MSP430 Assembler Tasarımı
Proje Raporu

Takım Üyeleri:

Adem COŞKUN – B200109003

Emine KAYIT – B210109372

Sedanur PEKER - 22010903060

Zeliha POLAT - 22010903069

Giriş

Bu projede, MSP430 mimarisi için temel assembler işlemlerini gerçekleştiren bir uygulama geliştirilmiştir. Uygulama, yüksek seviyeli Python dili kullanılarak Tkinter arayüzü üzerinden kullanıcıya görsel bir ortam sağlamaktadır. Kullanıcı, assembly komutlarını girerek, bu komutları makine diline çevirebilmekte ve sonucu bir dosyaya kaydedebilmektedir.

Amaç

MSP430 mikrodenetleyicisine uygun sadeleştirilmiş bir assembler geliştirmek. Kullanıcıdan assembly formatında giriş alıp, bu girdileri opcode tablosu ve adresleme modları kullanarak makine koduna çevirmek.

Kullanılan Teknolojiler

- **Python 3.x:** Esnek ve hızlı geliştirme imkanı sağlar.
- **Tkinter:** Basit ve etkili bir GUI kütüphanesi.
- **hashlib:** MD5 ile etiketlerin hashlenmesi için.
- **os:** Dosya işlemleri için.

Kodun Açıklaması:

1)Opcode Tablosu

```python
# MSP430 Opcode Tablosu (Sadeleştirilmiş)
opcode_tablosu = {
    "MOV": "4",     # Veri taşıma
    "ADD": "5",     # Toplama
    "SUB": "8",     # Çıkarma
    "CMP": "9",     # Karşılaştırma
    "AND": "F",     # Mantıksal VE
    "XOR": "E",     # Mantıksal ÖZEL VEYA
    "JNE": "20",    # Eşit değilse dallan
    "JEQ": "24",    # Eşitse dallan
    "JMP": "3C",    # Koşulsuz dallanma
    "CALL": "128", # Alt program çağrısı
    "RET": "130",  # Alt programdan dönüş
    "PUSH": "12",  # Yığına ekleme
    "POP": "13",    # Yığından çıkarma
    "NOP": "4303"  # Boş işlem
}
```

Yandaki tablo, desteklenen MSP430 komutlarının opcode karşılıklarını içerir. Bu sayede metinsel komutlar makine koduna çevrilebilmektedir.

Opcode tablosu, MSP430'un üç temel komut türünü destekler:

- Çift Operandlı Komutlar: MOV, ADD, SUB gibi komutlar iki operand alır (kaynak ve hedef). Örneğin, MOV R5, R6 (R5'teki değeri R6'ya kopyalar).
- Tek Operandlı Komutlar: RRA, SXT, PUSH gibi komutlar tek bir operand üzerinde işlem yapar. Örneğin, PUSH R5 (R5'teki değeri yığına ekler).

- Zıplama Komutları: JNE, JMP gibi komutlar program akışını kontrol eder. Örneğin, JNE LOOP (eşit değilse LOOP etiketine zıplar).

2)Adresleme Modları

```python
# Adresleme modları
addressing_modes = {
    "Rn": 0x0, "@Rn": 0x2, "@Rn+": 0x3, "#N": 0x3, "X(Rn)": 0x1, "&ADDR": 0x1
}
```

MSP430'da farklı adresleme türlerine karşılık gelen kodları tanımlar. Kod çözümlerken hangi adresleme türü kullanıldığını belirlemek için kullanılır.

