DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SREE NARAYANA GURUKULAM COLLEGE OF ENGINEERING

KADAYIRUPPU, KOLENCHERY

SYSTEM SOFTWARE AND MICROPROCESSOR LAB MANUAL

SUBJECT CODE: CSL331

SEMESTER: 5

YEAR: 3

VISION OF THE DEPARTMENT

To be a center of excellence in the discipline of Computer Science & Engineering to provide self motivated,employable individuals to society.

MISSION OF THE DEPARTMENT

M1:Human resources with Ethical values and leadership qualities

M2:Sound knowledge in Computing

M3:Research capability

M4:Contribute to society

PROGRAM OUTCOMES:

PO1 : Engineering Knowledge: Apply knowledge of mathematics, science, engineering fundamentals and an engineering specialization to the solution of complex engineering problems.

PO2: Problem Analysis: Identify, formulate, research literature and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and engineering sciences.

PO3: Design/ Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.

PO4: Conduct investigations of complex problems using research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.

PO5: Modern Tool Usage: Create, select and apply appropriate techniques, resources and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6: The Engineer and Society: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to professional engineering practice.
PO7: Environment and Sustainability: for various Understand the impact of professional engineering solutions in societal and environmental contexts and demonstrate knowledge of and need for sustainable development.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

PO9: Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams and in multi-disciplinary settings.

PO10: Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.

PO11: Project Management and Finance: Demonstrate knowledge and understanding of engineering and management principles and apply these to one‟s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12: Life-long Learning: Recognize the need for and have the preparation and ability to Engage in independent and life- long learning in the broadest context of technological Change.

PROGRAM-SPECIFIC OBJECTIVES

Students of the Computer Science and Engineering program

PSO1: Shall enhance the employability skills by finding innovative solutions for challenges and problems in various domains of CS.

PSO2: Shall apply the acquired knowledge to develop software solutions and innovative mobile apps(applications) problems.

COURSE OBJECTIVE:

The aim of this course is to give hands-on experience in how microcontrollers, and microprocessors can be programmed. The course also aims to enable students to design and implement system software. The student should get familiar with assembly level programming of microprocessors and microcontrollers, interfacing of devices to microcontrollers, resource allocation algorithms in operating systems and design and implementation of system software.

COURSE OUTCOMES

CSL 331.1	Develop 8086 programs and execute it using a microprocessor kit.
CSL 331.2	Develop 8086 programs and, debug and execute it using MASM assemblers
CSL 331.3	Develop and execute programs to interface stepper motor, 8255, 8279 and digital to analog converters with 8086 trainer kit
CSL 331.4	Implement and execute different scheduling and paging algorithms in OS
CSL 331.5	Design and implement assemblers, Loaders and macroprocessors.

CYCLE 1				
SL. NO	**QUESTION**	**COs**	**LEVEL**	**POs**
1	**.Simulate the following non-preemptive CPU scheduling algorithms to find turn around time and waiting time.** **a)FCFS b)SJF c)Round Robin(preemptive) d)Priority**	CO4	3	PO1,PO2,PO3
2	**Implement Banker's Algorithm for Deadlock Avoidance**	CO4	3	PO1,PO2,PO3
3	**Simulate the following disk scheduling algorithms** **a)FCFS b)SCAN c)C -SCAN**	CO4	3	PO1,PO2,POPO3
CYCLE II(8086 Trainer Kit)				
4	**WAP to a)Add two 8 bit nos** **b)Subtract two 8 bit nos** **c) Multiply two 8 bit no** **d) Division of two 8 bit nos**	CO1	3	PO1,PO2,PO3
5	**Write a program to generate n terms of the Fibonacci series**	CO1	3	PO1,PO2,PO3
6	**Write a program to find the largest and smallest of n numbers**	CO1	3	PO1,PO2,PO3
CYCLE III(MASM Programs)				
7	**1.WAP to find factorial of a number.** **2.Write a program to generate n terms of the Fibonacci series**	CO2	3	PO1,PO2,PO3

	3.WAP to count even and odd numbers from an array of n numbers. **4.WAP to find average of n numbers.**			
8	**WAP to find the smallest of n numbers from an array of n numbers.** **6.WAP to find the length of a string.** **7. WAP to read a character from keyboard and display.** **8. WAP to read a string from keyboard and display its length.**	CO2	3	PO1,PO2,PO3
9	**Interface stepper motor to rotate in Clockwise direction.**	CO3	3	PO1,PO2,PO3
CYCLE IV				
10	Implement Macropreprocessor	CO5	3	PO1,PO2,PO3
11	Implement one pass of an assembler	CO5	3	PO1,PO2,PO3

INTRODUCTION TO OPERATING SYSTEMS

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes state

- New: The process is being created
- Running: Instructions are being executed
- Waiting: The process is waiting for some event to occur
- Ready: The process is waiting to be assigned to a process
- Terminated : The process has finished execution

Apart from the program code, it includes the current activity represented by

- Program Counter,
- Contents of Processor registers,
- Process Stack which contains temporary data like function parameters, return addresses and local variables
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

Scheduling Algorithms

CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. The scheduling criteria include

- CPU utilization:
- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the time of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

1. **FCFS: First Come First Serve Scheduling**

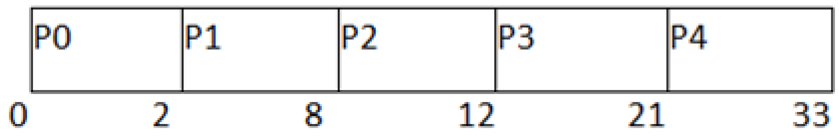
- It is the simplest algorithm to implement.
- The process with the minimal arrival time will get the CPU first.

- The lesser the arrival time, the sooner will the process gets the CPU.
- It is the non-pre-emptive type of scheduling.
- The Turnaround time and the waiting time are calculated by using the following formula.

Turn Around Time = Completion Time - Arrival Time
Waiting Time = Turnaround time - Burst Time

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	8	4
3	3	9	21	18	9
4	4	12	33	29	17

Avg Waiting Time=31/5



2. SJF: Shortest Job First Scheduling

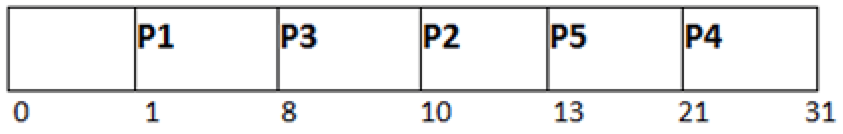
- The job with the shortest burst time will get the CPU first.
- The lesser the burst time, the sooner will the process get the CPU.
- It is the non-pre-emptive type of scheduling.
- However, it is very difficult to predict the burst time needed for a process hence this algorithm is very difficult to implement in the system.
- In the following example, there are five jobs named as P1, P2, P3, P4 and P5. Their arrival time and burst time are given in the table below.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time
1	1	7	8	7	0
2	3	3	13	10	7

3	6	2	10	4	2
4	7	10	31	24	14
5	9	8	21	12	4

Since, No Process arrives at time 0 hence; there will be an empty slot in the **Gantt chart** from time 0 to 1 (the time at which the first process arrives)
.

- According to the algorithm, the OS schedules the process which is having the lowest burst time among the available processes in the ready queue.
- Till now, we have only one process in the ready queue hence the scheduler will schedule this to the processor no matter what is its burst time.
- This will be executed till 8 units of time.
- Till then we have three more processes arrived in the ready queue hence the scheduler will choose the process with the lowest burst time.
- Among the processes given in the table, P3 will be executed next since it is having the lowest burst time among all the available processes.



Avg Waiting Time = 27/5

3. SRTF: Shortest Remaining Time First Scheduling

- It is the pre-emptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution

4. Priority Scheduling

- In this algorithm, the priority will be assigned to each of the processes.
- The higher the priority, the sooner will the process get the CPU.
- If the priority of the two processes is same then they will be scheduled according to their arrival time.

5. Round Robin Scheduling

- In the Round Robin scheduling algorithm, the OS defines a time quantum (slice).

- All the processes will get executed in the cyclic way.
- Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a pre-emptive type of scheduling.

6. Multilevel Queue Scheduling

- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues.
- The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type.
- Each queue has its own scheduling algorithm.

7. Multilevel Feedback Queue Scheduling

- Multilevel feedback queue scheduling, however, allows a process to move between queues.
- The idea is to separate processes with different CPU-burst characteristics.
- If a process uses too much CPU time, it will be moved to a lower-priority queue.
- Similarly, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.
- This form of aging prevents starvation.

Pgm.No.1

CPU SCHEDULING

AIM

Similate the following non pre-emptive CPU scheduling algorithms to find turnaround time and waiting tme.

a). FCFS
b). SJF
c). Priority
d). Round Robin (Pre-emptive)

FCFS (First Come First Serve)

PROGRAM

```
#include<stdio.h>
void main()
{
	int i=0,j=0,b[i],g[20],p[20],w[20],t[20],a[20],n=0,m;
	float avgw=0,avgt=0;
	printf("Enter the number of process : ");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		printf("Process ID : ");
		scanf("%d",&p[i]);

		printf("Burst Time : ");
		scanf("%d",&b[i]);

		printf("Arrival Time: ");
		scanf("%d",&a[i]);
	}

	int temp=0;
	for(i=0;i<n-1;i++)
	{
		for(j=0;j<n-1;j++)
		{
			if(a[j]>a[j+1])
			{
				temp=a[j];
				a[j]=a[j+1];
				a[j+1]=temp;
```

```
                                        temp=b[j];
                                        b[j]=b[j+1];
                                        b[j+1]=temp;

                                        temp=p[j];
                                        p[j]=p[j+1];
                                        p[j+1]=temp;
                                }
                        }
                }

                g[0]=0;
                for(i=0;i<=n;i++)
                        g[i+1]=g[i]+b[i];
                for(i=0;i<n;i++)
                {

                        t[i]=g[i+1]-a[i];
                        w[i]=t[i]-b[i];
                        avgw+=w[i];
                        avgt+=t[i];
                }
                avgw=avgw/n;
                avgt=avgt/n;
                printf("pid\tarrivalT\tBrustT\tCompletionT\tWaitingtime\tTurnaroundTi\n");
                for(i=0;i<n;i++)
                {
                        printf("%d\t%d\t%d\t%d\t\t%d\t\t\t%d\n",p[i],a[i],b[i],g[i+1],w[i],t[i]);
                }
                printf("\nAverage waiting time %f",avgw);
                printf("\nAverage turnarround time %f",avgt);
}
```

OUTPUT 1

```
Enter the number of process : 5
Process ID : 1
Burst Time :  4
Arrival Time: 0
Process ID : 2
Burst Time : 3
Arrival Time: 1
Process ID : 3
Burst Time : 1
Arrival Time: 2
Process ID : 4
Burst Time : 2
Arrival Time: 3
Process ID : 5
Burst Time : 5
```

```
Arrival Time: 4

pid     arrivalT               BrustT        CompletionT  Waitingtime   TurnaroundTi

1       0             4        4             0                    4
2       1             3        7             3                    6
3       2             1        8             5                    6
4       3             2        10            5                    7
5       4             5        15            6                    11

Average waiting time 3.800000
Average turnaround time 6.800000
```

OUTPUT 2

```
Enter the number of process : 3
Process ID : 1
Burst Time : 24
Arrival Time: 0
Process ID : 2
Burst Time : 3
Arrival Time: 0
Process ID : 3
Burst Time : 3
Arrival Time: 0

pid     arrivalT      BrustT CompletionT  Waitingtime   TurnaroundTi
1       0             24     24           0                    24
2       0             3      27           24                   27
3       0             3      30           27                   30

Average waiting time 17.000000
Average turnaround time 27.000000
```

OUTPUT 3

```
Enter the number of process : 3
Process ID : 1
Burst Time : 24
Arrival Time: 0
Process ID : 2
Burst Time : 3
Arrival Time: 2
Process ID : 3
Burst Time : 3
Arrival Time: 3
```

```
pid     arrivalT        BurstT CompletionT  Waitingtime   TurnaroundTi
1       0               24     24           0                       24
2       2               3      27           22                      25
3       3               3      30           24                      27

Average waiting time 15.333333
Average turnaround time 25.333334
```

SJF (Shortest Job First)

