Assembler Assignment-1

Varnit Mittal
Vedant Mangrulkar

October 2023

1 Introduction

This project is done as part of the Computer Architecture Course under Prof. Nanditha Rao. This assignment has two parts to it. The first part itself is an MIPS assembly line code. We chose to implement Encryption and Decryption of strings in the MIPS assembly line. For that, we implemented reverse ciphering which means $A \rightarrow Z$ and vice versa. The program opens with formal and welcoming messages and then leads the user to a menu where he/she can choose to either Encrypt their string or Decrypt it.
The second part of the assignment was to implement an assembler which can assemble your MIPS code and generate a proper machine code for each instruction. We use Python Programming language to implement the assembler. We used all the instructions used in our code and made a generalized MIPS assembler for those instructions only. Our assembler uses file IO in order to read through basic instructions, pseudo-instructions and registers. The simulated assembler is a two-pass assembler implementation where in the first pass, we are storing the addresses of the instructions and in the second pass, we are storing and generating the machine code of the instructions used. The code is well commented and modularized for easy reading and ease of use.

2 Code Explanation

The Assembler itself has 3 Python files and 3 text files. The assembler reads from your assembly(.asm) file and generates instruction addresses and machine codes. These both are stored in two types of files an Excel file(.xlsx) and a text file(.txt) for easy reading. The 3 text files contain instructions, pseudo-instructions and register information. The assembler itself is designed in such a way that it is generalized for any instruction used in our assembly code.

2.1 addressStore.py

```
def storeAddress():
    """
    This function reads the assembly code from a file and stores the addresses of instructions and data in a di
    """
    instAddress='0x00400000' #Starting address of the instructions
    control=0
    proc=0  #procedure control variable
    total=0
    procName=""  #procedure name storing variable for temporary purposes
    for line in fp:
        x=line.strip()
        k=0
        found=0
        for i in range(len(x)):  #this iteration is done in order to remove comments in the code
            if(x[i]=='#'):
                k=i
                found=1
                break
            else:
                k=len(x)
        if(k!=0 and found==1):
            x=x[0:k-1]
        elif(found==1 and k==0):
            x=x[0:k]
        if(x==".data"):
            continue
        if(x==".text"):  #this marks the starting of the instructions by changing the control to 1
            control=1
            continue
```

Figure 1: Reading from .asm

This part of the code reads from.asm file and remove all the comments written in the code. This part also controls the assembler reading between dynamically allocated data (.data) and instructions(.text) in your assembly code.

```
if(x != ".data" and x !=".text" and x!='\n' and x!=''):
    if(control==0):  #Here the sizes of the dynamically allocated memory are stored wit
        temp=x.split(': .')
        p=0
        for i in range(len(temp[1])):
            if(temp[1][i]==' '):
                p=i
                break

        op=temp[1][0:p]
        s=""
        a=0
        if(op!="space"):
            p=p+2
            s=temp[1][p:-1]
            a=len(s)
            for i in s:
                if(i=='\\'):
                    a-=1
            a+=1
        else:
            s=temp[1][p+1:]
            a=int(s)
        size[temp[0]]=total
        total+=a
```

Figure 2: Storing sizes

The above part allocates sizes to dynamically allocated data and stores them in a dictionary for generating machines code of la where ori basic instruction needs these sizes.

```
for row in inst:
    temp=row.split()  #parsing through the instructions
    if(temp[0]==t[0]):
        instTrue=True #marking the instruction as basic instruction
        add=int(instAddress,16)
        instruct[t[0]]=hexa(add)  #to generte 8-bit hexadecimal address of the instruction
        add+=4
        instAddress=hexa(add)
        address.append(instruct[t[0]])
        # print(t[0],instruct[t[0]])
        if(temp[0]=='jr' or temp[0]=='lb' or temp[0]=='sb' or temp[0]=='lw' or temp[0]=='syscall' or
            machine[t[0]]=temp[1]
            if(proc==1):
                procedure[procName]=instruct[t[0]] #storing the address of procedure      You, 13 ho
                procName=''
                proc=0
            break

        if(proc==1): #confirming the storage
            procedure[procName]=instruct[t[0]]
            procName=''
            proc=0

        if(temp[1]=='000000'):  #checking for R- Type instruction
            l=[]
            l.append(temp[1])
            l.append(temp[2])
            rinst[t[0]]=l

        elif(temp[-1]=='jump'): #checking for jump instruction
            Jinst[t[0]]=temp[1]
```

Figure 3: Basic Instruction Processing

This part of the code stores the addresses of the basic instructions. We are also processing the addresses of the procedures and storing them in a dictionary.