Adresleme Modu	Değer	Açıklama
Rn	0x0	Doğrudan kaydedici (örneğin, R5)
@Rn	0x2	Dolaylı (örneğin, @R5)
@Rn+	0x3	Otomatik artırmalı dolaylı (örneğin, @R5+)
#N	0x3	Sabit değer (örneğin, #10)
X(Rn)	0x1	İndeksli (örneğin, 2(R5))
&ADDR	0x1	Mutlak adres (örneğin, &0x0200)

3)Hash Tablosu

```python
# Hash tablosu için sınıf
class HashSymbolTable:
    def __init__(self):  # ← DOĞRU
        self.table = {}
        self.original_labels = {}


    def hash_key(self, key):
        return hashlib.md5(key.encode()).hexdigest()

    def add(self, key, value):
        hashed_key = self.hash_key(key)
        self.table[hashed_key] = value
        self.original_labels[hashed_key] = key  # Orijinal etiketi sakla

    def get(self, key):
        hashed_key = self.hash_key(key)
        return self.table.get(hashed_key)

    def exists(self, key):
        hashed_key = self.hash_key(key)
        return hashed_key in self.table

    def clear(self):
        self.table.clear()
        self.original_labels.clear()

    def get_original_label(self, hashed_key):
        return self.original_labels.get(hashed_key, "Bilinmeyen Etiket")
```

Etiketlerin daha güvenli ve çakışmasız saklanması için MD5 hash tabanlı sembol tablosu kullanılmaktadır. Bu sınıf, etiketleri hashleyerek saklar ve istenildiğinde orijinal haliyle erişimi mümkün kılar.

- add(label, address): Etiket ve adres ekler.
- get(label): Etiketin adresini döndürür.

4) Operandların Ayrıştırılması

```python
def parse_operand(operand, is_source=True):
    operand = operand.strip()
    try:
        if operand.startswith("#"):
            parts = operand[1:].split(",", 1)
            value = int(parts[0], 16) if parts[0].startswith("0x") else int(parts[0])
            if len(parts) > 1 and parts[1].strip() in registers:
                return addressing_modes["#N"], value, True, registers[parts[1].strip()]
            return addressing_modes["#N"], value, True, 0
        elif operand.startswith("&"):
            return addressing_modes["&ADDR"], int(operand[1:], 16), True, 0
        elif operand in registers:
            return addressing_modes["Rn"], registers[operand], False, registers[operand]
        elif operand.startswith("@"):
            reg = operand[1:].replace("+", "")
            mode = "@Rn+" if "+" in operand else "@Rn"
            if reg not in registers:
                raise ValueError(f"Geçersiz register: {reg}")
            return addressing_modes[mode], registers[reg], False, registers[reg]
        elif "(" in operand:
            idx, reg = operand.split("(")
            reg = reg[:-1]
            if reg not in registers:
                raise ValueError(f"Geçersiz register: {reg}")
            return addressing_modes["X(Rn)"], registers[reg], int(idx, 16), registers[reg]
        raise ValueError(f"Geçersiz operand: {operand}")
    except Exception as e:
        raise ValueError(f"Operand hatası: {str(e)}")
```

Bu fonksiyon, bir assembler için kritik olan operand (işlenen) çözümleme işlemini yapar; yani assembly dilindeki #10, @R4, X(R5) gibi operandların hangi adresleme moduna ait olduğunu, hangi register'ın kullanıldığını ve ek veri (örneğin sabit bir sayı) gerekip gerekmediğini belirler. Böylece assembler, bu operandların nasıl makine koduna dönüştürüleceğini anlayabilir. Fonksiyon her durumu kontrol ederek operandı parçalar, geçersiz ifadelerde hata verir, doğruysa adresleme modunun sayısal kodunu, değeri ve ilgili register bilgisini döndürür. Bu işlem, doğru makine kodu üretimi için hayati önem taşır çünkü MSP430 işlemcisi, operand türlerine göre farklı kodlama biçimleri kullanır.