PROGRAM

```
#include<stdio.h>
void main()
{
        int i=0,j=0,p[i],b[i],g[20],w[20],t[20],a[20],n=0,m;
        int k=1,min=0,btime=0;
        float avgw=0,avgt=0;
        printf("Enter the number of process : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nProcess id : ");
                scanf("%d",&p[i]);

                printf("Burst Time : ");
                scanf("%d",&b[i]);

                printf("Arrival Time: ");
                scanf("%d",&a[i]);
        }

//sort the jobs based on burst time.
        int temp=0;
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;

                                temp=b[j];
                                b[j]=b[j+1];
                                b[j+1]=temp;

                                temp=p[j];
                                p[j]=p[j+1];
```

```
                    p[j+1]=temp;
                }
            }
        }

        for(i=0;i<n;i++)
        {
            btime=btime+b[i];
            min=b[k];
            for(j=k;j<n;j++)
            {
                if(btime >= a[j] && b[j]<min)
                {
                    temp=a[j];
                    a[j]=a[j-1];
                    a[j-1]=temp;

                    temp=b[j];
                    b[j]=b[j-1];
                    b[j-1]=temp;

                    temp=p[j];
                    p[j]=p[j-1];
                    p[j-1]=temp;
                }
            }
            k++;
        }
        g[0]=a[0];
        for(i=0;i<n;i++)
        {
            g[i+1]=g[i]+b[i];
            if(g[i]<a[i])
                g[i]=a[i];
        }
        for(i=0;i<n;i++)
        {


            t[i]=g[i+1]-a[i];
            w[i]=t[i]-b[i];
            avgw+=w[i];
            avgt+=t[i];
        }
        avgw=avgw/n;
        avgt=avgt/n;
        printf("pid\tBrustTime\tGantChart\tWaiting time\t\tTurnarround Time\n");
        for(i=0;i<n;i++)
        {
            printf(" %d\t %d\t\t%d-%d\t\t%d\t\t\t%d\n",p[i],b[i],g[i],g[i+1],w[i],t[i]);
        }
        printf("\nAverage waiting time %f",avgw);
        printf("\nAverage turnarround time %f\n",avgt);

}
```

OUTPUT 1

```
Enter the number of process : 5

Process id : 1
Burst Time : 7
Arrival Time: 0

Process id : 2
Burst Time : 5
Arrival Time: 1

Process id : 3
Burst Time : 1
Arrival Time: 2

Process id : 4
Burst Time : 2
Arrival Time: 3

Process id : 5
Burst Time : 8
Arrival Time: 4



pid     Brust Time     GantChart     Waiting time         Turnarround Time
 8          7             0-7        0                    7
 3          1             7-8        5                    6
 4          2             8-10       5                    7
 2          5             10-15      9                    14
 5          8             15-23      11                   19

Average waiting time 6.000000
Average turnaround time 10.600000
```

OUTPUT 2

```
Enter the number of process : 4

Process id : 1
Burst Time : 7
Arrival Time: 0
```

```
Process id : 2
Burst Time : 4
Arrival Time: 2

Process id : 3
Burst Time : 1
Arrival Time: 4

Process id : 4
Burst Time : 4
Arrival Time: 5


pid     Burst Time     GantChart     Waiting time          Turnarround Time
 1         7              0-7        0                     7
 3         1              7-8        3                     4
 2         4              8-12       6                     10
 4         4              12-16      7                     11

Average waiting time 4.000000
Average turnaround time 8.000000
```

Priority Scheduling

```
#include<stdio.h>
int main()
{
    int burst_time[20], process[20], waiting_time[20], turnaround_time[20], priority[20];
    int i, j, limit, sum = 0, position, temp;
    float average_wait_time, average_turnaround_time;
    printf("Enter Total Number of Processes:\t");
    scanf("%d", &limit);
    printf("\nEnter Burst Time and Priority For %d Processes\n", limit);
    for(i = 0; i < limit; i++)
    {
        printf("\nProcess[%d]\n", i + 1);
        printf("Process Burst Time:\t");
        scanf("%d", &burst_time[i]);
        printf("Process Priority:\t");
        scanf("%d", &priority[i]);
        process[i] = i + 1;
    }
    for(i = 0; i < limit; i++)
    {
        position = i;
        for(j = i + 1; j < limit; j++)
        {
            if(priority[j] < priority[position])
            {
                position = j;
            }
        }
        temp = priority[i];
        priority[i] = priority[position];
        priority[position] = temp;
        temp = burst_time[i];
        burst_time[i] = burst_time[position];
        burst_time[position] = temp;
        temp = process[i];
        process[i] = process[position];
        process[position] = temp;
    }
    waiting_time[0] = 0;
    for(i = 1; i < limit; i++)
    {
        waiting_time[i] = 0;
        for(j = 0; j < i; j++)
        {
            waiting_time[i] = waiting_time[i] + burst_time[j];
        }
        sum = sum + waiting_time[i];
    }
    average_wait_time = sum / limit;
    sum = 0;
    printf("\nProcess ID\t\tBurst Time\t Waiting Time\t Turnaround Time\n");
    for(i = 0; i < limit; i++)
    {
        turnaround_time[i] = burst_time[i] + waiting_time[i];
        sum = sum + turnaround_time[i];
        printf("\nProcess[%d]\t\t%d\t\t %d\t\t %d\n", process[i], burst_time[i],
waiting_time[i], turnaround_time[i]);
    }
    average_turnaround_time = sum / limit;
    printf("\nAverage Waiting Time:\t%f", average_wait_time);
    printf("\nAverage Turnaround Time:\t%f\n", average_turnaround_time);
    return 0;
}
```

OUTPUT

```
Enter the number of process : 3

Process id : 1
Burst Time : 15
Priority: 3

Process id : 2
Burst Time : 10
Priority: 2
```

```
Process id : 3
Burst Time : 90
Priority: 1


pid      Burst Time      Waiting time              Turnarround Time
 3           90              0                              90
 2           10              90                             100
 1           15              100                            115

Average waiting time 63.000000
Average turnaround time 101.000000
```

Round Robin (pre-emptive)

```
#include<stdio.h>
 int main()
{
      int i, limit, total = 0, x, counter = 0, time_quantum;
      int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
      float average_wait_time, average_turnaround_time;
      printf("\nEnter Total Number of Processes:\t");
      scanf("%d", &limit);
      x = limit;
      for(i = 0; i < limit; i++)
      {
            printf("\nEnter Details of Process[%d]\n", i + 1);
            printf("Arrival Time:\t");
            scanf("%d", &arrival_time[i]);
            printf("Burst Time:\t");
            scanf("%d", &burst_time[i]);
            temp[i] = burst_time[i];
      }
      printf("\nEnter Time Quantum:\t");
      scanf("%d", &time_quantum);
      printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
      for(total = 0, i = 0; x != 0;)
      {
            if(temp[i] <= time_quantum && temp[i] > 0)
            {
                  total = total + temp[i];
                  temp[i] = 0;
                  counter = 1;
            }
            else if(temp[i] > 0)
            {
                  temp[i] = temp[i] - time_quantum;
                  total = total + time_quantum;
            }
            if(temp[i] == 0 && counter == 1)
            {
                  x--;
                  printf("\nProcess[%d]\t\t%d\t\t %d\t\t\t %d", i + 1, burst_time[i], total -
arrival_time[i], total - arrival_time[i] - burst_time[i]);
                  wait_time = wait_time + total - arrival_time[i] - burst_time[i];
                  turnaround_time = turnaround_time + total - arrival_time[i];
                  counter = 0;
            }
            if(i == limit - 1)
            {
                  i = 0;
            }
            else if(arrival_time[i + 1] <= total)
            {
                  i++;
            }
            else
            {
                  i = 0;
            }
      }
      average_wait_time = wait_time * 1.0 / limit;
      average_turnaround_time = turnaround_time * 1.0 / limit;
      printf("\n\nAverage Waiting Time:\t%f", average_wait_time);
      printf("\nAvg Turnaround Time:\t%f\n", average_turnaround_time);
      return 0;
}
```

OUTPUT

```
Enter Total Number of Processes:      4

Enter Details of Process[1]
Arrival Time:  0
Burst Time:     9

Enter Details of Process[2]
Arrival Time:  1
Burst Time:     5

Enter Details of Process[3]
Arrival Time:  2
Burst Time:     3

Enter Details of Process[4]
Arrival Time:  3
```

```
Burst Time:     4

Enter Time Quantum: 5

Process ID              Burst Time      Turnaround Time         Waiting Time

Process[2]              5               9                       4
Process[3]              3               11                      8
Process[4]              4               14                      10
Process[1]              9               21                      12

Average Waiting Time:          8.500000
Avg Turnaround Time:           13.750000
```

Expt No:2

DISK SCHEDULING ALGORITHMS

AIM

Simulate the following disk scheduling algorithms
a). FCFS
b). SCAN
c). C-SCAN

FIRST COME FIRST SERVE (FCFS)

PROGRAM

```
#include<stdio.h>
void main(){

        int ioq[20],i,n,ihead,tot;
        float seek=0,avgs;

        printf("Enter the number of requests\t:");
        scanf("%d",&n);
        printf("Enter the initial head position\t:");
        scanf("%d",&ihead);
        ioq[0] = ihead;
        ioq[n+1] =0;

        printf("Enter the I/O queue requests \n");
        for(i=1;i<=n;i++){
                scanf("%d",&ioq[i]);
        }
        ioq[n+1] =ioq[n];// to set the last seek zero

        printf("\nOrder of request served\n");
        for(i=0;i<=n;i++){

                tot = ioq[i+1] - ioq[i];
                if(tot < 0)
                        tot = tot * -1;
                seek += tot;
               // printf("%d\t%d\n",ioq[i],tot);// to display each seek
                printf("%d --> ",ioq[i]);

        }

        avgs = seek/(n);

        printf("\nTotal Seek time\t\t: %.2f",seek);
        printf("\nAverage seek time\t: %.2f\n\n",avgs);
}
```

OUTPUT 1

```
Enter the number of requests  :5
Enter the initial head position :100
Enter the I/O queue requests
23
89
132
42
187

Order of request served
100 --> 23 --> 89 --> 132 --> 42 --> 187 -->
Total Seek time          : 421.00
Average seek time        : 84.20
```

OUTPUT 2

```
Enter the number of requests  :5
Enter the initial head position :100
Enter the I/O queue requests
23
89
132
42
187

Order of request served
100     77
23      66
89      43
132     90
42      145
187     0

Total Seek time          : 421.00
Average seek time        : 84.20
```

<u>SCAN</u>

PROGRAM

```
#include<stdio.h>
void main()
{
        int ioq[20],i,n,j,ihead,temp,scan,tot;
        float seek=0,avgs;

        printf("Enter the number of requests\t:");
```

```
scanf("%d",&n);
printf("Enter the initial head position\t:");
scanf("%d",&ihead);
ioq[0] = ihead;
ioq[1] = 0;
n += 2;
printf("Enter the I/O queue requests \n");
for(i=2;i<n;i++){
        scanf("%d",&ioq[i]);
}

for(i=0;i<n-1;i++){
        for(j=0;j<n-1;j++)
        {

                if(ioq[j] > ioq[j+1]){

                        temp = ioq[j];
                        ioq[j] = ioq[j+1];
                        ioq[j+1] = temp;

                }

        }
}
ioq[n]=ioq[n-1];
for(i=0;i<n;i++){

        if(ihead == ioq[i]){

                scan = i;
                break;

        }

}

printf("\nOrder of request served\n\n");
tot = 0;
for(i=scan;i>=0;i--){
         //rai  tot = ioq[i+1] - ioq[i];
         tot = ioq[i] – ioq[i-1]; // me
         if(i==0) // me
                tot=ioq[i]-ioq[scan+1];//me
        if(tot < 0)
                tot = tot * -1;
        //seek += tot;
        printf("%d\t%d\n",ioq[i],tot);
}
```

```
        for(i=scan+1;i<n;i++){
                tot = ioq[i+1] - ioq[i];
                if(tot < 0)
                        tot = tot * -1;
                //seek += tot;
                printf("%d\t%d\n",ioq[i],tot);
        }
        seek = ihead + ioq[n-1];

        avgs = seek/(n-2);

        printf("\n\nTotal Seek time\t\t: %.2f",seek);
        printf("\nAverage seek time\t: %.2f\n\n",avgs);

}
```

OUTPUT

```
Enter the number of requests  :8
Enter the initial head position :53
Enter the I/O queue requests
98
183
37
122
14
124
65
67

Order of request served

53      16
37      23
14      14
0       65
65      2
67      31
98      24
122     2
124     59
183     0


Total Seek time         : 236.00
Average seek time       : 29.50
```