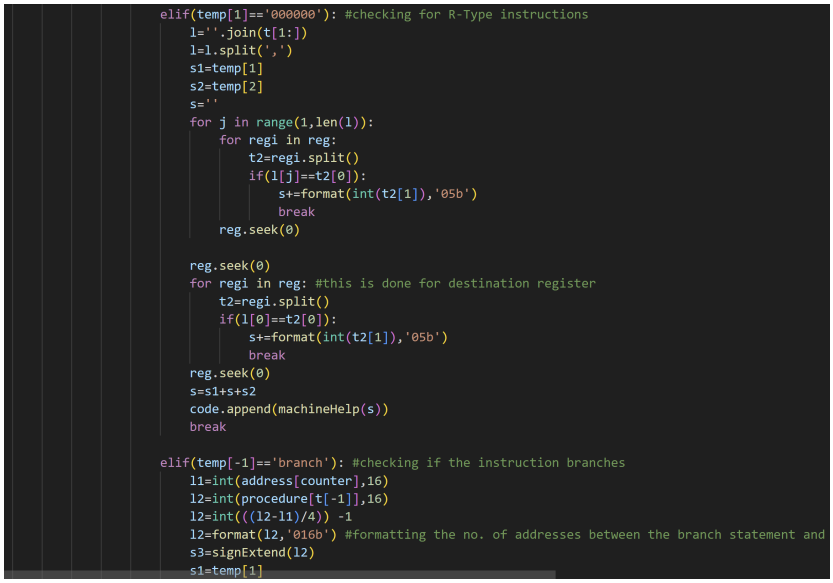
```
inst.seek(0)
if(instTrue==False):  #if the instruction is psuedo then it breaks the instruction into basic and st

    for row in pinst: #parsing through psuedo instructions file
        temp=row.split()
        if(temp[0]==t[0]):
            for i in range(1,len(temp)):
                for a in inst:
                    m=a.split()
                    if(m[0]==temp[i]):
                        add=int(instAddress,16)
                        instruct[t[0]]=hexa(add)
                        add+=4
                        instAddress=hexa(add)
                        address.append(instruct[t[0]])
                        if(proc==1):
                            procedure[procName]=instruct[t[0]]
                            procName=''
                            proc=0
                        if(m[1]=='000000'):
                            l=[]
                            l.append(m[1])
                            l.append(m[2])
                            rinst[t[0]]=l
                        elif(m[-1]=='jump'):
                            Jinst[t[0]]=m[1]
                        elif(m[-1]=='branch'):
                            branch[t[0]]=m[1]
                        else:
                            Iinst[t[0]]=m[1]
                        break
            break
```

Figure 4: Pseudo-Instruction Processing

This part of the code handles the address processing of the pseudo-instructions and breaks them into basic ones.

2.2 machine.py

This module of the assembler reads from the .asm file in the second pass and generates machine code in 8-bit hexadecimal. This module uses addresses stored during the first pass for better and faster implementation. The address generation can be treated as pre-processing.



```
elif(temp[1]=='000000'): #checking for R-Type instructions
    l=''.join(t[1:])
    l=l.split(',')
    s1=temp[1]
    s2=temp[2]
    s=''
    for j in range(1,len(l)):
        for regi in reg:
            t2=regi.split()
            if(l[j]==t2[0]):
                s+=format(int(t2[1]),'05b')
                break
        reg.seek(0)

    reg.seek(0)
    for regi in reg: #this is done for destination register
        t2=regi.split()
        if(l[0]==t2[0]):
            s+=format(int(t2[1]),'05b')
            break
    reg.seek(0)
    s=s1+s+s2
    code.append(machineHelp(s))
    break

elif(temp[-1]=='branch'): #checking if the instruction branches
    l1=int(address[counter],16)
    l2=int(procedure[t[-1]],16)
    l2=int(((l2-l1)/4)) -1
    l2=format(l2,'016b') #formatting the no. of addresses between the branch statement and
    s3=signExtend(l2)
    s1=temp[1]
```

Figure 5: Generating Basic Instruction Machine Code

This part of the code generates the machine code of basic instructions and categorizes them as R-type, I-Type or J-Type. The processing is done accordingly. Sign Extension is also done based on the immediate value field in I-Type instructions. A helper function is created to make an 8-bit hexadecimal machine code.

```
if(m[0]==temp[1]):
    if(m[1]=='000000'): #checking for R-type instruction
        s1=m[1]
        s2=m[-1]
        s=''
        if(t[0]=='bgt'):
            for j in range(1,-1,-1):
                for regi in reg:
                    t2=regi.split()
                    if(l[j]==t2[0]):
                        s+=format(int(t2[1]),'05b')
                        break
                reg.seek(0)
        else:
            for j in range(0,len(l)-1):
                for regi in reg:
                    t2=regi.split()
                    if(l[j]==t2[0]):
                        s+=format(int(t2[1]),'05b')
                        break
                reg.seek(0)

        s=s1+s+'00001'+s2
        code.append(machineHelp(s))
        break

    elif(m[-1]=='branch'): #checking for instruction if it branches
        if(t[0]=='bgt'):
            code.append('0x14200008')
        elif(t[0]=='blt'):
            code.append('0x14200006')
        break
```

Figure 6: Generating Pseudo-Instruction Machine Code

Breaks the instructions into basic ones and generates the machine code for those instructions.

2.3 main.py

```
You, 16 hours ago | 1 author (You) |
import pandas as pd
import addressStore
import machine

address=addressStore.address
code=machine.code

machine.machineCode()

addressStore.storeAddress()

df=pd.DataFrame(columns=['Address','Machine Code'])

df['Address']=address
df['Machine Code']=code

# This is the main calling module which stores the generated machines codes and addresses into

df.to_excel('MachineCodeGenerated.xlsx',index=False)
df.to_csv('MachineText.txt',index=False)
```

Figure 7: Main Code

This is the main code which generates the result in two different files Excel File and text file. Pandas Dataframe is used for formatting the result into a proper table. This module compiles the assembler together.

3 Authors

1. Varnit Mittal (IMT2022025)
2. Vedant Magrulkar (IMT2022519)