5) PASS 1

```python
def pass1(kaynak_kod):
    global location_counter
    satirlar = kaynak_kod.split("\n")
    location_counter = 0x2000
    symbol_table.clear()
    prev_lc = location_counter

    for satir in satirlar:
        satir = satir.split(";")[0].strip()
        if not satir:
            continue

        if satir.startswith("START"):
            location_counter = int(satir.split()[1], 16)
            prev_lc = location_counter
            continue

        parts = satir.split(maxsplit=2)
        if parts[0].endswith(":"):
            etiket = parts[0][:-1]
            if etiket.upper() != "END":
                symbol_table.add(etiket, f"{prev_lc:04x}")

            if len(parts) > 1:
                komut = parts[1]
                operand = parts[2] if len(parts) > 2 else ""
            else:
                continue
        else:
            komut = parts[0]
            operand = parts[1] if len(parts) > 1 else ""

        if komut and komut in opcode_tablosu:
            prev_lc = location_counter
            if komut in ["JNE", "JMP", "JC", "JNC", "CALL"]:
                location_counter += 2
            elif "#" in operand or "&" in operand or "(" in operand:
                location_counter += 4
            else:
                location_counter += 2
        elif komut and komut.upper() not in ["END", "START"] and komut not in opcode_tablosu:
            raise ValueError(f"Geçersiz komut: {komut}")
```

Assembler'ın pass1 aşamasında, assembly kodu satır satır işlenerek etiketlerin adresleri hesaplanır ve hash tabanlı sembol tablosuna kaydedilir. Kod satırları yorumlardan arındırılır, START komutu varsa başlangıç adresi belirlenir. Etiketler prev_lc değeriyle tabloya eklenir. Komut ve operand bilgisi ayrıştırılarak, tanımlı bir komutsa location_counter değeri komutun türüne göre artırılır. Geçersiz komutlarda hata fırlatılır. Bu geçişin temel amacı, pass2 de kullanılmak üzere etiket adreslerinin doğru şekilde belirlenmesidir.

6) Sembollerin Terminalde Gösterilmesi

```
    # Sembolleri terminalde göster
    print("\n--- Sembol Tablosu ---")
    for hashed_key, value in symbol_table.table.items():
        original_label = symbol_table.get_original_label(hashed_key)
        print(f"Etiket: {original_label}, Adres: 0x{value}")
    print("--------------------\n")
```

Sembol tablosunun içeriği terminal ekranına yazdırılmaktadır. Bu işlem, assembler çalıştırıldığında hangi etiketin hangi adrese karşılık geldiğini görsel olarak kontrol edebilmek için eklenmiştir.

Terminal Çıktısı:

```
--- Sembol Tablosu ---
Etiket: MAIN, Adres: 0x2000
Etiket: LOOP, Adres: 0x200c
Etiket: EXIT, Adres: 0x2012
--------------------
```