<u>CSCAN</u>

PROGRAM

```
#include<stdio.h>
void main()
{
	int ioq[20],i,n,j,ihead,itail,temp,scan,tot=0;
	float seek=0,avgs;

	printf("Enter the number of requests\t: ");
	scanf("%d",&n);
	ioq[0] = 0;
	printf("Enter the initial head position\t: ");
	scanf("%d",&ihead);
	ioq[1] = ihead;
	printf("Enter the maximum track limit\t: ");
	scanf("%d",&itail);
	ioq[2] = itail;
	n += 3;

	printf("Enter the I/O queue requests \n");
	for(i=3;i<n;i++){
		scanf("%d",&ioq[i]);
	}


	for(i=0;i<n-1;i++){
		for(j=0;j<n-1;j++)
		{

			if(ioq[j] > ioq[j+1]){

				temp = ioq[j];
				ioq[j] = ioq[j+1];
				ioq[j+1] = temp;

			}

		}
	}

	for(i=0;i<n+1;i++){

		if(ihead == ioq[i]){

			scan = i;
			break;

		}
```

```
        }

        i = scan;
        temp = n;

        printf("\nOrder of request served\n");
        printf("\n");

        while(i != temp){

                if(i < temp-1){
                        tot = ioq[i+1] - ioq[i];

                        if(tot < 0)
                                tot = tot * -1;
                        seek += tot;
                }
                printf("%d --> ",ioq[i]);
                // printf("%d\t%d\n",ioq[i],tot);
                i++;

                if(i == n){

                        i = 0;
                        temp = scan;
                        seek += itail;

                }

        }

         avgs = seek/(n-3);

        printf("\n\nTotal Seek time\t\t: %.2f",seek);
        printf("\nAverage seek time\t: %.2f\n\n",avgs);
 }
```

OUTPUT

```
Enter the number of requests  : 8
Enter the initial head position : 50
Enter the maximum track limit           : 200
Enter the I/O queue requests
90
120
35
122
38
128
65
```

```
68

Order of request served

50 --> 65 --> 68 --> 90 --> 120 --> 122 --> 128 --> 200 --> 0 --> 35 --> 38 -->

Total Seek time        : 388.00
Average seek time      : 48.50
```

Expt No:3

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

AIM

Implement banker's algorithm for deadlock avoidance

PROGRAM

```
#include<stdio.h>
struct pro{
        int all[10],max[10],need[10];
        int flag;
};
int i,j,pno,r,nr,id,k=0,safe=0,exec,count=0,wait=0,max_err=0;
struct pro p[10];
int aval[10],seq[10];
void safeState()
{
        while(count!=pno){
                safe = 0;
                for(i=0;i<pno;i++){
                        if(p[i].flag){
                                exec = r;
                                for(j=0;j<r;j++)
                                {
                                        if(p[i].need[j]>aval[j]){
                                                exec =0;
                                        }
                                }
                                if(exec == r){
                                        for(j=0;j<r;j++){
                                                aval[j]+=p[i].all[j];
                                        }
                                        p[i].flag = 0;
                                        seq[k++] = i;
                                        safe = 1;
                                        count++;

                                }
                        }
                }
                if(!safe)
                {
                        printf("System is in Unsafe State\n");
                        break;
                }
        }
        if(safe){
                printf("\n\nSystem is in safestate \n");
                printf("Safe State Sequence \n");
                for(i=0;i<k;i++)
                        printf("P[%d]   ",seq[i]);
                printf("\n\n");
        }
}
void reqRes(){
        printf("\nRequest for new Resourses");
        printf("\nProcess id ? ");
        scanf("%d",&id);
        printf("Enter new Request details ");
        for(i=0;i<r;i++){

                scanf("%d",&nr);
                if( nr <= p[id].need[i])
                {
                        if( nr <= aval[i]){
                                aval[i] -= nr;
                                p[id].all[i] += nr;
                                p[id].need[i] -= nr;
                        }
                        else
                                wait = 1;
                }
                else
                        max_err = 1;
        }
        if(!max_err && !wait)
                safeState();
        else if(max_err){
                printf("\nProcess has exceeded its maximum usage \n");
        }
        else{
                printf("\nProcess need to wait\n");
        }


}
void main()
{

        printf("Enter no of process ");
        scanf("%d",&pno);

        printf("Enter no. of resourses ");
        scanf("%d",&r);

        printf("Enter Available Resourse of each type ");
        for(i=0;i<r;i++){
                scanf("%d",&aval[i]);
```

```
}

printf("\n\n---Resourse Details---");
for(i=0;i<pno;i++){

        printf("\nResourses for process %d\n",i);
        printf("\nAllocation Matrix\n");
        for(j=0;j<r;j++){
                scanf("%d",&p[i].all[j]);
        }
        printf("Maximum Resourse Request \n");
        for(j=0;j<r;j++){
                scanf("%d",&p[i].max[j]);
        }
        p[i].flag = 1;

}
// Calcualting need
for(i=0;i<pno;i++){
        for(j=0;j<r;j++){
                p[i].need[j] = p[i].max[j] - p[i].all[j];
        }
}

//Print Current Details
printf("\nProcess Details\n");
printf("Pid\t\tAllocattion\t\tMax\t\tNeed\n");
for(i=0;i<pno;i++)
{
        printf("%d\t\t",i);
        for(j=0;j<r;j++){
                printf("%d  ",p[i].all[j]);
        }
        printf("\t\t");
        for(j=0;j<r;j++){
                printf("%d  ",p[i].max[j]);
        }
        printf("\t\t");
        for(j=0;j<r;j++){
                printf("%d  ",p[i].need[j]);
        }
        printf("\n");

}

//Determine Current State in Safe State
safeState();
int ch=1;
do{
        printf("Request new resourse ?[0/1] :");
```

```
                    scanf("%d",&ch);
                    if(ch)
                              reqRes();
          }while(ch!=0);

          //end:printf("\n");

}
```

OUTPUT

```
Enter no of process 5
Enter no. of resourses 3
Enter Available Resourse of each type 3
3
2


---Resourse Details---
Resourses for process 0

Allocation Matrix
0 1 0
Maximum Resourse Request
7 5 3

Resourses for process 1

Allocation Matrix
3 0 2
Maximum Resourse Request
3 2 2

Resourses for process 2

Allocation Matrix
3 0 2
Maximum Resourse Request
9 0 2

Resourses for process 3

Allocation Matrix
2 1 1
Maximum Resourse Request
2 2 2

Resourses for process 4

Allocation Matrix
```

```
0 0 2
Maximum Resource Request
4 3 3

Process Details
Pid             Allocation          Max                 Need
0               0  1  0             7  5  3             7  4  3
1               3  0  2             3  2  2             0  2  0
2               3  0  2             9  0  2             6  0  0
3               2  1  1             2  2  2             0  1  1
4               0  0  2             4  3  3             4  3  1


System is in safe state

Safe State Sequence

P[1]   P[2]   P[3]   P[4]   P[0]

Request new resource ?[0/1] :
```

Cycle 2

a)

Aim

Addition of two numbers (Immediate, Indirect Addressing)

Immediate data

Program

```
        MOV AX, 3322
        ADD AX, 1111
        MOV [5100], AX
        HLT
```

Output

```
        [5100]          33
        [5101]          44
```

Indirect Addressing

Program to add two numbers in memory locations 5100 and 5101 and store the result in memory location 5102

Program

```
        MOV AL, [5100]
        ADD AL, [5101]
        MOV [5102], AL
        HLT
```

Input

```
        [5100]          0A
        [5101]          00
        [5102]          A0
        [5103]          00
```

Output

```
        [5104]          AA
        [5105]          00
```

Aim

Program to add two 16 bit numbers

Program

```
        MOV AX, [5100]
        ADD AX, [5102]
        MOV [5104], AX
        HLT
```

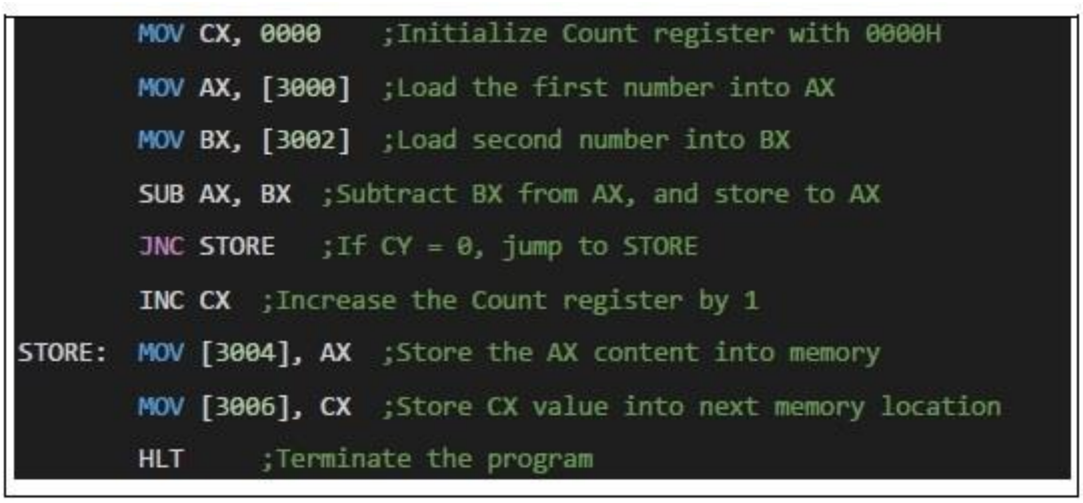
Input

[5100]	0A
[5101]	00
[5102]	A0
[5103]	00

Output

[5104]	AA
[5105]	00

b)Subtract two nos



```
        MOV CX, 0000    ;Initialize Count register with 0000H
        MOV AX, [3000]  ;Load the first number into AX
        MOV BX, [3002]  ;Load second number into BX
        SUB AX, BX  ;Subtract BX from AX, and store to AX
        JNC STORE   ;If CY = 0, jump to STORE
        INC CX  ;Increase the Count register by 1
STORE:  MOV [3004], AX  ;Store the AX content into memory
        MOV [3006], CX  ;Store CX value into next memory location
        HLT     ;Terminate the program
```

c and d)

Aim

Programs to perform multiplication and division

Program to multiply two 8bit numbers

Program

```
        MOV AL, [6000]
        MOV BL, [6001]
        MUL BL
        MOV [6002], AX
        HLT
```

Input

```
        [6000]          CC
        [6001]          0A
```

Output

```
        [6002]          F8
        [6003]          07
```

Program to divide a 16bit number by an 8bit number

Program

```
        MOV AX, [6000]
        MOV BL, [6002]
        DIV BL
        MOV [6004], AX
        HLT
```

Input

```
        [6000]          0C
        [6001]          07
        [6002]          0A
```

Output

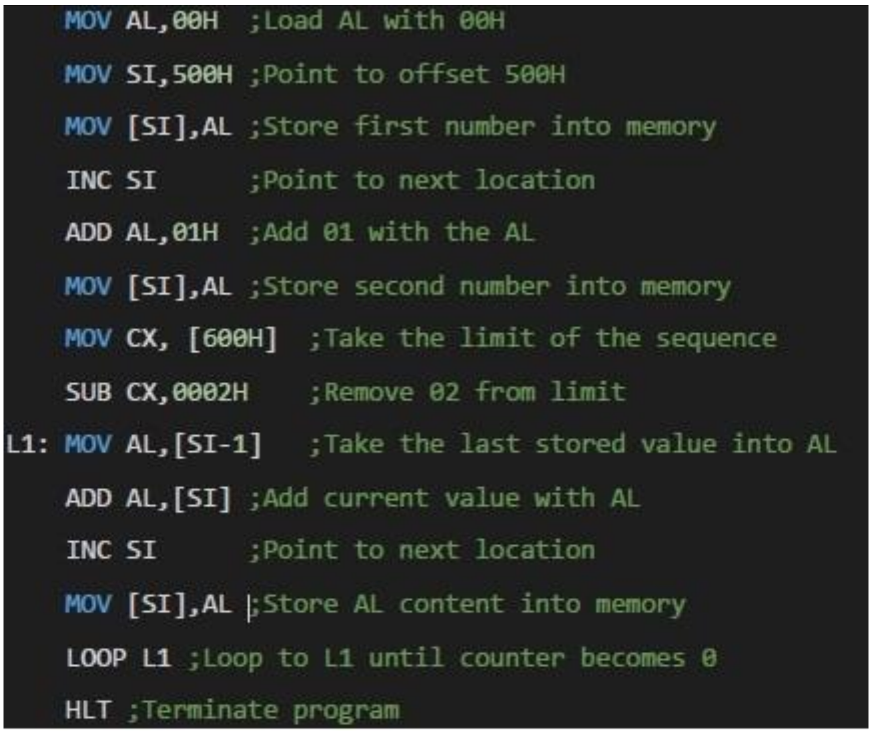
```
        [6004]          B4
        [6005]          04
```

Expt No:5

Aim:

To genrerate fibonocci series

Program



```
    MOV AL,00H  ;Load AL with 00H
    MOV SI,500H ;Point to offset 500H
    MOV [SI],AL ;Store first number into memory
    INC SI      ;Point to next location
    ADD AL,01H  ;Add 01 with the AL
    MOV [SI],AL ;Store second number into memory
    MOV CX, [600H]  ;Take the limit of the sequence
    SUB CX,0002H    ;Remove 02 from limit
L1: MOV AL,[SI-1]   ;Take the last stored value into AL
    ADD AL,[SI] ;Add current value with AL
    INC SI      ;Point to next location
    MOV [SI],AL ;Store AL content into memory
    LOOP L1 ;Loop to L1 until counter becomes 0
    HLT ;Terminate program
```

Expt No:6

Find largest number in an array

AIM: To implement assembly language program for find largest number in an array using 8086 trainer kit

ALGORITHM:

1. Load the array count in a register CL.
2.Get the first two numbers.
3. Compare the numbers and exchange if the number is small.
4.Get the third number from the array and repeat the process until CL is 0.

