Lab Assignment – 1

1. Describe advanced assembler directives with an example.

Answer:

Advanced Assembler Directives are a set of commands that are used to provide additional information to the assembler while assembling the source code. These directives help in organizing the code, control the generation of object code, and help in making the code more readable and maintainable. Some of the commonly used advanced assembler directives include:

1. **.equ:** This directive is used to define constants in the assembly language. The syntax is .equ constant_name, value. The constant_name is a label that represents the constant value. Once defined, the constant can be used anywhere in the code as a substitute for the value.

Consider the following **example:**

.equ MAX_COUNT, 100

In this example, the constant MAX_COUNT is defined with a value of 100. This constant can be used anywhere in the code as a substitute for the value 100.

2. .org: This directive is used to specify the starting address of the next instruction. The syntax is .org address. The address specified with this directive is treated as the starting location for the next instruction. This directive is commonly used to specify the starting address of a data section in memory.

Consider the following **example:**

.org 0x2000

In this example, the .org directive is used to specify the starting address of the next instruction as 0x2000. The next instruction will be located at address 0x2000 in memory.

3. .data: This directive is used to define the data section in the assembly language code. The data section contains the constant values and data that are used in the program. The syntax is .data. The data section is usually located after the text section.

Consider the following **example:**

.data

num: .word 10

str: .ascii "Hello World"

In this example, the .data directive is used to define the data section in the assembly language code. The data section contains the constant values and data that are used in the program. The num variable is defined as a word (2 bytes) with a value of 10 and the str variable is defined as an ASCII string with a value of "Hello World".

4. .bss: This directive is used to define the uninitialized data section in the assembly language code. The uninitialized data section contains variables that are not initialized with any values. The syntax is .bss. The uninitialized data section is usually located after the data section.

Consider the following **example:**

.bss

counter: .space 4

In this example, the .bss directive is used to define the uninitialized data section in the assembly language code. The uninitialized data section contains variables that are not initialized with any values. The counter variable is defined as a space of 4 bytes.

5. .section: This directive is used to define a new section in the assembly language code. The syntax is .section section_name. This directive is used to create sections for different purposes such as data section, text section, or bss section.

Consider the following **example:**

.section .text

.globl main

main:

In this example, the .section directive is used to define a new section in the assembly language code. The .text section is created for the purpose of containing the text (instructions) of the program. The .globl main directive is used to declare the main symbol as a global symbol, making it accessible from other object files.

6. .ascii: This directive is used to define a string of characters in the assembly language code. The syntax is .ascii string_of_characters. The string of characters can be used in the program as a constant.

Consider the following **example:**

.ascii "Hello World"

In this example, the .ascii directive is used to define a string of characters in the assembly language code. The string of characters "Hello World" can be used in the program as a constant.

7. .asciz: This directive is similar to the .ascii directive. The only difference is that the .asciz directive automatically adds a null character (\0) at the end of the string of characters.

Consider the following **example:**

.asciz "Hello World"

In this example, the .asciz directive is similar to the .ascii directive. The only difference is that the .asciz directive automatically adds a null character (\0) at the end of the string of characters.

8. .align: This directive is used to align the instructions or data to a specified boundary. The syntax is .align boundary. The boundary is specified in terms of the number of bytes. The assembler inserts padding bytes to align the instructions or data to the specified boundary.

Consider the following **example:**

.align 2

In this example, the .align directive is used to align the instructions or data to a boundary of 2 bytes. The assembler inserts padding bytes to align the instructions or data to the specified boundary.

These are some of the commonly used advanced assembler directives in compiler design. The use of these directives helps in organizing the code, improving its readability, and making it easier to maintain.

2. **Write a C program that reads text from a file and prints on the terminal each input line, *preceded by the line number*. The output will look like -**

    1 This is the first trial line in the file,
    2 and this is the second line.

Try the problem once using fgetc() function and once using fgets() function for reading the input. Why is fread() not suitable for this purpose?

Do not ignore the value returned by the functions fgetc() and fgets().

