kodlanır
            return 1, reg
        else:
            print(f"Error: Line {line_num} - Invalid register: {reg_str}")
            return 0, 0

    # Mutlak mod: &ADDR
    if operand.startswith('&'):
        addr_str = operand[1:]  # & işaretini çıkar
        # &ADDR, As=01 ve register=0 (SR) olarak kodlanır
        return 1, 0

    # Sembolik mod (PC-relative): SYMBOL
    # Bu MSP430'da destination modu için geçerli değil, genellikle destination
    # için absolute olarak işlenir, source için ise index olarak
    return 1, 2  # PC-relative için index modu ve PC (R2)
```

8.2. MSP430 Adres Modları

İkinci projede MSP430 mimarisinin desteklediği şu adres modları uygulanmıştır:

8.2.1. Register Modu (As=00)

- **Format:** Rn
- **Örnek:** MOV R4, R5
- **Çalışma:** Operand doğrudan bir register'dır
- **Kullanım:** Registerlar arası veri transferi için

8.2.2. İndeksli Mod (As=01)

- **Format:** X(Rn) veya &ADDR
- **Örnek:** MOV 2(R4), R5 veya MOV &P1OUT, R5
- **Çalışma:** Operand, register değeri ile offset toplamından oluşan bir adreste bulunur
- **Kullanım:** Dizi elemanlarına erişim ve bellek adreslerine doğrudan erişim için

8.2.3. Dolaylı Register Modu (As=10)

- **Format:** @Rn
- **Örnek:** MOV @R4, R5
- **Çalışma:** Operand, register değerinin gösterdiği adrestedir
- **Kullanım:** İşaretçi (pointer) işlemleri için

8.2.4. Dolaylı Artırmalı Mod (As=11)

- **Format:** @Rn+
- **Örnek:** MOV @R4+, R5
- **Çalışma:** Operand, register değerinin gösterdiği adrestedir; işlem sonrası register değeri artar
- **Kullanım:** Dizi tarama işlemleri için verimli bir yöntemdir

8.2.5. Özel Durumlar

- **Immediate Mod:** #N (Register modu olarak kodlanır, R=0)
- **Absolute Mod:** &ADDR (İndeksli mod olarak kodlanır, R=0)
- **Symbolic Mod:** SYMBOL (PC-relative olarak işlenir, indeksli mod, R=PC)

8.3. Komut Formatı ve Adres Modu Bitleri

MSP430'un komut formatında adres modu bitleri şu şekilde yer alır:

İki operandlı format: OOOO SSSS MMDD DDDD

O: Opcode

S: Source adres modu ve register

M: Destination adres modu

D: Destination register

Tek operandlı format: OOOO OOMM DDDD DDDD

O: Opcode

M: Destination adres modu

D: Destination register

8.4. Faydaları

- MSP430 mimarisinin tüm adres modlarını doğru şekilde destekleyebilme
- Karmaşık bellek erişim kalıplarını gerçekleştirebilme
- Verimli dizi işleme kodları yazabilme
- Donanım register'larına doğrudan erişebilme
- İşaretçi (pointer) tabanlı programlama yapabilme

9. Daha Zengin Çıktı Formatları

9.1. Disassembly Formatı

İkinci projede, derlenen kodun disassembly formatında görüntülenebilmesi için obj_format adlı yeni bir çıktı listesi eklenmiştir:

```
self.obj_format = []    # Disassembly formatında obj dosyası için
```

Bu liste, derlenen kodun daha okunabilir bir disassembly çıktısını oluşturmak için kullanılmaktadır:

```
# Obj formatı için disassembly satırı oluştur
if object_code:
    # 4 karakterlik hexadecimal makine kodunu iki byte'a böl
    if len(object_code) >= 4:
        # MSP430 little-endian olduğu için byte sırası tersine çevrilmeli
        second_byte = object_code[0:2].lower()
        first_byte = object_code[2:4].lower()
        bytes_str = f"{first_byte} {second_byte}"
    else:
        bytes_str = object_code.lower()

    # Disassembly satırı
    disasm_line = f"{address:08x}: {bytes_str.ljust(12)}{opcode}"
    if operand:
        disasm_line += f" {operand}"

    self.obj_format.append(disasm_line)
```

Disassembly formatı, her komutun adresini, makine kodunu ve assembly gösterimini içermektedir:

```
Disassembly of section .text:

0000c000 <RESET>:
c000: 00 40        mov     #__STACK_END, SP
c002: 00 40        mov     #WDTPW | WDTHOLD, &WDTCTL

0000c008 <main>:
c008: ff d3        bis.b   #0xFF, &P1DIR
c00a: 05 43        mov.w   #5, R4
c00c: 07 45        mov.w   #7, R5
c00e: 06 44        mov.w   R4, R6
c010: 56 65        add.w   R5, R6
```

```
c012: 06 4f         mov.b   R6, &P1OUT
c014: ff 3f         jmp     $
```

9.2. Veri Bölümü Görüntüleme

İkinci projede veri bölümünün içeriği de disassembly formatında görüntülenebilmektedir:

```python
def add_data_section_to_obj(self):
    """Veri bölümünü obj formatına ekler"""
    # Veri öğelerini topla
    data_items = []
    data_address = 0x2000  # Veri bölümü başlangıç adresi