PROGRAM:

```
1000        MOV SI,9000H        Initialize array size
1002        MOV CL,[SI]         Initialize the count
1004        INC SI              Go to next memory location
1006         MOV AL,[SI]        Move the first data in AL
1007        DEC CL               Reduce the count
1009         INC SI             Move the SI pointer to next data
100A  L2    CMP AL,[SI]         Compare two data’s
100E        JNB L1              If AL > [SI] then go to L1 ( no swap)
1011        MOV AL,[SI]         Else move the large number to AL
1012  L1 :  DEC CL              Decrement the count
1014        JNZ L2              If count is not zero go to L2
1016        MOV DI,9500H        Initialize DI with 9500H
1018        MOV [DI],AL     Else store the biggest number in 9500 location
1010         HLT                 Stop the program
```

RESULT:

Thus the largest number is found in a given array.

Aim

Program to find the sum of squares of first N numbers

Program

```
           MOV CL, [5000]
           XOR CH, CH
           XOR BX, BX
 NEXT :MOV AL,CL
           MUL AL
           ADD BX, AX
           LOOP NEXT
           MOV [5002], BX
           HLT
```

Input

```
        [5000]   05
```

Output

```
        [5002]   37
        [5003]   00
```

Aim

Program to find sum and average of N numbers

Program

```
        MOV CX,[5000]
        XOR AX,AX
        MOV BX,CX
        MOV SI,5001
NEXT :INC SI
        ADD AL,[SI]
        ADC AH,00
        LOOP NEXT
        MOV [5500],AX
        IDIV BX
        MOV [5502],AX
        MOV [5504],DX
        HLT
```

Input

```
        [5000]  :  05
        [5001]  :  00
        [5002]  :  01
        [5003]  :  02
```

```
[5004]  :  03
[5005]  :  04
[5006]  :  05
```

Output

```
[5500]  :  0F
[5501]  :  00
[5502]  :  03
[5503]  :  00
[5504]  :  00
[5505]  :  00
```

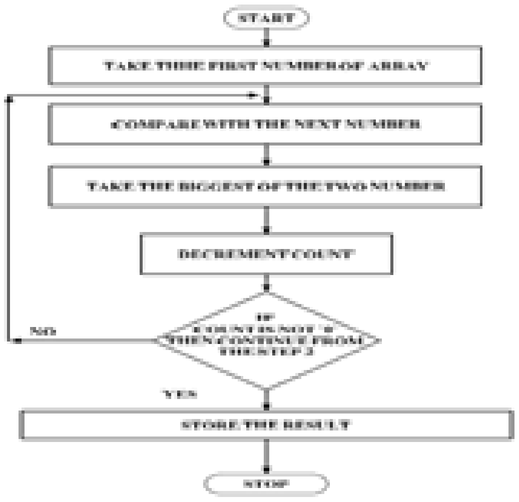
Find largest number in an array

AIM: To implement assembly language program for find largest number in an array using 8086 trainer kit

ALGORITHM:

1. Load the array count in a register C1.
2.Get the first two numbers.
3. Compare the numbers and exchange if the number is small.
4.Get the third number from the array and repeat the process until C1 is 0.

FLOWCHART

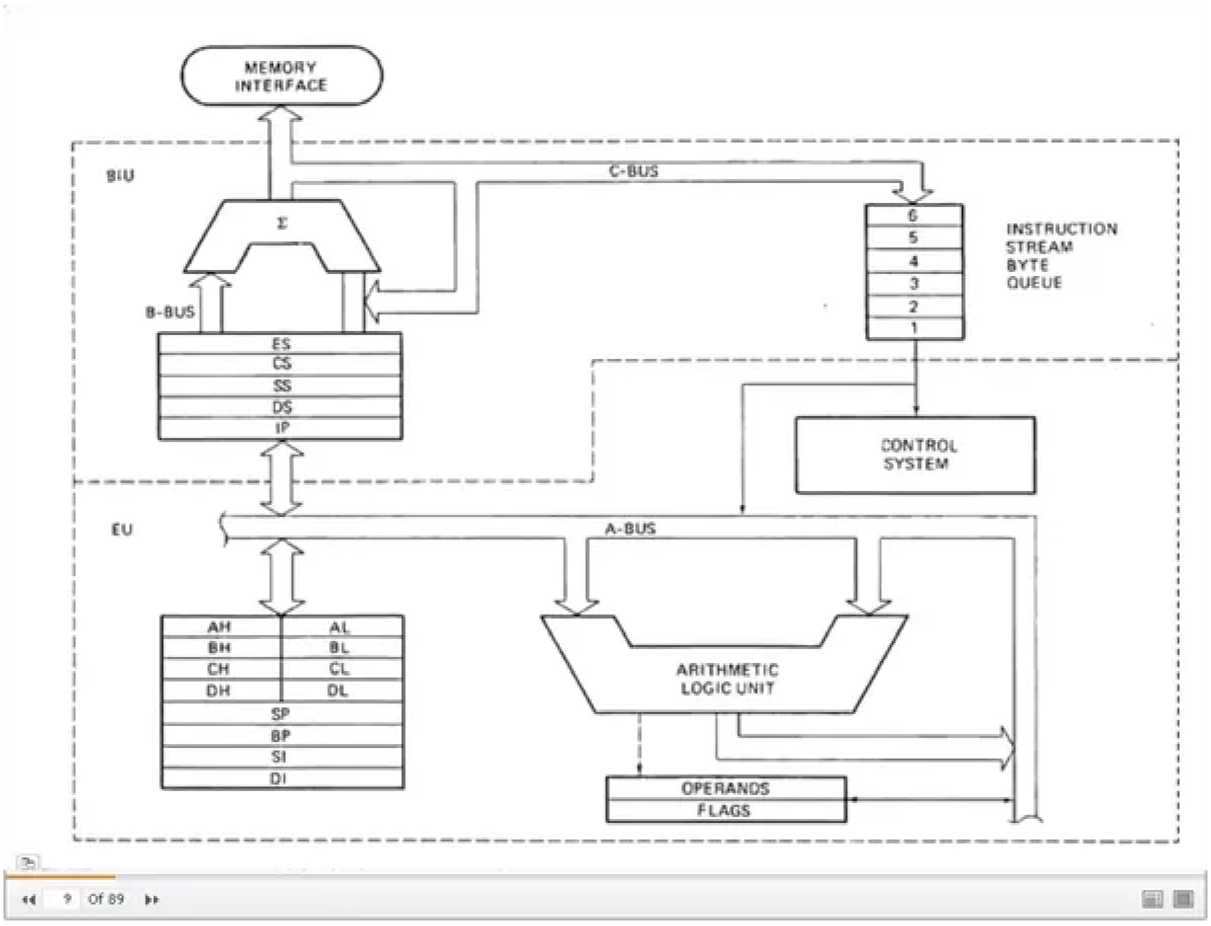

START
TAKE THE FIRST NUMBER OF ARRAY
COMPARE WITH THE NEXT NUMBER
TAKE THE BIGGEST OF THE TWO NUMBER
DECREMENT COUNT
IF COUNT IS NOT 0 THEN CONTINUE FROM THE STEP 2
NO
YES
STORE THE RESULT
STOP

PROGRAM:

```
1000 MOV SI,9000H Initialize array size
1002 MOV CL,[SI] Initialize the count
1004 INC SI Go to next memory location
1006 MOV AL,[SI] Move the first data in AL
1007 DEC CL Reduce the count
1009 INC SI Move the SI pointer to next data
100A L2 CMP AL,[SI] Compare two data’s
100E JNB L1 If AL > [SI] then go to L1 ( no swap)
1011 L1 MOV AL,[SI] Else move the large number to AL
1012 L1 : DEC CL Decrement the count
1014 JNZ L2 If count is not zero go to L2
1016 MOV DI,9500H Initialize DI with 1300H
1018 MOV [DI],AL Else store the biggest number in 1300 location
1010 HLT Stop the program
```

RESULT:
Thus the largest number is found in a given array.

Cycle 3
8086 ARCHITECTURE



INSTRUCTION SET

The 8086 instructions are categorized into the following main types.

1. Data Copy / Transfer Instructions
2. Arithmetic and Logical Instructions
3. Shift and Rotate Instructions
4. Loop Instructions
5. Branch Instructions
6. String Instructions
7. Flag Manipulation Instructions
8. Machine Control Instructions

1 Data Copy / Transfer Instructions:

MOV:

This instruction copies a word or a byte of data from some source to a destination. The destination can be a register or a memory location. The source can be a register, a memory location, or an immediate number.

MOV AX, BX MOV AX, 5000H MOV AX, [SI] MOV AX, [2000H] MOV AX, 50H[BX] MOV [734AH], BX

MOV DS, CX MOV CL, [357AH]

Direct loading of the segment registers with immediate data is not permitted.

PUSH: Push to Stack

This instruction pushes the contents of the specified register/memory location on to the stack. The stack pointer is decremented by 2, after each execution of the instruction.

E.g. PUSH AX
• PUSH DS

• PUSH [5000H]

POP: Pop from Stack

This instruction when executed, loads the specified register/memory location with the

contents of the memory location of which the address is formed using the current stack

segment and stack pointer.

The stack pointer is incremented by 2 Eg. POP AX

POP DS POP [5000H]

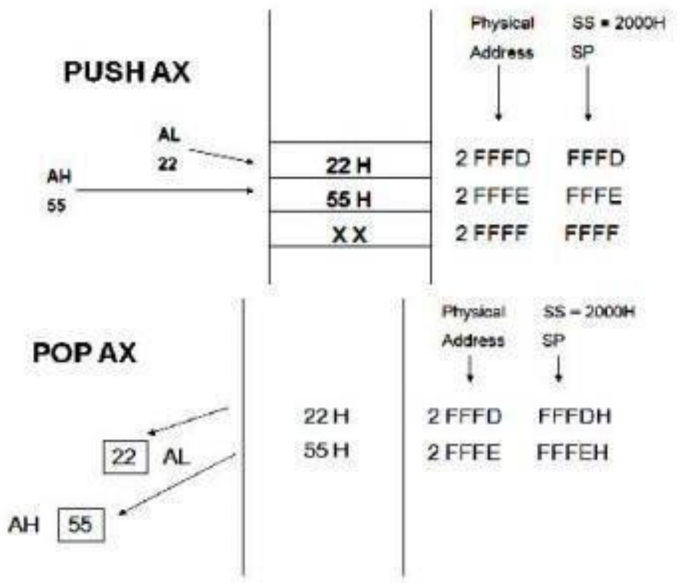


Fig 1.8 Push into and Popping Register Content from Stack Memory

XCHG: Exchange byte or word

This instruction exchange the contents of the specified source and destination operands
Eg. XCHG [5000H], AX

XCHG BX, AX

XLAT:
Translate byte using look-up table

Eg. LEA BX, TABLE1
MOV AL, 04H
XLAT

Input and output port transfer instructions:

IN:
Copy a byte or word from specified port to accumulator.

Eg. IN AL,03H
IN AX,DX

OUT:
Copy a byte or word from accumulator specified port.

Eg. OUT 03H, AL
OUT DX, AX

LEA:

Load effective address of operand in specified register. [reg] offset portion of address in DS

Eg. LEA reg, offset

LDS:

Load DS register and other specified register from memory. [reg] [mem]

[DS] [mem + 2] Eg. LDS reg, mem

LES:

Load ES register and other specified register from memory. [reg] [mem]

[ES] [mem + 2] Eg. LES reg, mem

Flag transfer instructions:

LAHF:

Load (copy to) AH with the low byte the flag register. [AH] [Flags low byte]

Eg. LAHF

SAHF:

Store (copy) AH register to low byte of flag register. [Flags low byte] [AH]

Eg. SAHF

PUSHF:

Copy flag register to top of stack. [SP] [SP] – 2

[[SP]] [Flags] Eg. PUSHF

POPF:

Copy word at top of stack to flag register. [Flags] [[SP]]

[SP] [SP] + 2

2 Arithmetic Instructions:

The 8086 provides many arithmetic operations: addition, subtraction, negation, multiplication and comparing two values.