Answer:

The following code is written using **fgets**() function:

```
#include <stdio.h>

int main()

{

   FILE *fp;

   char line[1000];

   int count = 1;

   fp = fopen("input.txt", "r");

   while (fgets(line, sizeof(line), fp) != NULL)

   {

      printf("%d: %s", count, line);

      count++;

   }

   fclose(fp);
```

```
    return 0;
}
```

The following code is written using **fgetc**() function:

```
#include <stdio.h>
int main()
{
    FILE *fp;
    int c;
    int count = 1;
    int new_line = 1;
    fp = fopen("input.txt", "r");
    while ((c = fgetc(fp)) != EOF)
    {
        if (new_line)
        {
            printf("%d: ", count);
            count++;
            new_line = 0;
        }
        putchar(c);
        if (c == '\n')
            new_line = 1;
    }
    fclose(fp);
    return 0;
```

```
}
```

The **fread**() function:

- The function returns the number of elements successfully read, which may be less than the requested count if the end of the file is reached.
- One of the advantages of using fread() is that it can read data in a single operation, which can be more efficient than reading the data byte-by-byte using functions like fgetc(). This is because fread() can read multiple bytes at once, which can reduce the number of I/O operations required and improve performance. Another advantage of fread() is that it can be used to read binary data as well as text data. When reading binary data, it is important to use the correct size parameter to ensure that the data is read correctly.
- One potential disadvantage of fread() is that it can be more difficult to use for reading text data, since it reads data in binary form and does not automatically handle line breaks or other text formatting. However, it is still possible to use fread() to read text data by reading in blocks of data and then parsing the text manually.

3. **Write a program that takes from the user the name of a file and a "field-number", and then reads that file and for each line in the file prints on the terminal word at position "field- number".**

Answer:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   char filename[50];
   int field_num;
   printf("Enter the name of the file: ");
   scanf("%s", filename);
   printf("Enter the field number: ");
   scanf("%d", &field_num);
   FILE *fp = fopen(filename, "r");
   if (fp == NULL)
   {
      printf("Error opening file\n");
      return 1;
   }
   char line[100];
   while (fgets(line, 100, fp))
   {
      char *token = strtok(line, " ");
      int field_count = 1;
      while (token != NULL)
      {
         if (field_count == field_num)
         {
            printf("%s\n", token);
            break;
         }
         token = strtok(NULL, " ");
         field_count++;
      }
   }
   fclose(fp);
   return 0;
}
```

Lab Assignment – 2

1. **Write a C program to test whether the entered identifier is valid or not.**

Program:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
   char string[25];
   int count = 0, flag;
   printf("Enter any string: ");
   gets(string);
   if ((string[0] >= 'a' && string[0] <= 'z') || (string[0] >= 'A' && string[0] <= 'Z') ||
(string[0] == '_'))
   {
      for (int i = 1; i <= strlen(string); i++)
      {
         if ((string[i] >= 'a' && string[i] <= 'z') || (string[i] >= 'A' && string[i] <= 'Z') ||
(string[i] >= '0' && string[i] <= '9') || (string[i] == '-'))
         {
            count++;
         }
      }
      if (count == strlen(string))
      {
         flag = 0;
      }
   }
   else
   {
      flag = 1;
   }
   if (flag == 1)
      printf("%s is not valid
identifier", string);
   else
      printf("%s is valid
identifier", string);
   return 0;}
```

2. Write a C program to recognize strings under ‘a*’, ‘a*b+’, ‘abb’.

Program:

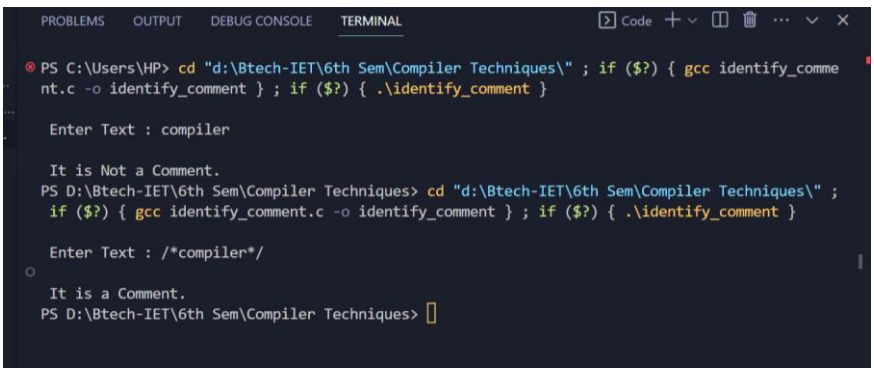
```
#include <stdio.h>

#include <conio.h>

#include <stdlib.h>

// #include

void main()
{
   char s[20], c;
   int state = 0, i = 0;
   // clrscr();
   printf("\n Enter a string:");
   gets(s);
   while (s[i] != '\0')
   {
      switch (state)
      {
      case 0:
         c = s[i++];
         if (c == 'a')
            state = 1;
         else if (c == 'b')
            state = 2;
         else
            state = 6;
         break;
      case 1:
         c = s[i++];
         if (c == 'a')
            state = 3;
         else if (c == 'b')
            state = 4;
         else
            state = 6;
         break;
      case 2:
         c = s[i++];
         if (c == 'a')
            state = 6;
         else if (c == 'b')
```

```
            state = 2;
        else
            state = 6;
        break;
    case 3:
        c = s[i++];
        if (c == 'a')
            state = 3;
        else if (c == 'b')
            state = 2;
        else
            state = 6;
        break;
    case 4:
        c = s[i++];
        if (c == 'a')
            state = 6;
        else if (c == 'b')
            state = 5;
        else
            state = 6;
        break;
    case 5:
        c = s[i++];
        if (c == 'a')
            state = 6;
        else if (c == 'b')
            state = 2;
        else
            state = 6;
        break;
    case 6:
        printf("\n %s is not recognised.", s);
        exit(0);
    }
  }
  if (state == 3)
    printf("\n %s is accepted under rule 'a*'", s);
  else if ((state == 2) || (state == 4))
    printf("\n %s is accepted under rule 'a*b+'", s);
  else if (state == 5)
    printf("\n %s is accepted under rule 'abb'", s);
  getch();
}
```

3. **Write a C program to identify whether a given line is a comment or not.**

Program:

```
#include <stdio.h>
void main()
{
   char com[30];
   int i = 2, a = 0;

   printf("\n Enter Text : ");

   gets(com);
   if (com[0] == '/')
   {
      if (com[1] == '/')

         printf("\n It is a Comment.");

      else if (com[1] == '*')
      {
         for (i = 2; i <= 30; i++)
         {
            if (com[i] == '*' && com[i + 1] == '/')
            {
               printf("\n It is a Comment.");
               a = 1;
               break;
            }
            else
               continue;
         }
         if (a == 0)

            printf("\n It is Not a Comment.");
      }
      else

         printf("\n It is Not a
Comment.");
   }
   else

      printf("\n It is Not a
Comment.");}
```



4. Write a C program to simulate lexical analyzer for validating operators.

Program:

```
#include <stdio.h>

#include <conio.h>
void main()
{
   char s[5];
   printf("\n Enter any operator:");
   gets(s);
   switch (s[0])
   {
   case '>':
      if (s[1] == '=')
         printf("\n Greater than or equal");
      else
         printf("\n Greater than");
      break;
   case '<':
      if (s[1] == '=')
         printf("\n Less than or equal");
      else
         printf("\nLess than");
      break;
   case '=':
      if (s[1] == '=')
         printf("\nEqual to");
      else
         printf("\nAssignment");
      break;
   case '!':
      if (s[1] == '=')
         printf("\nNot Equal");
      else
         printf("\n Bit Not");
      break;
   case '&':
      if (s[1] == '&')
         printf("\nLogical AND");
      else
         printf("\n Bitwise AND");
      break;
   case '|':
```

```
        if (s[1] == '|')
            printf("\nLogical OR");
        else
            printf("\nBitwise OR");
        break;
    case '+':
        printf("\n Addition");
        break;
    case '-':
        printf("\nSubstraction");
        break;
    case '*':
        printf("\nMultiplication");
        break;
    case '/':
        printf("\nDivision");
        break;
    case '%':
        printf("Modulus");
        break;
    default:
        printf("\n Not a operator");
    }
    getch();
}
```