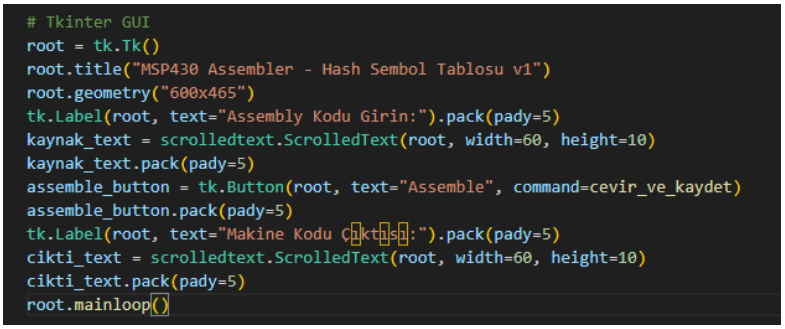
7) PASS 2

```
def pass2(kaynak_kod):
    satirlar = kaynak_kod.split("\n")
    start_adresi = 0x2000
    for satir in satirlar:
        satir = satir.split(";")[0].strip()
        if satir.startswith("START"):
            start_adresi = int(satir.split()[1], 16)
            break

    header = f"H^PROG^{start_adresi:06x}"
    text = f"T^{start_adresi:06x}^"
    text_data = ""
    length = 0
    end = f"E^{start_adresi:06x}"
    current_lc = start_adresi

    for satir in satirlar:
        satir = satir.split(";")[0].strip()
        if not satir or satir.startswith("START") or satir.strip().upper() == "END"
            continue

        if ":" in satir:
            etiket, kalan = satir.split(":", 1)
            kalan = kalan.strip()
            if kalan and etiket.strip().upper() != "END":
                parts = kalan.split(maxsplit=1)
                komut = parts[0]
                operand = parts[1] if len(parts) > 1 else ""
            else:
                continue
        else:
            parts = satir.split(maxsplit=1)
            komut = parts[0]
            operand = parts[1] if len(parts) > 1 else ""

        if komut.upper() == "END":
            break

        if komut not in opcode_tablosu:
            return f"Hata: Geçersiz komut '{komut}'!"

        opcode = opcode_tablosu[komut]
        operand = operand.strip()

        try:
            if komut in ["JNE", "JMP", "JC", "JNC"]:
                label = operand.strip()
                if not label:
                    return f"Hata: '{komut}' için etiket eksik!"
                if not symbol_table.exists(label):
                    return f"Hata: '{label}' etiketi tanımlı değil!"
                target_addr = int(symbol_table.get(label), 16)
                offset = (target_addr - (current_lc + 2)) // 2
                if -512 <= offset <= 511:
                    offset = offset & 0x3FF
                    makine_kod = f"{int(opcode, 16):02x}{offset:02x}"
                    text_data += makine_kod
                    length += 2
                    current_lc += 2
                else:
                    return "Hata: Jump offset sınır dışında!"
            elif komut == "CALL":
                label = operand.strip()
                if not label:
                    return f"Hata: 'CALL' için etiket eksik!"
                if not symbol_table.exists(label):
                    return f"Hata: '{label}' etiketi tanımlı değil!"
                target_addr = int(symbol_table.get(label), 16)
                makine_kod = f"{opcode:04x}"
                text_data += makine_kod
                length += 2
                current_lc += 2
            elif komut in ["PUSH", "RRA"]:
                if not operand:
                    return f"Hata: '{komut}' için operand eksik!"
                mode, reg, extra_word, reg_val = parse_operand(operand)
                makine_kod = f"{opcode}{reg:x}0"[-4:]
                text_data += makine_kod
                length += 2
                current_lc += 2
            else:
                if not operand:
                    return f"Hata: '{komut}' için operand eksik!"
                if "," not in operand:
                    return f"Hata: '{komut}' için iki operand formatı gerekli!"
                src_dst = operand.split(",", 1)
                src = src_dst[0].strip()
                dst = src_dst[1].strip() if len(src_dst) > 1 else ""
                if not dst:
                    return f"Hata: Hedef operand eksik!"
                src_mode, src_val, src_extra, src_reg = parse_operand(src, True)
                dst_mode, dst_val, dst_extra, dst_reg = parse_operand(dst, False)

                if src.startswith("#"):
                    makine_kod = f"{opcode}{dst_reg:x}40"[-4:]
```

```
                    text_data += makine_kod
                    text_data += f"{src_val:04x}"
                    length += 4
                    current_lc += 4
                else:
                    makine_kod = f"{opcode}{dst_reg:x}{src_mode:x}{src_val:x}"[-4:]
                    text_data += makine_kod
                    length += 2
                    current_lc += 2
            except ValueError as e:
                return f"Hata: {str(e)}"
            except Exception as e:
                return f"Hata: Operand işlenirken hata oluştu: {str(e)}"

        calculated_length = len(text_data) // 2
        if length != calculated_length:
            length = calculated_length

        text += f"{length:02x}^{text_data}"
        return f"{header}\n{text}\n{end}"

    def assembler_cevir(kaynak_kod):
        try:
            pass1(kaynak_kod)
            return pass2(kaynak_kod)
        except ValueError as e:
            return f"Hata: {str(e)}"
        except Exception as e:
            return f"Hata: Kod işlenirken hata oluştu: {str(e)}"

    def cevir_ve_kaydet():
        kaynak_kod = kaynak_text.get("1.0", tk.END).strip()
        if not kaynak_kod:
            messagebox.showwarning("Hata", "Lütfen kod giriniz!")
            return

        makine_kod = assembler_cevir(kaynak_kod)
        cikti_text.delete("1.0", tk.END)
        cikti_text.insert(tk.END, makine_kod)

        if not makine_kod.startswith("Hata"):
            try:
                dosya_yolu = os.path.join(os.getcwd(), "assembler_cikti.txt")
                with open(dosya_yolu, "w") as dosya:
                    dosya.write(makine_kod)
                messagebox.showinfo("Başarılı", f"Makine kodu '{dosya_yolu}' dosyasına kaydedildi.")
            except Exception as e:
```