ADD:

The add instruction adds the contents of the source operand to the destination operand. Eg. ADD AX, 0100H

ADD AX, BX

ADD AX, [SI] ADD AX, [5000H]

ADD [5000H], 0100H ADD 0100H

ADC: Add with Carry

This instruction performs the same operation as ADD instruction, but adds the carry flag to the result.

Eg. ADC 0100H
ADC AX, BX

ADC AX, [SI] ADC AX, [5000] ADC [5000], 0100H

SUB: Subtract

The subtract instruction subtracts the source operand from the destination operand and the result is left in the destination operand.

Eg. SUB AX, 0100H

SUB AX, BX SUB AX, [5000H]
SUB [5000H], 0100H

SBB: Subtract with Borrow

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand
Eg. SBB AX, 0100H

SBB AX, BX SBB AX, [5000H]

SBB [5000H], 0100H

INC: Increment

This instruction increases the contents of the specified Register or memory location by 1. Immediate data cannot be operand of this instruction.

Eg. INC AX INC [BX] INC [5000H]

DEC: Decrement

The decrement instruction subtracts 1 from the contents of the specified register or memory location.

Eg. DEC AX

DEC [5000H]
NEG: Negate

The negate instruction forms 2’s complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.
Eg. NEG AL
AL = 0011 0101 35H Replace number in AL with its 2’s complement

AL = 1100 1011 = CBH
CMP: Compare

This instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory location

Eg. CMP BX, 0100H CMP AX, 0100H CMP [5000H], 0100H CMP BX, [SI]

CMP BX, CX

MUL:Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL. Eg. MUL BH; (AX) (AL) x (BH)

MUL CX; (DX)(AX) (AX) x (CX)
MUL WORD PTR [SI]; (DX)(AX) (AX) x ([SI])

IMUL:Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg. IMUL BH
IMUL CX
IMUL [SI]

CBW: Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Eg. CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX. Result in AX = 1111 1111 1001 1000

CWD: Convert Signed Word to Double Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Eg. CWD

Convert signed word in AX to signed double word in DX: AX DX= 1111 1111 1111 1111

Result in AX = 1111 0000 1100 0001

DIV: Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Eg. DIV CL; Word in AX / byte in CL; Quotient in AL, remainder in AH
DIV CX; Double word in DX and AX / word; in CX, and Quotient in AX; remainder in
DX

AAA: ASCII Adjust After Addition

The AAA instruction is executed after an ADD instruction that adds two ASCII coded operand to give a byte of result in AL. The AAA instruction converts the resulting contents of Al to a unpacked decimal digits.

Eg. ADD CL, DL; [CL] = 32H = ASCII for 2; [DL] = 35H = ASCII for 5; Result [CL] = 67H

MOV AL, CL; Move ASCII result into AL since; AAA adjust only [AL] AAA; [AL]=07, unpacked BCD for 7

AAS: ASCII Adjust AL after Subtraction

This instruction corrects the result in AL register after subtracting two unpacked ASCII operands. The result is in unpacked decimal format. The procedure is similar to AAA instruction except for the subtraction of 06 from AL.

AAM: ASCII Adjust after Multiplication
This instruction, after execution, converts the product available In AL into unpacked BCD format.

Eg. MOV AL, 04; AL = 04 MOV BL ,09; BL = 09

MUL BL; AX = AL*BL; AX=24H AAM; AH = 03, AL=06

AAD: ASCII Adjust before Division

This instruction converts two unpacked BCD digits in AH and AL to the equivalent binary number in AL. This adjustment must be made before dividing the two unpacked BCD digits in AX by an unpacked BCD byte. In the instruction sequence, this instruction appears Before DIV instruction.

Eg. AX 05 08
AAD result in AL 00 3A 58D = 3A H in AL

The result of AAD execution will give the hexadecimal number 3A in AL and 00 in AH where 3A is the hexadecimal Equivalent of 58 (decimal).

DAA: Decimal Adjust Accumulator

This instruction is used to convert the result of the addition of two packed BCD numbers to a valid BCD number. The result has to be only in AL.

Eg. AL = 53 CL = 29

ADD AL, CL; AL (AL) + (CL); AL 53 + 29; AL 7C

DAA; AL 7C + 06 (as C>9); AL 82

DAS: Decimal Adjust after Subtraction

This instruction converts the result of the subtraction of two packed BCD numbers to a valid BCD number. The subtraction has to be in AL only.

Eg. AL = 75, BH = 46

SUB AL, BH; AL 2 F = (AL) - (BH) ; AF = 1

DAS; AL 2 9 (as F>9, F - 6 = 9)

Logical instructions
AND: Logical AND

This instruction bit by bit ANDs the source operand that may be an immediate register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.
Eg. AND AX, 0008H
AND AX, BX

OR: Logical OR

This instruction bit by bit ORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.
Eg. OR AX, 0008H
OR AX, BX

NOT: Logical Invert

This instruction complements the contents of an operand register or a memory location, bit by bit.

Eg. NOT AX
NOT [5000H]

OR: Logical Exclusive OR
This instruction bit by bit XORs the source operand that may be an immediate, register or a memory location to the destination operand that may a register or a memory location. The result is stored in the destination operand.
Eg. XOR AX, 0098H
XOR AX, BX

TEST: Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.
Eg. TEST AX, BX
TEST [0500], 06H

3 Shift and Rotate Instructions
SAL/SHL: SAL / SHL destination, count.

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination to the left and 0 is stored at LSB position. The MSB is shifted into the carry flag. The destination can be a byte or a word. It can be in a register or in a memory location. The number of shifts is indicated by count.

Eg. SAL CX, 1
SAL AX, CL

SHR: SHR destination, count

This instruction shifts each bit in the specified destination to the right and 0 is stored at MSB position. The LSB is shifted into the carry flag. The destination can be a byte or a word.

It can be a register or in a memory location. The number of shifts is indicated by count.
Eg. SHR CX, 1

MOV CL, 05H
SHR AX, CL

SAR: SAR destination, count

This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. The LSB will be shifted into CF.
Eg. SAR BL, 1
MOV CL, 04H

SAR DX, CL

ROL Instruction: ROL destination, count

This instruction rotates all bits in a specified byte or word to the left some number of bit positions. MSB is placed as a new LSB and a new CF.

Eg. ROL CX, 1
MOV CL, 03H
ROL BL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the right some number of bit positions. LSB is placed as a new MSB and a new CF.

Eg. ROR CX, 1
MOV CL, 03H
ROR BL, CL

RCL Instruction: RCL destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is place as new LSB.
Eg. RCL CX, 1
MOV CL, 04H
RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right*along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.
Eg. RCR CX, 1
MOV CL, 04H
RCR AL, CL

ROR Instruction: ROR destination, count

This instruction rotates all bits in a specified byte or word to the *right* some number of bit positions. LSB is placed as a new MSB and a new CF.

Eg. ROR CX, 1
MOV CL, 03H
ROR BL, CL

RCL Instruction: RCL destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the left along with the carry flag. MSB is placed as a new carry and previous carry is place as new LSB.

Eg. RCL CX, 1
MOV CL, 04H
RCL AL, CL

RCR Instruction: RCR destination, count

This instruction rotates all bits in a specified byte or word some number of bit positions to the right*along with the carry flag*. LSB is placed as a new carry and previous carry is place as new MSB.

Eg. RCR CX, 1
MOV CL, 04H
RCR AL, CL

4 Loop Instructions:

Unconditional LOOP Instructions
LOOP: LOOP Unconditionally

This instruction executes the part of the program from the Label or address specified in the instruction upto the LOOP instruction CX number of times. At each iteration, CX is decremented automatically and JUMP IF NOT ZERO structure.
Example: MOV CX, 0004H

Conditional LOOP Instructions
LOOPZ / LOOPE Label
Loop through a sequence of instructions from label while ZF=1 and CX=0.

LOOPNZ / LOOPENE Label
Loop through a sequence of instructions from label while ZF=1 and CX=0.

5 Branch Instructions:

Branch Instructions transfers the flow of execution of the program to a new address specified in the instruction directly or indirectly. When this type of instruction is executed, the CS and IP registers get loaded with new values of CS and IP corresponding to the location to be transferred.

The Branch Instructions are classified into two types

1. Unconditional Branch Instructions.
2. Conditional Branch Instructions.

1.4.5.1 Unconditional Branch Instructions:

In Unconditional control transfer instructions, the execution control is transferred to the specified location independent of any status or condition. The CS and IP are unconditionally modified to the new CS and IP.

CALL: Unconditional Call

This instruction is used to call a Subroutine (Procedure) from a main program. Address of procedure may be specified directly or indirectly. There are two types of procedure depending upon whether it is available in the same segment or in another segment.

i. Near CALL i.e., ±32K displacement.
ii. For CALL i.e., anywhere outside the segment.

On execution this instruction stores the incremented IP & CS onto the stack and loads the CS & IP registers with segment and offset addresses of the procedure to be called.

RET: Return from the Procedure.

At the end of the procedure, the RET instruction must be executed. When it is executed, the previously stored content of IP and CS along with Flags are retrieved into the CS, IP and Flag registers from the stack and execution of the main program continues further.

INT N: Interrupt Type N.

In the interrupt structure of 8086, 256 interrupts are defined corresponding to the types from 00H to FFH. When INT N instruction is executed, the type byte N is multiplied by 4 and the contents of IP and CS of the interrupt service routine will be taken from memory block in 0000 segment.

INTO: Interrupt on Overflow

This instruction is executed, when the overflow flag OF is set. This is equivalent to a Type 4 Interrupt instruction.

JMP: Unconditional Jump

This instruction unconditionally transfers the control of execution to the specified address using an 8-bit or 16-bit displacement. No Flags are affected by this instruction.

IRET: Return from ISR

When it is executed, the values of IP, CS and Flags are retrieved from the stack to continue the execution of the main program.
MOV BX, 7526H
Label 1 MOV AX, CODE
OR BX, AX LOOP Label 1

Conditional Branch Instructions

When this instruction is executed, execution control is transferred to the address specified relatively in the instruction, provided the condition implicit in the Opcode is satisfied. Otherwise execution continues sequentially.
JZ/JE Label
Transfer execution control to address ‘Label’, if ZF=1.
JNZ/JNE Label
Transfer execution control to address ‘Label’, if ZF=0

JS Label
Transfer execution control to address ‘Label’, if SF=1.
JNS Label
Transfer execution control to address ‘Label’, if SF=0.
JO Label
Transfer execution control to address ‘Label’, if OF=1.
JNO Label
Transfer execution control to address ‘Label’, if OF=0.
JNP Label
Transfer execution control to address ‘Label’, if PF=0.
JP Label
Transfer execution control to address ‘Label’, if PF=1.
JB Label
Transfer execution control to address ‘Label’, if CF=1.
JNB Label
Transfer execution control to address ‘Label’, if CF=0.
JCXZ Label
Transfer execution control to address ‘Label’, if CX=0

6 String Manipulation Instructions

A series of data byte or word available in memory at consecutive locations, to be referred as Byte String or Word String. A String of characters may be located in consecutive memory locations, where each character may be represented by its ASCII equivalent. The 8086 supports a set of more powerful instructions for string manipulations for referring to a string, two parameters are required.

I. Starting and End Address of the String.
II. Length of the String.

The length of the string is usually stored as count in the CX register. The incrementing or decrementing of the pointer, in string instructions, depends upon the Direction Flag (DF) Status. If it is a Byte string operation, the index registers are updated by one. On the other hand, if it is a word string operation, the index registers are updated by two.

REP: Repeat Instruction Prefix

This instruction is used as a prefix to other instructions, the instruction to which the REP prefix is provided, is executed repeatedly until the CX register becomes zero (at each iteration CX is automatically decremented by one).
i. REPE / REPZ - repeat operation while equal / zero.

ii. REPNE / REPNZ - repeat operation while not equal / not zero. These are used for CMPS, SCAS instructions only, as instruction prefixes.
MOVSB / MOVSW: Move String Byte or String Word

Suppose a string of bytes stored in a set of consecutive memory locations is to be moved to another set of destination locations. The starting byte of source string is located in the memory location whose address may be computed using SI (Source Index) and DS (Data Segment) contents. The starting address of the destination locations where this string has to be relocated is given by DI (Destination Index) and ES (Extra Segment) contents.

CMPS: Compare String Byte or String Word

The CMPS instruction can be used to compare two strings of byte or words. The length of the string must be stored in the register CX. If both the byte or word strings are equal, zero Flag is set.

The REP instruction Prefix is used to repeat the operation till CX (counter) becomes zero or the condition specified by the REP Prefix is False.

