**EXPERIMENT NO. 02**

**AIM:** To design and implement second pass of a two pass assembler for IBM 360/370 Processor.

**THEORY:**

An assembler performs the following functions

1. Generate Instructions

a. Evaluate the mnemonic in the operator field to produce its machine code.

b. Evaluate subfields- find value of each symbol, process literals & assign address.

2. Process pseudo-ops.

**PASS 1: DATABASE**

1. Copy of source program input to pass 1.

2. Location Counter (LC).

3. Machine Operation Table (MOT) that indicates for each instruction

a. Symbolic mnemonic

b. Length

c. Binary machine op-code

d. Format (e.g. RR,RX,…..)

4. Pseudo Operation Table (POT) that indicates for each pseudo-opcode the symbolic mnemonic and the action to be taken in pass 2.

5. Symbol Table (ST) prepared by pass 1, containing each label and its corresponding value.

6. Literal Table (LT) prepared by pass 1, containing each literal encountered and its assigned location.

7. Base Table (BT) that indicated which registers are specified as base registers by USING pseudo-opcodes and what the specified contents of these registers are.

8. INSTRUCTION workspace which is used to hold each instruction as its various parts is being assembled together.

9. PRINT LINE workspace used to produce printed listing.

10. PUNCH CARD workspace used to punch the assembled instructions in the format needed by the loader.

11. An output deck of assembled instructions in the format needed by the loader.

**FORMAT OF DATABASES:**

The Machine Operation Table (MOT) and Pseudo Operation Table (POT) are examples of fixed tables. During the assembly process the contents of this table are not filled in or altered.

1. Machine Operation Table (MOT)

6-bytes per entry

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mnemonic  Op-codes  (4 bytes)  Characters | Binary  Op-codes  (1 byte)  hexadecimal | Instruction  Length  (2-bits)  Binary | Instruction  Format  (3-bits)  binary | Not used in this  Design  (3-bits) |
| “Abbb” | 5A | 10 | 001 |  |
| “AHbb” | 4A | 10 | 001 |  |
| “ALbb” | 5E | 10 | 001 |  |
| “ALRb” | 1E | 01 | 000 |  |
| …………… | ……………. | ……………. | …………… |  |

b: blank space

Codes: Instruction Length-01=1 half-words=2 bytes,10=2 half-words=4 bytes, 11=3 half-words=6 bytes

Instruction Format- 000=RR, 001=RX, 010=RS, 011=SI, 100=SS.

2. Pseudo Operation Table (POT)

8-bytes per entry

|  |  |
| --- | --- |
| Pseudo-op  (5-bytes)  Characters | Address of routine to process  Pseudo-op  (3-bytes=24 bits address) |
| “DROPb” | P1DROP |
| “ENDbb” | P1END |
| “EQUbb” | P1EQU |
| “START” | P1START |
| “USING” | P1USING |

Let us consider following source code and find the contents of symbol table and literal table.

|  |  |  |  |
| --- | --- | --- | --- |
| Stmt No. | Symbol | Op-code | Operands |
| 1 | SAMPLE | START | 0 |
| 2 |  | USING | \*,15 |
| 3 |  | A | 1,FOUR |
| 4 |  | A | 2,FIVE |
| 5 | TEMP | EQU | 10 |
| 6 |  | A | 3,=F’3’ |
| 7 |  | USING | TEMP,15 |
| 8 | FOUR | DC | F’4’ |
| 9 | FIVE | DC | F’5’ |
| 10 |  | END |  |

1. Symbol Table (ST)

14-bytes per entry

|  |  |  |  |
| --- | --- | --- | --- |
| Symbol  (8-bytes)  characters | Value  (4-bytes)  hexadecimal | Length  (1-byte)  hexadecimal | Relocation  (1-byte)  character |
| “SAMPLEbbb” | 0000 | 01 | “R” |
| “TEMPbbbb” | 0010 | 04 | “A” |
| “FOURbbbb” | 0012 | 04 | “R” |
| “FIVEbbbb” | 0016 | 04 | “R” |

1. Literal Table (LT)

7-bytes per entry

|  |  |  |  |
| --- | --- | --- | --- |
| Literal  (1-byte) | Value  (4-bytes)  Hexadecimal | Length  (1-byte)  hexadecimal | Relocation  (1-byte)  character |
| F’3’ | 0020 | 04 | “R” |

5. Code after Pass 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Stmt No. | Relative  Address | Statement | |  |
| 1 |  | SAMPLE | START | 0 |
| 2 |  |  | USING | \*,15 |
| 3 | 0 |  | A | 1, \_ (0,15) |
| 4 | 4 |  | A | 2, \_ (0,15) |
| 5 |  | - | | |
| 6 | 8 |  | A | 3, \_ (0,15) |
| 7 |  | - | | |
| 8 | 12 | FOUR | 4 |  |
| 9 | 16 | FIVE | 5 |  |
| 10 |  | - | | |

6. Base Table (BT)

|  |  |  |
| --- | --- | --- |
| Sr. No. | Availability indicator  (1-byte) character | Designated relative-address.  Contents of base register  (3-bytes=24-bit address) hexadecimal |
| 1 | “N” | …….. |
| 2 | “N” | …….. |
| …… | …….. | …….. |
| 14 | “N” | …….. |
| 15 | “Y” | 00 00 00 |

Availability

Y- Register specified in USING pseudo-op

N- Register never specified in USING pseudo-op or subsequently made unavailable by the DROP pseudo-op

**ALGORITHM:**

It generates object code into the appropriate format for later processing by the loader.