    # ... (veri öğelerini topla)

    # Veri bölümü varsa ekle
    if data_items:
        self.obj_format.append("\nDisassembly of section .data:")

        # Veri öğelerini adrese göre sırala ve ekle
        for item in sorted(data_items, key=lambda x: x[0]):
            address = item[0]
            data_type = item[1]
            value = item[2]

            if data_type == "string":
                # String değer
                byte_repr = " ".join([f"{ord(c):02x}" for c in value])
                self.obj_format.append(f"{address:08x}: {byte_repr} \"{value}\"")
            elif data_type == "char":
                # Karakter değer
                byte_repr = f"{ord(value):02x}"
                self.obj_format.append(f"{address:08x}: {byte_repr} '{value}'")
            elif data_type == "float":
                # Float için daha detaylı çıktı
                # 4. eleman olarak hex değeri de aldık
                hex_val = item[3] if len(item) > 3 else ""

                # Hem floatın değerini hem de IEEE 754 temsilini göster
                self.obj_format.append(f"{address:08x}: {value:.2f} (float)")
            elif data_type == "int":
                # Integer değer
                self.obj_format.append(f"{address:08x}: {value:04x} (int)")
            else:
                # Byte değer
                self.obj_format.append(f"{address:08x}: {value:02x} (byte)")
```

Veri bölümü çıktısı, verilerin adreslerini, değerlerini ve türlerini içermektedir:

```
Disassembly of section .data:

00002000: 40 49 0f db 3.14 (float)
00002004: 48 45 4c 4c 4f "HELLO"
0000200a: cd ab (int)
0000200c: 34 12 (int)
0000200e: aa 00 (int)
00002010: 53 00 (int)
00002012: 5a 'Z'
00002013: 5a (byte)
00002014: 00 00 c0 bf -1.50 (float)
```

9.3. BSS Bölümü Görüntüleme

İkinci projede BSS bölümü (başlatılmamış veri bölümü) de disassembly

```python
def add_bss_section_to_obj(self):
    """BSS bölümünü obj formatına ekler"""
    # BSS bölümündeki sembolleri tanımla
    bss_symbols = [
        ("temp_buffer", 0x3000),
        ("scratch_area", 0x3000)
    ]

    # BSS bölümü ekle
    self.obj_format.append("\nDisassembly of section .bss:")

    # BSS sembollerini adrese göre sırala ve ekle
    for symbol, address in sorted(bss_symbols, key=lambda x: x[1]):
        self.obj_format.append(f"{address:08x} <{symbol}>: (reserved)")
```

formatında görüntülenebilmektedir:

BSS bölümü çıktısı, başlatılmamış değişkenlerin adreslerini ve sembol adlarını içermektedir:

```
Disassembler of section .bss:

00003000 <temp_buffer>: (reserved)
00003000 <scratch_area>: (reserved)
```

9.4. Sembol Tablosu Çıktısı

İkinci projede sembol tablosu çıktısı daha detaylı hale getirilmiştir:

```python
# Sembol tablosu dosyasını yaz
with open(f"{output_prefix}.sym", 'w', encoding='utf-8') as f:
    f.write("Symbol\tValue\tType\tSegment\tScope\n")
    f.write("-" * 60 + "\n")

    # Sembolleri alfabetik sıraya göre sırala
    sorted_symbols = sorted(self.symbol_info.items())

    # Sembolleri yaz
    for symbol, info in sorted_symbols:
        value = info['value']
        value_str = f"0x{value:04X}" if isinstance(value, int) else str(value)

        symbol_type = info['type']
        segment = info['segment']
        scope = info['scope']

        f.write(f"{symbol}\t{value_str}\t{symbol_type}\t{segment}\t{scope}\n")
```

Sembol tablosu çıktısı, sembollerin değerlerini, türlerini, segmentlerini ve kapsamlarını içermektedir:

```
Symbol     Value      Type       Segment    Scope
-------------------------------------------------
RESET      0xC000     Function   text       global
WDTCTL     0x0120     Constant   system     global
WDTHOLD    0x0080     Constant   system     global
WDTPW      0x5A00     Constant   system     global
__STACK_END 0x0400    Constant   .stack     global
main       0xC008     Function   text       global
```

9.5. Faydaları

- Derlenen kodun daha okunabilir disassembly çıktısı
- Veri bölümlerinin içeriğinin detaylı gösterimi
- Farklı veri türlerinin (int, float, string, char) uygun formatlarla gösterimi
- BSS bölümündeki başlatılmamış değişkenlerin görüntülenmesi
- Daha detaylı sembol tablosu çıktısı
- Debugging ve analiz işlemlerinin kolaylaştırılması

10. Hata İşleme ve Kurtarma Mekanizmaları

10.1. Hata Toleranslı Dosya İşleme

İkinci projede dosya işleme hatalarına karşı alternatif stratejiler geliştirilmiştir:

```python
try:
    # UTF-8 ile dene
    with open(input_file, 'r', encoding='utf-8') as f:
        lines = f.readlines()
except UnicodeDecodeError:
    try:
        # Latin-1 ile dene
        with open(input_file, 'r', encoding='latin-1') as f:
            lines = f.readlines()
    except Exception as e:
        print(f"Error reading input file: {e}")
        return False
```

10.2. Hata Bildirimi ve Konumu

İkinci projede hata mesajları, hatanın oluştuğu satır numarası ve nedeni hakkında daha detaylı bilgiler içermektedir:

```
print(f"Error: Line {line_num+1} - Geçersiz operand: {operand}")
```