SCAN: Scan String Byte or String Word

This instruction scans a string of bytes or words for an operand byte or word specified in the register AL or AX. The String is pointed to by ES: DI register pair. The length of the string s stored in CX. The DF controls the mode for scanning of the string. Whenever a match to the specified operand is found in the string, execution stops and the zero Flag is set. If no match is found, the zero flag is reset.

LODS: Load String Byte or String Word

The LODS instruction loads the AL / AX register by the content of a string pointed to by DS: SI register pair. The SI is modified automatically depending upon DF, If it is a byte transfer (LODSB), the SI is modified by one and if it is a word transfer (LODSW), the SI is modified by two. No other Flags are affected by this instruction.

STOS: Store String Byte or String Word

The STOS instruction Stores the AL / AX register contents to a location in the string pointer by ES: DI register pair. The DI is modified accordingly, No Flags are affected by this instruction.

The direction Flag controls the String instruction execution, The source index SI and Destination Index DI are modified after each iteration automatically. If DF=1, then the execution follows auto decrement mode, SI and DI are decremented automatically after each iteration. If DF=0, then the execution follows auto increment mode. In this mode, SI and DI are incremented automatically after each iteration.

7 Flag Manipulation and a Processor Control Instruction

These instructions control the functioning of the available hardware inside the processor chip. These instructions are categorized into two types:

1. Flag Manipulation instructions.
2. Machine Control instructions.

Flag Manipulation instructions
The Flag manipulation instructions directly modify some of the Flags of 8086.

i. CLC – Clear Carry Flag.

ii. CMC – Complement Carry Flag.
iii. STC – Set Carry Flag.
iv. CLD – Clear Direction Flag.
v. STD – Set Direction Flag.
vi. CLI – Clear Interrupt Flag.
vii. STI – Set Interrupt Flag.

8 Machine Control instructions

The Machine control instructions control the bus usage and execution

i. WAIT – Wait for Test input pin to go low.
ii. HLT – Halt the process.
iii. NOP – No operation.
iv. ESC – Escape to external device like NDP
v. LOCK – Bus lock instruction prefix.

Assembler directives:

Assembler directives help the assembler to correctly understand the assembly language programs to prepare the codes. Another type of hint which helps the assembler to assign a particular constant with a label or initialize particular memory locations or labels with constants is called an operator. Rather, the operators perform the arithmetic and logical tasks unlike directives that just direct the assembler to correctly interpret the program to code it appropriately. The following directives are commonly used in the assembly language programming practice using Microsoft Macro Assembler
(MASM) or Turbo Assembler (TASM).

↔□**DB: Define Byte**□The DB directive is used to reserve byte or bytes of memory locations□in the available memory. While preparing the EXE file, this directive directs the assembler to allocate the specified number of memory bytes to the said data type that may be a constant, variable, string, etc. Another option of this directive also initializes the reserved memory bytes with the ASCII codes of the characters specified as a string. The following examples show how the DB directive is used for different purposes.

Example:

LIST DB 0lH, 02H, 03H, 04H

This statement directs the assembler to reserve four memory locations for a list named LIST and initialize them with the above specified four values.

Experiment 1

INTRODUCTION TO MASM

The Microsoft macro assembler is an x86 high level assembler for DOS and Microsoft windows. It supports wide varieties of macro facilities and structured programming idioms including high level functions for looping and procedures A program called **assembler** used to convert the mnemonics of instructions along with the data into the equivalent object code modules, these object code may further converted into executable code using linked and loader programs. This type of program is called as ASSEMBLY LANGUAGE PROGRAMMING. The assembler converts and Assembly language source file to machine code the binary equivalent of the assembly language program. In this respect, the assembler reads an ASCII source file from the disk and program as output. The major different between compilers for a high level language like PASCAL and an Assembler is that the compiler usually emits several machine instructions for each PASCAL statement. The assembler generally emits a single machine instruction for each assembler language statement.

Attempting to write a program in machine language is not particularly bright. This process is very tedious, mistakes, and offers almost no advantages over programming in assembly language. The major disadvantages over programming in assembly language over pure machine code are that you must first assemble and link a program before you can execute it. However attempting to assemble the code by hand would take for longer than the small amount of time that the assembler takes the perform conversion for you.

An assembler like Microsoft Macro Assembler (MASM) provides a large number of features for assembly language programmers. Although learning about these features take a fair amount of time. They are so useful that it is well worth the effort. Microsoft MASM version 6.11 contains updated software capable of processing printing instructions. Machine codes and instruction cycle counts are generated by MASM for all instructions on each processor beginning with 8086.

To assemble the file PROG.ASM use this command: (better to use DOS command line)

MASM PROG.ASM

The MASM program will assemble the PROG.ASM file. (To create PROG.OBJ
from PROG.ASM)

To create PROG.EXE from PROG.OBJ, use this LINK command:

LINK PROG.OBJ

It converts the contents of PROG.OBJ into PROG.EXE.
To link more than one object file use + signs between their file names as in:
LINK PROGA+PROGB+PROGC

USING DEBUG TO EXECUTE THE 80x86 PROGRAM:

DEBUG is a utility program that allows a user to load an 80x 86 programs into memory and execute it step by step. DEBUG displays the contents of all processor registers after each instruction execute, allowing the user to determine if the code is performing the desired task. DEBUG only displays the 16-bit portion of the general purpose registers. Code view is capable of displaying the entire 32 bits. DEBUG is a very useful debugging tool. We will use DEBUG to step through a number of simple programs, gaining familiarity with Debug’s commands as we do so. DEBUG contains commands that can display and modify memory, assemble instructions, disassemble code already placed into memory, trace single or multiple instructions, load registers with data and do much more.

DEBUG loads into memory like any other program, in the first available slot.

The memory space used by DEBUG for the user program begins after the end of Debug’s code. If an .EXE or .COM file were specified, DEBUG would load the program according to accepted DOS conventions.

To execute the program file PROG.EXE use this command

DEBUG PROG.EXE

DEBUG uses a minus sign as its command prompt, so should see a “-“ appear on display.

To get a list of some commands available with DEBUG is :

T trace (step by step execution)

U un assemble

D dump

G go (complete execution)

H Hex

To execute the program file PROG.ASM use the following procedure:

```
.MASM PROG.ASM
.LINK PROG.OBJ
.DEBUG PROG.EXE
```

EXECUTION OF ASSEMBLY LANGUAGE PROGRAMMING IN MASM SOFTWARE:

Assembly language programming has 4 steps.

1. Entering Program
2. Compile Program
3. Linking a Program
4. Debugging a Program

Structure of masm program

```
TITLE ...

.MODEL small/tiny/medium/large

.STACK some number

.DATA          ; Begining of data segment.
            ; Initialization of data which is used in program
            ; variable declaration here

.CODE          ; Begining of code segment.
start:         ; Indicates the beginning of instructions.
......
......
```

```
Main PROC   ;Begining of procedure (if neccessary)
……………
…………..
Main ENDP ;End of procedure.

END start        ;End of instruction.

END
```

1. TITLE	Identifies the program listing title. Any text typed to the right side of the directive is printed at the top of each page in the listing file
1. .MODEL	Selects a standard memory model for the programs.
1. .STACK	Sets the size of the program stack which may b any size up to 64kb.
1. .CODE	Identifies the part of the program that contains instructions .
1. PROC	Creates a name and a address for the beginning of a procedure.
1. ENDP	Indicates the end of the procedure.
1. .DATA	All variables pertaining to the program are defined in the area following this directive called data segment.
1. END	Terminates assembly of the program. Any lines of text placed after this directive is ignored.

The following is a list of MASM reserved words:
ASSUME assume definition
CODE begin code segment
DATA begin data segment DB define byte
DD define double word
DQ define quad word
DS define storage
DUP duplicate
DW define word
ELSE else statement
END end program
ENDM end macro
ENDIF end if statement
ENDP end procedure
ENDS end segment
EQU equate
IF if statement
FAR far reference
MACRO define macro
.MODEL model type
NEAR near reference
OFFSET offset
ORQ origin
PARA paragraph
PROC define procedure
.EXIT generate exit code
PUBLIC public reference
SEG locate segment

SEGMENT define segment
PTR pointer

ASSEMBLER DIRECTIVES: The limits are given to the assembler using some pre defined alphabetical strings called Assembler Directives which help assembler to correctly understand. The assembly language programs to prepare the codes.
DB GROUP EXTRN
DW LABEL TYPE
DQ LENGTH EVEN
DT LOCAL SEGMENT
ASSUME NAME
END OFFSET
ENDP ORG
ENDS PROC
EQU PTR
DB-Define Byte: The DB drive is used to reserve byte of memory locations in the available on memory.
DW-Define Word: The DW drive is used to reserve 16 byte of memory location available on memory.
DQ-Define Quad Word (4 words): The DB directives is used to reserve 8 bytes of memory locations in the memory available.
DT-Define Ten Byte: The DT directive is used to reserve 10 byte of memory locations in the available memory.
ASSUME: Assume local segment name the Assume directive is used to inform the assembler. The name of the logical segments to be assumed for different segment used in programs.
END: End of the program the END directive marks the end of an ALP.
ENDP: End of the procedure.
ENDS: End of the segment.
EQU: The directive is used to assign a label with a variable or symbol. The directive is just to reduce recurrence of the numerical values or constants in the program.
OFFSET: Specifies offset address.
SEGMENT: The segment directive marks the starting of the logical segment.

Expt No:7

AIM: To implement assembly language program for factorial of a given number using MASM

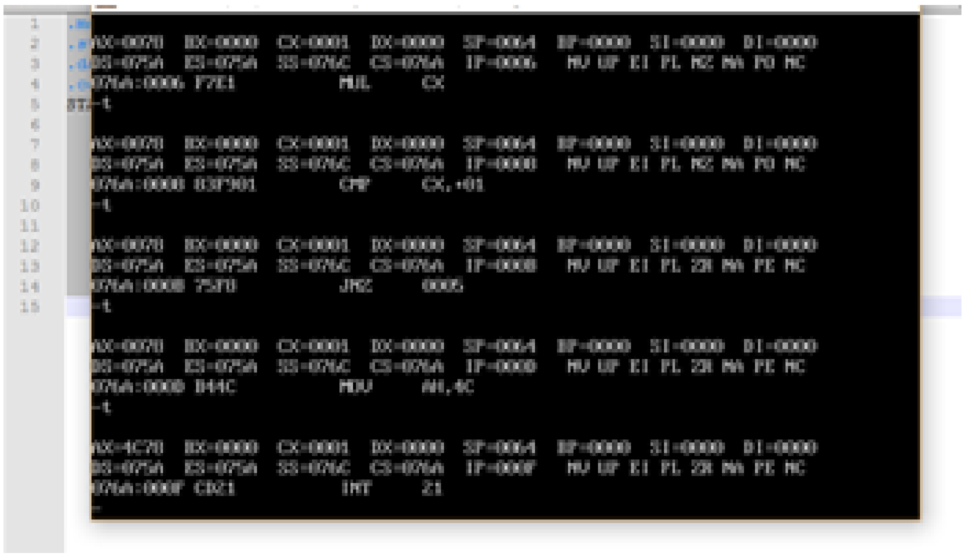
ALGORITHM:

1. Start.
2. Initialize the number in AX
3. Copy the value into CX.
4. Decrement the value of CX
5. Multiply the number with CX
6. Compare with no.1
7. CX not equal to 1 goto step 4
8. Stop.