1. Location Counter is initialized.

2. Statement is read from the copy file created by pass 1.

3. Opcode is checked to find whether it is pseudo-opcode or MOT to locate the match for source statements. If the entry found in MOT is matched then the MOT entry gives the length of the instruction, binary op-code and the format type of the instruction. The Operand fields of the different instruction format types require somewhat different processing.

a. RR- Format Instruction

b. RX- Register and Indexed Storage Operations

c. RS- Register and Storage Operations

d. SI- Storage and Immediate Operations

e. SS- Storage and Storage Operatons

4. After the instruction has been assembled, it is put into the necessary format for later processing by the loader. Typically, several instructions are placed on a single card.

5. A listing line containing a copy of the source card, its assigned storage location, and its hexadecimal representation is then printed.

6. Finally, the location counter is incremented.

7. Go to step 2.

**Processing of Pseudo-opcodes:**

1. DS and DC

Due to the requirements that the full words must start on a byte whose address is a multiple of four we adjust the location counter. In case of Dc we form constants and insert in the assembled program and increment the LC as per the length of the data field.

2. EQU:

EQU pseudo-op only prints the EQU card as a part of the printed listing.

3. USING:

The operand fields of the pseudo-opcodes are evaluated and then the corresponding base table entry is marked as available.

4. DROP:

The operand field of the pseudo-opcodes are evaluated and then the corresponding base table entry is marked as unavailable.

5. END:

END indicates the end of the source program and terminated the assembly, and the literals are generated for entries in Literal Table (LT).

**CONCLUSION:**

Thus, we have studied and implemented the functionality of second pass of a two pass assembler. The object code for the given program is generated into the appropriate format for later processing by the loader.

**Program:**

import java.util.\*;

import java.io.\*;

class Pass2

{

static int lc=0,index=0;

static String litrl[][]=new String[10][4]; //assumning 10 literals

static int basetable[][]=new int[10][2]; //assuming 10 entries

public static void main(String args[])

{

pass2();

}

static void pass2()

{

int first=0,val=0,i,temp,mask;

String s,s0="",str[],output,mot;

int value[]=new int[2];

index=0;

try

{

BufferedReader p1o = new BufferedReader(new FileReader("pass1output.txt"));

File p2 = new File("pass2output.txt");

if (!p2.exists())

p2.createNewFile();

BufferedWriter op = new BufferedWriter(new FileWriter(p2.getAbsoluteFile()));

for(;(s=p1o.readLine())!=null;s0=str[val],val=0)

{

StringTokenizer st=new StringTokenizer(s);

str=new String[st.countTokens()];

for(i=0;i<str.length;i++)

str[i]=st.nextToken();

if(str.length==3)

val=1;

st=new StringTokenizer(str[val+1],","); //delimiter is comma

String lit[]=new String[st.countTokens()];

for(i=0;i<lit.length;i++)

lit[i]=st.nextToken();

if(str[val].equalsIgnoreCase("USING")) //dealing with pot

{

if(lit[0].equals("\*"))

{

if(first==0)

value[0]=0;

else

value[0]=getValue(s0,0); //0=symbol table & 1=literal table value

value[1]=Integer.parseInt(lit[1]);

first=1;

}

else

{

for(i=0;i<lit.length;i++)

{

value[i]=getValue(lit[i],0);

if(value[i]==-1)

value[i]=Integer.parseInt(lit[i]);

}

}

basetable[index][0]=value[1];

basetable[index++][1]=value[0];

}

else //dealing with mot

{

if(str[val].equalsIgnoreCase("BNE"))

output="BC\t"+7;

else

if(str[val].equalsIgnoreCase("BR"))

output="BCR\t"+15+",";

else

output=str[val]+"\t";

BufferedReader mt = new BufferedReader(new FileReader("mot.txt"));

while((mot=mt.readLine())!=null)

{

st=new StringTokenizer(mot);

String mts[]=new String[st.countTokens()];

for(i=0;i<mts.length;i++)

mts[i]=st.nextToken();

if(str[val].equals(mts[0]))

{

if(mts[3].equals("RR"))

{

for(i=0;i<lit.length;i++)

{

if(i!=0)

output+=",";

temp=getValue(lit[i],0);

if(temp!=-1)

output+=temp;

else

output+=lit[i];

}

}

else

{

if(lit.length==1)

output+=offset(lit[0]);

else

{

temp=getValue(lit[0],0);

if(temp!=-1)

output+=temp;

else

output+=lit[0];

output+=offset(lit[1]);