Bu fonksiyon, assembler'ın ikinci geçişini (pass2) gerçekleştirir ve ilk geçişte (pass1) belirlenen etiket adreslerine dayanarak her assembly satırını gerçek makine koduna dönüştürür. Önce START adresini alarak işlem başlangıç adresini belirler, ardından her satırı analiz eder: Eğer komut bir sıçrama (JMP, JNE, vs.) ya da çağrı (CALL) ise hedef etiketin adresini sembol tablosundan alır ve offset veya adres hesaplaması yaparak uygun makine kodunu üretir. Diğer komutlar için operandlar parse_operand fonksiyonu ile çözülür, uygun opcode ile birleştirilir ve makine kodu olarak text_data değişkenine eklenir. Her komutun uzunluğu kadar konum sayacı (current_lc) ilerletilir. Son olarak, makine kodu H (header), T (text) ve E (end) formatında bir çıktı olarak döndürülür. Bu yapı sayesinde assembler, okunabilir assembly kodunu çalıştırılabilir makine koduna başarıyla çevirmiş olur.

8)Arayüz Tasarımı



```
# Tkinter GUI
root = tk.Tk()
root.title("MSP430 Assembler - Hash Sembol Tablosu v1")
root.geometry("600x465")
tk.Label(root, text="Assembly Kodu Girin:").pack(pady=5)
kaynak_text = scrolledtext.ScrolledText(root, width=60, height=10)
kaynak_text.pack(pady=5)
assemble_button = tk.Button(root, text="Assemble", command=cevir_ve_kaydet)
assemble_button.pack(pady=5)
tk.Label(root, text="Makine Kodu Çıktısı:").pack(pady=5)
cikti_text = scrolledtext.ScrolledText(root, width=60, height=10)
cikti_text.pack(pady=5)
root.mainloop()
```

Kullanıcı dostu bir grafiksel arayüz sağlanmıştır. Kullanıcı, assembly komutlarını yazabileceği bir alan, sonucu görebileceği bir çıktı penceresi ve "Assemble" butonuyla tüm işlemi gerçekleştirebilir.

Giriş Assembly Kodu:

```
START 0x2000
MAIN: MOV #0x1234, R5
      ADD R5, R6
      CMP #0x0000, R6
      JNE LOOP
      JMP EXIT
LOOP: SUB #1, R6
      JMP MAIN
EXIT: END
```

Sembol Tablosu Çıktısı:

```
--- Sembol Tablosu ---
Etiket: MAIN, Adres: 0x2000
Etiket: LOOP, Adres: 0x200c
Etiket: EXIT, Adres: 0x2012
--------------------
```

Makine Kodu Çıktısı:

```
H^PROG^002000
T^002000^14^4500141234560650932 06fffe3c003
E^002000
```

Ayrıştırılmış Açıklama:

- **H^PROG^002000**: Header, program adı "PROG" ve başlangıç adresi 0x2000.
- **T^002000^14^...**: Text, adres 0x2000'de başlayan 20 bayt (hex: 14) makine kodu:
  - 4500141234: MOV #0x1234, R5.
  - 5606: ADD R5, R6.
  - 5093: CMP #0x0000, R6.
  - 206fffe: JNE LOOP.
  - 3c003: JMP EXIT.
- **E^002000**: Programın başlangıç adresi.