PROGRAM:

```
ASSUME DS:DATA,CS:CODE
DATA SEGMENT
F DW 00
DATA ENDS
CODE SEGMENT
START:    MOV AL,05
          MOV AH,00
   MOV CX,AX
L1:     DEC CX
   MUL CL
   CMP CX,01
   JNZ L1
   MOV F,AX
   MOV AH,4CH
   INT 21H
CODE ENDS
 END START
```

OUTPUT:



```
AX=0078  BX=0000  CX=0001  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076C  CS=076A  IP=0006   NV UP EI PL NZ NA PO NC
076A:0006 F7E1          MUL     CX
-t

AX=0078  BX=0000  CX=0001  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076C  CS=076A  IP=0008   NV UP EI PL NZ NA PO NC
076A:0008 83F901        CMP     CX,+01
-t

AX=0078  BX=0000  CX=0001  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076C  CS=076A  IP=000B   NV UP EI PL ZR NA PE NC
076A:000B 75F8          JNZ     0005
-t

AX=0078  BX=0000  CX=0001  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076C  CS=076A  IP=000D   NV UP EI PL ZR NA PE NC
076A:000D B44C          MOV     AH,4C
-t

AX=4C78  BX=0000  CX=0001  DX=0000  SP=0064  BP=0000  SI=0000  DI=0000
DS=075A  ES=075A  SS=076C  CS=076A  IP=000F   NV UP EI PL ZR NA PE NC
076A:000F CD21          INT     21
-
```

Fibonocci Series

AIM: To implement assembly language program for Fibonacci series using MASM

ALGORITHM:

```
1. Start.
2. Initialize data segment.
3. Get the number in AL.
4. Add the number with previous data segment value
5. Decrement the counter.
6. If counter is not zero go to step 4
7. Display  Fibonacci series
8. Stop.
```

Aim

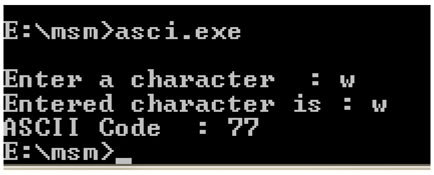
Program to read a character from the keyboard and display its ascii code

Program

```
assume cs:code, ds:data
data segment
    msg1 db 0dh,0ah, "Enter a character  : $"
    msg2 db 0dh,0ah, "Entered character is : $"
    msg3 db 0dh,0ah, "ASCII Code  : $"
    asc1 db ?
    asc2 db ?
data ends
code segment
start:
    mov ax,data
    mov ds,ax

    lea dx,msg1
    mov ah,09h
    int 21h

    mov ah,01h
    int 21h
```

```
        mov bl,al
        and al,0fh
        or al,30h
        mov asc1,al
        mov al,bl
        mov cx,0004h
        shr al,cl
        and al,0fh
        or al,30h
        mov asc2,al

        lea dx,msg2
        mov ah,09h
        int 21h

        mov dl,bl
        mov ah,02h
        int 21h


        lea dx,msg3
        mov ah,09h
        int 21h

        mov dl,asc2
        mov ah,02h
        int 21h

        mov dl,asc1
        mov ah,02h
        int 21h
        mov ah,4ch
        int 21h
    code ends
    end start
```

<u>Output</u>



```
E:\msm>asci.exe

Enter a character  : w
Entered character is : w
ASCII Code  : 77
E:\msm>_
```

EXPERIMENT NO: 12

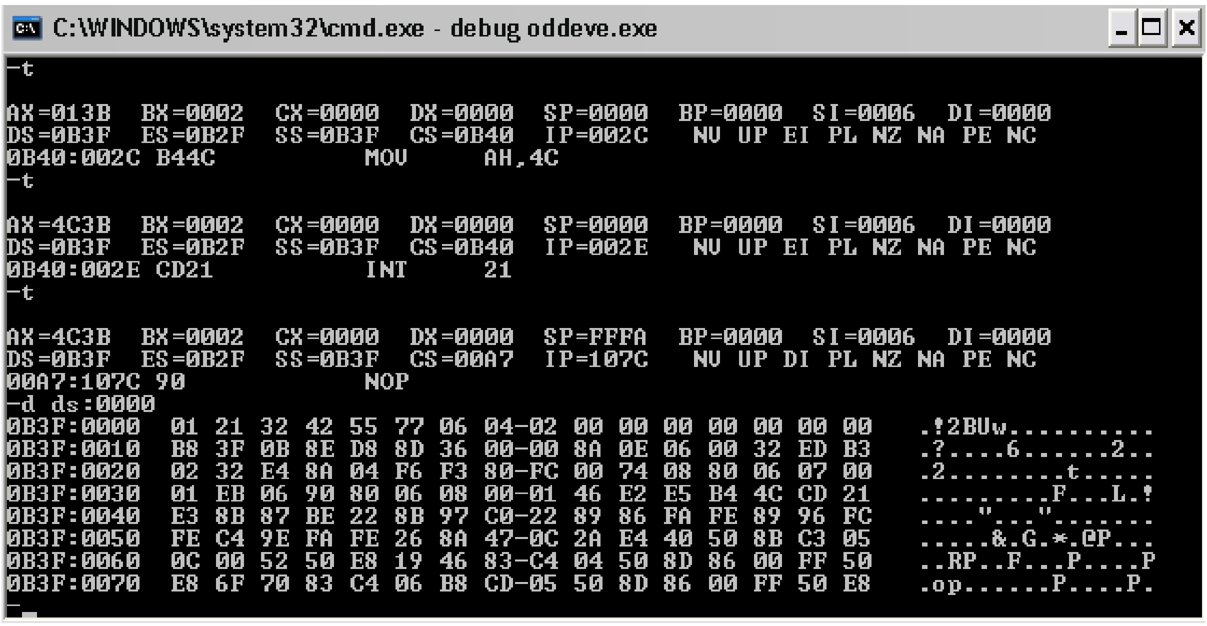
Aim

Program to count the number of Even numbers and odd numbers from a given series of numbers.

Program

```
assume cs:code, ds:data
data segment
num db 01h,21h, 32h, 42h,55h,77h
count db 06h

odd db 0h
eve db 0h
data ends
code segment
start:
            mov ax,data
            mov ds,ax
            lea si,num
            mov cl,count
            xor ch, ch
            mov bl, 02h
next_no:xor ah,ah
            mov al,[si]
            div bl
            cmp ah,00
            je inceven
            add odd,1
            jmp next
inceven:add eve,01
next:   inc si
            loop next_no
            mov ah, 4ch
            int 21h
code ends
end start
```

Output



```
-t

AX=013B  BX=0002  CX=0000  DX=0000  SP=0000  BP=0000  SI=0006  DI=0000
DS=0B3F  ES=0B2F  SS=0B3F  CS=0B40  IP=002C   NV UP EI PL NZ NA PE NC
0B40:002C B44C          MOV     AH,4C
-t

AX=4C3B  BX=0002  CX=0000  DX=0000  SP=0000  BP=0000  SI=0006  DI=0000
DS=0B3F  ES=0B2F  SS=0B3F  CS=0B40  IP=002E   NV UP EI PL NZ NA PE NC
0B40:002E CD21          INT     21
-t

AX=4C3B  BX=0002  CX=0000  DX=0000  SP=FFFA  BP=0000  SI=0006  DI=0000
DS=0B3F  ES=0B2F  SS=0B3F  CS=00A7  IP=107C   NV UP DI PL NZ NA PE NC
00A7:107C 90            NOP
-d ds:0000
0B3F:0000  01 21 32 42 55 77 06 04-02 00 00 00 00 00 00 00   .!2BUw..........
0B3F:0010  B8 3F 0B 8E D8 8D 36 00-00 8A 0E 06 00 32 ED B3   .?....6......2..
0B3F:0020  02 32 E4 8A 04 F6 F3 80-FC 00 74 08 80 06 07 00   .2........t.....
0B3F:0030  01 EB 06 90 80 06 08 00-01 46 E2 E5 B4 4C CD 21   .........F...L.!
0B3F:0040  E3 8B 87 BE 22 8B 97 C0-22 89 86 FA FE 89 96 FC   ...."..."........
0B3F:0050  FE C4 9E FA FE 26 8A 47-0C 2A E4 40 50 8B C3 05   .....&.G.*.@P...
0B3F:0060  0C 00 52 50 E8 19 46 83-C4 04 50 8D 86 00 FF 50   ..RP..F...P....P
0B3F:0070  E8 6F 70 83 C4 06 B8 CD-05 50 8D 86 00 FF 50 E8   .op......P....P.
-
```

Count of even and odd numbers

AIM: To implement assembly language program for count of even and odd numbers using MASM

ALGORITHM:

1. Start.
2. Initialize data segment.
3. Get the number in AL.
4. Divide the given number with 2
5. if the reminder is zero then even count increment otherwise increment odd count
6. If counter is not zero go to step 3
8. Stop.

PROGRAM:

```
 .model small
.stack 64h
.data
A DW 1,2,3,4,5,6,7,8,9,10
.code

START:
    MOV AX,@data
    MOV DS,AX
    LEA SI,A
    MOV DX,0000
    MOV BL,02
    MOV CL,10
  L1:MOV AX,WORD PTR[SI]
    DIV BL
    CMP AH,00
    JNZ L2
    INC DH
    JMP L3
  L2:INC DL
  L3:
    ADD SI,2
    DEC CL
    CMP CL,00
    JNZ L1
    MOV AH,4CH
    INT 21H
END START
end
```

OUTPUT:

```
;OUTPUT:->
;-G CS: 0029
;
;AX=0005  BX=0002  CX=0000  DX=0505  SP=0000  BP=0000  SI=0014  DI=0000
;DS=0BF4  ES=0BE4  SS=0BF4  CS=0BF6  IP=0029   NV UP EI PL ZR NA PE NC
;0BF6:0029 B44C          MOV     AH,4C
```

Experiment-5

Addition of N numbers

AIM: To implement assembly language program for finding average of N number using MASM

ALGORITHM:

1. Start.
2. Initialize data segment.
3. Get the number in AL.
4. Add given number with sum
5. If counter is not zero go to step 3
6. Stop.

PROGRAM:

```
.model small
.stack 64h
.data
A DB 1,2,3,4,5,6,7,8,9,10
.code
START:
    MOV AX,DATA
    MOV DS,AX
    LEA BX,A
    MOV CL,10
    MOV AX,0000
  L1:ADD AL,BYTE PTR[BX]
    INC BX
    DEC CL
    CMP CL,00
    JNZ L1
    MOV SUM,AL
    MOV BH,10
    DIV BH
    MOV AH,4CH
    INT 21H
CODE ENDS
END START
```

OUTPUT:

```
;-G CS: 001E
;AX=0505  BX=0A0A  CX=0000  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
;DS=0BA8  ES=0B98  SS=0BA8  CS=0BA9  IP=001E   NV UP EI PL ZR NA PE NC
;0BA9:001E B44C          MOV     AH,
```

Experiment-7

Read and display a character

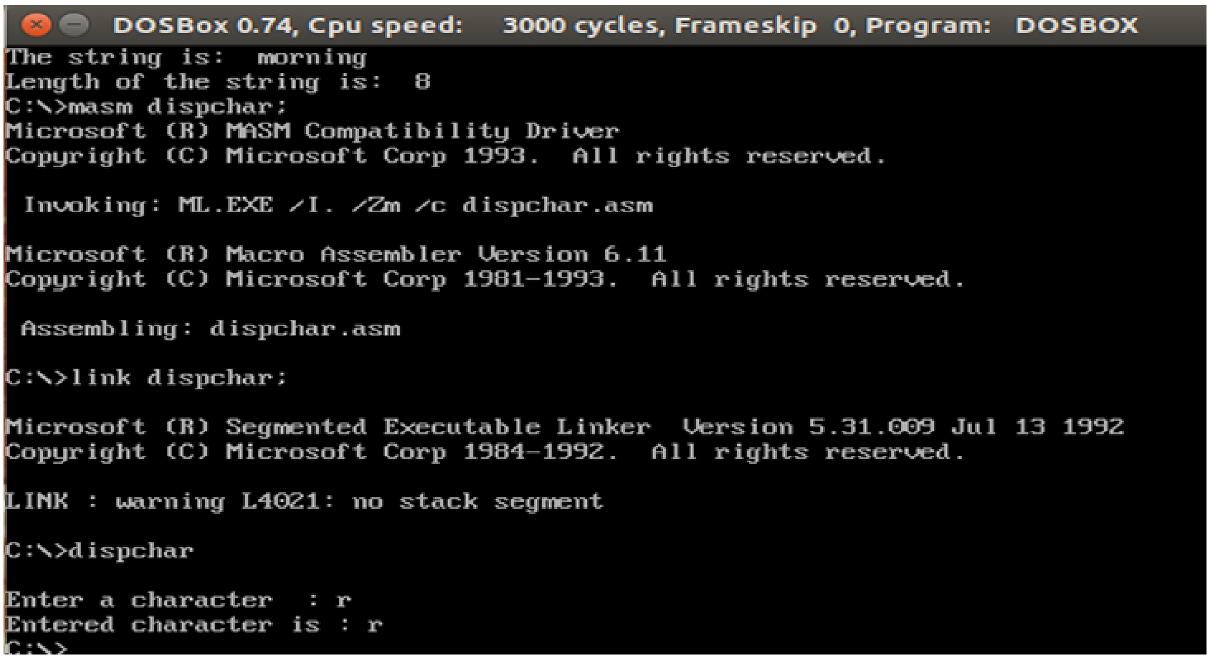
AIM: To implement assembly language program for read and display a character using MASM

ALGORITHM:

1. Start.
2. Initialize data segment.
3. Get the character in AL.
4. Move the character to DL
5. Display the character
6. Stop.

```
assume cs:code, ds:data
data segment
   msg1 db 0dh,0ah, "Enter a character  : $"
   msg2 db 0dh,0ah, "Entered character is : $"

data ends
code segment
start:
   mov ax,data
   mov ds,ax

   lea dx,msg1
   mov ah,09h
   int 21h

   mov ah,01h
   int 21h

   mov bl,al


   lea dx,msg2
   mov ah,09h
   int 21h

   mov dl,bl
   mov ah,02h
   int 21h


   mov ah,4ch
   int 21h
        code ends
end start
```

OUTPUT


DOSBox 0.74, Cpu speed: 3000 cycles, Frameskip 0, Program: DOSBOX

```
The string is:  morning
Length of the string is:  8
C:\>masm dispchar;
Microsoft (R) MASM Compatibility Driver
Copyright (C) Microsoft Corp 1993.  All rights reserved.

 Invoking: ML.EXE /I. /Zm /c dispchar.asm

Microsoft (R) Macro Assembler Version 6.11
Copyright (C) Microsoft Corp 1981-1993.  All rights reserved.

 Assembling: dispchar.asm

C:\>link dispchar;

Microsoft (R) Segmented Executable Linker  Version 5.31.009 Jul 13 1992
Copyright (C) Microsoft Corp 1984-1992.  All rights reserved.

LINK : warning L4021: no stack segment

C:\>dispchar

Enter a character  : r
Entered character is : r
C:\>
```

Experiment-8

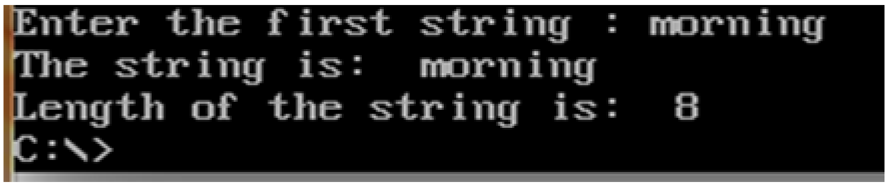
Display string and its length

AIM: To implement assembly language program for display string and its length using MASM

PROGRAM:

```
Assume CS:code, DS:data
data segment
 msg1 db "Enter the first string : $"
 msg2 db "The string is:  $"
 msg3 db "Length of the string is:  $"
 str1 db 20 dup(?)

 data ends

code segment
start:
     mov ax,data
      mov DS,ax
    mov cx,16h
    lea dx,msg1
    mov ah,09h
    int 21h

    lea dx,str1
    mov bx,00
    mov ah,3fh
    int 21h

    lea si,str1
        xor ah,ah
    add si,ax
    mov byte ptr[si],'$'


    lea dx,msg2
    mov ah,09h
    int 21h

    lea dx,str1
    mov ah,09h
    int 21h

    lea dx,msg3
    mov ah,09h
    int 21h

    dec al
    dec al
    mov dl,al
    add dl,30h
    mov ah,02h
    int 21h

    mov ah,4ch
     int 21h

code ends
end start
```

OUTPUT



```
Enter the first string : morning
The string is:  morning
Length of the string is:  8
C:\>
```

INTERFACING
WITH 8086

Expt No: 9

Steppor Motor Rotation

Aim:

To run a steppor motor with an angle 360 degree

THEORY:

A motor in which the rotor is able to assume only discrete stationary angular position is a stepper motor. The rotary motion occurs in a stepwise manner from one equilibrium position to the next. Two- phase scheme: Any two adjacent stator windings are energized. There are two magnetic fields active in quadrature and none of the rotor pole faces can be in direct alignment with the stator poles. A partial but symmetric alignment of the rotor poles is of course possible.

The stepper motor windings A1,B1,A2,B2 can be cyclically excited with a DC current to run the motor in a clockwise direction.By reversing the phase sequence A1,B2,A2,B1, we can obtain anticlockwise stepping.

Anticlockwise

Step	A1	A2	B1	B2
1	1	0	0	0
2	0	0	0	1
3	0	1	0	0
4	0	0	1	0

Clockwise

Step	A1	A2	B1	B2
1	1	0	0	1
2	0	1	0	1
3	0	1	1	0
4	1	0	1	0

ALGORITHM:

1. Get the first data from the lookup table.

2. Initialize the counter and move data into accumulator.

3. Drive the stepper motor circuitry and introduce delay

4. Decrement the counter is not zero repeat

from step(iii)

5.Repeat the above procedure for 256 steps.

PROGRAM:

Memory Location **Instructions**

```
5100   MOV CX,00FF

5104   MOV AL,09

5107   OUT C0,AL

5109   MOV DX,1010

510D   DEC DX

510E   JNZ 510D

5110   MOV AL,05

5113   OUT C0,AL

5115   MOV DX,1010

5119   DEC DX

511A   JNZ 5119

511C   MOV AL,06

511F   OUT C0,AL
```

```
5121   MOV DX,1010

5125   DEC DX

5126   JNZ 5125

5128   MOV AL,0A

512B   OUT C0,AL

512D   MOV DX,1010

5131   DEC DX

5132   JNZ 5131

5134   DEC CX

5135   JNZ 5104
```

OUTPUT:

The stepper motor runs successfully.

```
110A   HLT
```

OUTPUT:

The program runs successfully.

CYCLE 4

Expt No:10

Write a Program to implement Macropreprocessor

Aim:

To implement Macropreprocessor (#define)

Program

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
#include<ctype.h>

char c,temp[50],v[50][50],n[50][50],r[50],s[50];
int l=0;

void main()
{
FILE * fp;
int j,i,k,m;
clrscr();
fp=fopen("f.c","r");
while((c=getc(fp))!=EOF)
{
i=0;
while(isalpha(c)||isdigit(c)||c=='#')
{temp[i++]=c;
c=fgetc(fp);
}
temp[i]='\0';
if(strcmp(temp,"#define")==0)
{c=fgetc(fp);
j=0;
while((c!=' ')&&(c!=EOF))
{s[j++]=c;
c=fgetc(fp);
}
s[j]='\0';
c=fgetc(fp);
k=0;
while(c!='\n'&&c!=' '&&c!=';'&&c!=EOF)
{r[k++]=c;
c=fgetc(fp);
}
r[k]='\0';
strcpy(n[l],s);
strcpy(v[l],r);
l++;
}
else
{
for(i=0;i<l;i++)
{if(strcmp(temp,n[i])==0)
{printf("%s",v[i]);
strcpy(temp,"");
break;
}
}
if(strcmp(temp,"")!=0)
printf("%s",temp);}
printf("%c",c);
}
fclose(fp);
getch();
}
```

Input:

```
F.c

#define PI 3.14
void main()
{
int area,radius=5;
area=PI*radius*radius;
}
```

Output:

```
void main()
{
int area,radius=5;
area=3.14*radius*radius;
}
```

****

Expt No:11

PASS ONE OF TWO PASS ASSEMBLER

AIM

Write a C program to implement pass one of two pass assembler

PROGRAM

```
#include<stdio.h>
#include<string.h>
void main()
{
FILE *f1,*f2,*f3,*f4;
char s[100],lab[30],opcode[30],opa[30],opcode1[30],opa1[30];
int locctr,x=0;
f1=fopen("input.txt","r");
f2=fopen("opcode.txt","r");
f3=fopen("out1.txt","w");
f4=fopen("sym1.txt","w");
while(fscanf(f1,"%s%s%s",lab,opcode,opa)!=EOF)
{
        if(strcmp(lab,"**")==0)
        {
        if(strcmp(opcode,"START")==0)
        {
                fprintf(f3,"%s %s %s",lab,opcode,opa);
                locctr=(atoi(opa));

        }
        else
        {
                rewind(f2);
                x=0;
                while(fscanf(f2,"%s%s",opcode1,opa1)!=EOF)
                {
                if(strcmp(opcode,opcode1)==0)
                {
                x=1;
                }
                }
                if(x==1)
                {
                fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
                locctr=locctr+3;
                }
        }
        }
        else
        {
        if(strcmp(opcode,"RESW")==0)
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        fprintf(f4,"\n %d %s",locctr,lab);
        locctr=locctr+(3*(atoi(opa)));
        }
        else if(strcmp(opcode,"WORD")==0)
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        fprintf(f4,"\n %d %s",locctr,lab);
        locctr=locctr+3;
        }
        else if(strcmp(opcode,"BYTE")==0)
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        fprintf(f4,"\n %d %s",locctr,lab);
        locctr=locctr+1;
        }
        else if(strcmp(opcode,"RESB")==0)
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        fprintf(f4,"\n %d %s",locctr,lab);
        locctr=locctr+1;
        }
        else
        {
        fprintf(f3,"\n %d %s %s %s",locctr,lab,opcode,opa);
        fprintf(f4,"\n %d %s",locctr,lab);
        locctr=locctr+(atoi(opa));
        }
        }
}
}
```

INPUT FILES

<u>input.txt</u>

```
** START 2000
** LDA FIVE
** STA ALPHA
** LDCH CHARZ
** STCH C1
ALPHA RESW 1
FIVE WORD 5
CHARZ BYTE C'Z'
C1 RESB 1
** END **
```

<u>opcode.txt</u>

```
START *
LDA 03
STA 0F
LDCH 53
STCH 57
END
```

OUTPUT FILES

out1.txt

```
** START 2000
 2000 ** LDA FIVE
 2003 ** STA ALPHA
 2006 ** LDCH CHARZ
 2009 ** STCH C1
 2012 ALPHA RESW 1
 2015 FIVE WORD 5
 2018 CHARZ BYTE C'Z'
 2019 C1 RESB 1
 2020 ** END **
```

sym1.txt

```
 2012 ALPHA
 2015 FIVE
 2018 CHARZ
 2019 C1
```

Pgm.No.12

PASS TWO OF TWO PASS ASSEMBLER

AIM

Write a program to implement pass one of two pass assembler

PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
    {
char opcode[20],operand[20],symbol[20],label[20],code[20],mnemonic[25], character,
add[20],objectcode[20];
int flag,flag1,locctr,location,loc;
FILE *fp1,*fp2,*fp3,*fp4;

fp1=fopen("out3.txt","r"); fp2=fopen("twoout.txt","w");
fp3=fopen("opcode.txt","r"); fp4=fopen("sym1.txt","r");
fscanf(fp1,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{ fprintf(fp2,"%s\t%s\t%s\n",label,opcode,operand);
fscanf(fp1,"%d%s%s%s",&locctr,label,opcode,operand);
}
while(strcmp(opcode,"END")!=0)
{ flag=0;
fscanf(fp3,"%s%s",code,mnemonic);
while(strcmp(code,"END")!=0)
{ if((strcmp(opcode,code)==0) && (strcmp(mnemonic,"*"))!=0)
{ flag=1;
break;
}
fscanf(fp3,"%s%s",code,mnemonic);


}
if(flag==1)
{ flag1=0; rewind(fp4);
while(!feof(fp4))
{
fscanf(fp4,"%s%d",symbol,&loc);
if(strcmp(symbol,operand)==0)
{
flag1=1; break;
} }
if(flag1==1)
{
sprintf(add,"%d",loc);
strcpy(objectcode,strcat(mnemonic,add));
} }
else if(strcmp(opcode,"BYTE")==0 || strcmp(opcode,"WORD")==0)
{
if((operand[0]=='C') || (operand[0]=='X'))
{
character=operand[2];
sprintf(add,"%d",character);
strcpy(objectcode,add);
}
else
{
strcpy(objectcode,add);
} }
else
strcpy(objectcode,"\0");
fprintf(fp2,"%s\t%s\t%s\t%d\t%s\n",label,opcode,operand,locctr,objectcode);
fscanf(fp1,"%d%s%s%s",&locctr,label,opcode,operand);
}
fprintf(fp2,"%s\t%s\t%s\t%d\n",label,opcode,operand,locctr);
fclose(fp1);
fclose(fp2);
fclose(fp3);
 fclose(fp4);
}
```

INPUT FILES

opcode.txt

```
START *
LDA 03
STA 0F
LDCH 53
STCH 57
END +
```

out3.txt

```
** START 2000
2000 ** LDA FIVE
2003 ** STA ALPHA
2006 ** LDCH CHARZ
2009 ** STCH C1
2012 ALPHA RESW 1
2015 FIVE WORD 5
2018 CHARZ BYTE C'Z'
2019 C1 RESB 1
2020 ** END **
```

sym1.txt

```
 2012 ALPHA
 2015 FIVE
 2018 CHARZ
 2019 C1
```

OUTPUT FILES

twoout.txt

```
**      START           2000
**      LDA     FIVE    2000    032018
**      STA     ALPHA           2003    0F2015
**      LDCH    CHARZ           2006    532019
**      STCH    C1      2009    572019
ALPHA           RESW            1       2012
FIVE    WORD            5       2015    2019
CHARZ           BYTE    C'Z'    2018    90
C1      RESB    1       2019
**      END     **      2020
```