}

}

break;

}

}

op.write(output);

op.newLine();

}

}

op.close();

}

catch(FileNotFoundException ex)

{

System.out.println("Unable to find file ");

}

catch(IOException e)

{

e.printStackTrace();

}

}

static void ltorg(boolean flag)

{

int i,l=0;

if(flag)

{

l=lc+8;

lc=l-(l%8);

}

for(i=0;i<index;i++)

if(litrl[i][1].equals("-1"))

{

litrl[i][1]=""+lc;

lc+=4;

}

}

static String offset(String s)

{

int value,indx,i,ind=0,offst,new\_offst,indx\_reg=0;

String string=s;

if(s.charAt(0)=='=')

value=getValue(s.substring(1,s.length()),1); //0=symbol table & 1=literal table value

else

{

indx=s.indexOf("(");

if(indx!=-1)

{

s=s.substring(0,indx);

indx\_reg=getValue(string.substring(string.indexOf("(")+1,string.indexOf(")")),0);

}

value=getValue(s,0);

}

offst=Math.abs(value - basetable[ind][1]);

for(i=1 ; i<index ; i++)

{

new\_offst = Math.abs(value - basetable[i][1]);

if(new\_offst < offst)

{

offst = new\_offst;

ind = i;

}

}

String result = ","+offst + "(" + indx\_reg + ", " + basetable[ind][0] + ")";

return result;

}

static int getValue(String s,int flag)

{

try

{

String sym,file\_name;

if(flag==0)

file\_name="symbol\_table.txt";

else

file\_name="literal\_table.txt";

BufferedReader br = new BufferedReader(new FileReader(file\_name));

while((sym=br.readLine())!=null)

{

StringTokenizer st=new StringTokenizer(sym);

String str[]=new String[st.countTokens()];

for(int i=0;i<str.length;i++)

str[i]=st.nextToken();

if(str[0].equalsIgnoreCase(s))

return Integer.parseInt(str[1]);

}

}

catch(FileNotFoundException ex)

{

System.out.println("Unable to find file ");

}

catch(IOException e)

{

e.printStackTrace();

}

return -1;

}

}

**Input:**

**1. Input.txt**

PRGAM2 START 0

USING \*,15

LA 15,SETUP

SR TOTAL,TOTAL

AC EQU 2

INDEX EQU 3

TOTAL EQU 4

DATABASE EQU 13

SETUP EQU \*

USING SETUP,15

L DATABASE,=A(DATA1)

USING DATAAREA,DATABASE

SR INDEX,INDEX

LOOP L AC,DATA1(INDEX)

AR TOTAL,AC

A AC,=F'5'

ST AC,SAVE(INDEX)

A INDEX,=F'4'

C INDEX,=F'8000'

BNE LOOP

LR 1,TOTAL

BR 14

LTORG

SAVE DS 3F

DATAAREA EQU \*

DATA1 DC F'25,26,27'

END

**2. Mot.txt**

Mnemonic Binary Instruction Len Instruction format

(4-bytes) (1-byte) (2-bits) (3-bits)

LA 01h 4 RX

SR 02h 2 RR

L 03h 4 RX

AR 04h 2 RR

A 05h 4 RX

C 06h 4 RX

BNE 07h 4 RX

LR 08h 2 RR

ST 09h 4 RX

BR 15h 2 RR

**OUTPUT:**



**1. literal\_table.txt**

Literal Value Length Relocation

(1-byte) (4-bytes) (1-byte) (1-byte)

A(DATA1) 48 4 R

F'5' 52 4 R

F'4' 56 4 R

F'8000' 60 4 R

**2. pass1output.txt**

USING \*,15

LA 15,SETUP

SR TOTAL,TOTAL

USING SETUP,15

L DATABASE,=A(DATA1)

USING DATAAREA,DATABASE

SR INDEX,INDEX

LOOP L AC,DATA1(INDEX)

AR TOTAL,AC

A AC,=F'5'

ST AC,SAVE(INDEX)

A INDEX,=F'4'

C INDEX,=F'8000'

BNE LOOP

LR 1,TOTAL

BR 14

**3. symbol\_table.txt**

Symbol value Length Relocation

(8-bytes) (4-bytes) (1-byte) (1-byte)

PRGAM2 0 1 R

AC 2 1 A

INDEX 3 1 A

TOTAL 4 1 A

DATABASE 13 1 A

SETUP 6 1 R

LOOP 12 4 R

SAVE 64 4 R

DATAAREA 76 1 R

DATA1 76 4 R

**4. pass2output.txt**

LA 15,6(0, 15)

SR 4,4

L 13,42(0, 15)

SR 3,3

L 2,0(3, 13)

AR 4,2

A 2,24(0, 13)

ST 2,12(3, 13)

A 3,20(0, 13)

C 3,16(0, 13)

BC 7,6(0, 15)

LR 1,4

BCR 15,14

**CONCLUSION:**

Thus, we have studied and implemented the functionality of second pass of a two pass assembler. The object code for the given program is generated into the appropriate format for later processing by the loader.