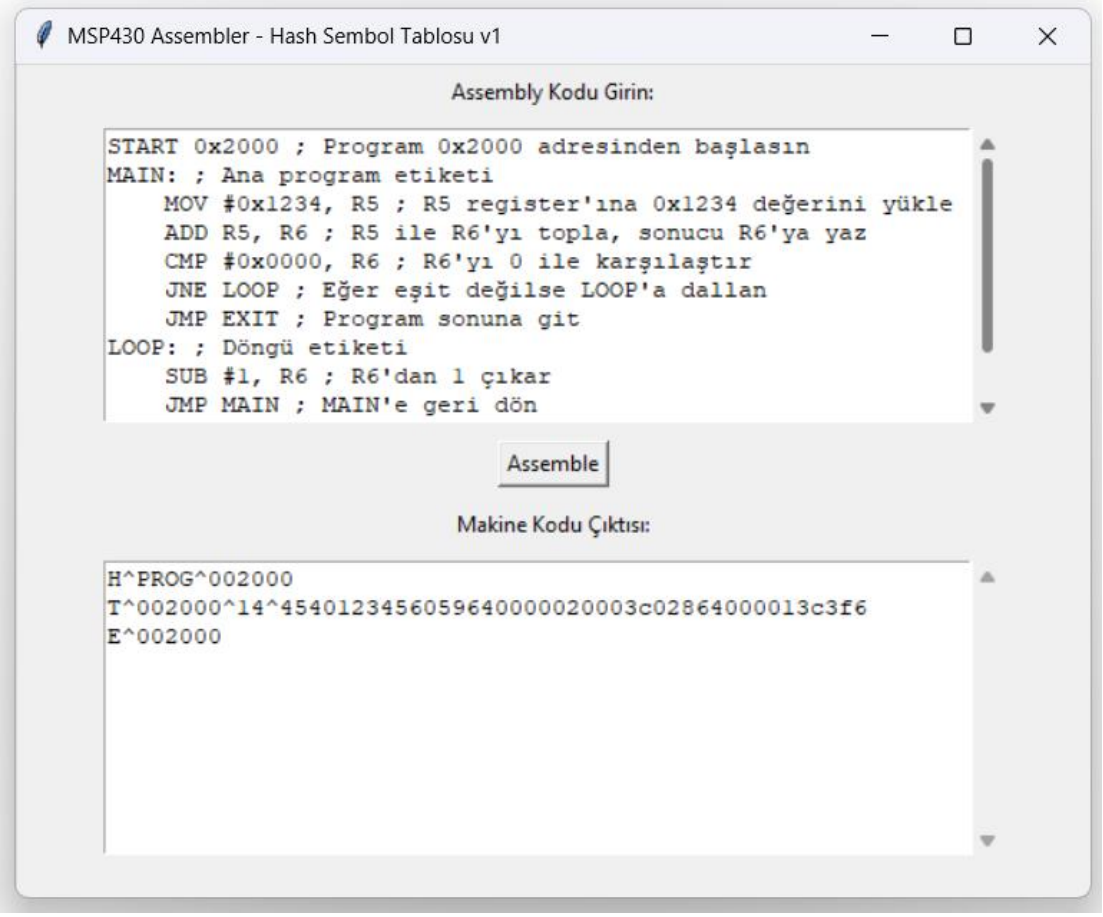
Çıktının Görselleştirilmesi:

Adres	Makine Kodu	Komut
0x2000	4500141234	MOV #0x1234, R5
0x2004	5606	ADD R5, R6
0x2006	5093	CMP #0x0000, R6
0x2008	206fffe	JNE LOOP
0x200a	3c003	JMP EXIT

Gelecekteki İyileştirmeler

- MSP430'un daha fazla komutunun desteklenmesi.
- Hata mesajlarının daha ayrıntılı ve kullanıcı dostu hale getirilmesi.
- GUI'nin görsel tasarımının geliştirilmesi (örneğin, renk temaları).

GUI Ekran Görüntüsü



İSİM	**PUAN**
Adem COŞKUN	25
Emine KAYIT	95
Sedanur PEKER	100
Zeliha POLAT	90

Ek Not: Puanlar %50 ödev çalışması, %25 sunuma girme, % 25 rapor yazma aşamasına katılıma göre değerlendirilmiştir.