GOVT. MODEL ENGINEERING COLLEGE

THRIKKAKKARA, ERNAKULAM

CSL 331 SYSTEM SOFTWARE AND MICROPROCESSOR LAB

....

NAME: KEERTHANA J

BRANCH: COMPUTER SCIENCE AND ENGINEERING

SEMESTER : 05 **ROLL NO : 22CSA35**

Certified that this is the bonafide work done by

KEERTHANA J

....

Staff- in Charge ***Head of the Department***

Register No:.. Date:...

Year and month.. Thrikkakkara

Internal Examiner **External Examiner**

INDEX

SL.NO.	DATE	NAME OF THE PROGRAM	REMARKS
2		**Experiments from System Software**	
2.1		Pass 1 of Two Pass Assembler	
2.2		Pass 2 of Two Pass Assembler	
2.3		Macro Processor	
2.4		Absolute Loader	
2.5		Relocation Loader	
3		**Experiments from Microprocessors and Microcontrollers**	
3.1		Study of Assembler and Debugging Commands	
3.2			
3.2(i)		Addition of 16-bit numbers	
3.2(ii)		Subtraction of 16-bit numbers	
3.2(iii)		Multiplication of 16-bit numbers	
3.2(iv)		Division of 16-bit numbers	
3.3		Sorting and Searching (Kit)	
3.3(i)		Sorting of 16-bit numbers	
3.3(ii)		Searching of 16-bit numbers	
3.4		String Manipulation in emulator	
3.4(i)		Palindrome	
3.4(ii)		String Reversal	
3.5		Stepper Motor	
3.6		Interfacing with 8257	

EXPERIMENTS FROM OPERATING SYSTEMS

PROGRAM CODE:

```
#include<stdio.h>
int main() {
   int n,at[10],bt[10],wt[10],tat[10],ct[10],sum,i,j,k;
   float totaltat=0,totalwt=0;
   printf("Enter the total number of processes: ");
   scanf("%d",&n);
   printf("\nEnter The Process Arrival Time & Burst Time\n");
   for(i=0;i<n;i++) {
       printf("Enter Arrival time of process[%d]:",i+1);
       scanf("%d",&at[i]);
       printf("Enter Burst time of process[%d]:",i+1);
       scanf("%d",&bt[i]);

   }
  /*Calculate completion time of processes*/
  sum=at[0];
  for(j=0;j<n;j++) {
      sum=sum+bt[j];
          ct[j]=sum;
  }
   /*Calculate Turn Around time */
  for(k=0;k<n;k++)
  {
          tat[k]=ct[k]-at[k];
          totaltat=totaltat+tat[k];
  }
     /*  Calculate Waiting time  */
  for(k=0;k<n;k++)
  {
          wt[k]=tat[k]-bt[k];
  totalwt=totalwt+wt[k];
  }

   printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n\n\n");
 for(i=0;i<n;i++)
 {
      printf("\nP%d\t %d\t %d\t %d\t %d\t %d\t\n",i+1,at[i],bt[i],ct[i],tat[i],wt[i]);
}
    printf("\nAverage TurnaroundTime:%f\n",totaltat/n);
    printf("\nAverage Waiting Time:%f",totalwt/n);

    return 0;
}
```

Name: Keerthana J　　　　　　　　　　Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.1(i)

FCFS CPU SCHEDULING

AIM:

Simulate FCFS CPU scheduling algorithm to find turn around time and waiting time.

ALGORITHM:

```
Step 0: Start
Step 1: Input Initialization:
        Read the number of processes n.
        For each process, read its arrival time at[i] and burst time bt[i].
Step 2: Calculate Completion Time (CT):
        Initialize the sum of times with the arrival time of the first process.
        For each process:
                Add the burst time of the current process to the sum to get the
                completion time.
                Store the completion time in the ct array.
Step 3: Calculate Turnaround Time (TAT):
        For each process:
                Calculate the turnaround time as the difference between the
                completion time and the arrival time.
                Add the turnaround time to the total turnaround time totaltat.
Step 4: Calculate Waiting Time (WT):
        For each process:
                Calculate the waiting time as the difference between the turnaround
                time and the burst time.
                Add the waiting time to the total waiting time totalwt.
Step 5: Display Results:
        Print the table header.
        For each process:
                Print the process number, arrival time, burst time, completion time,
                turnaround time, and waiting time.
        Print the average turnaround time and average waiting time.
Step 6: Stop
```

OUTPUT:

```
Enter the total number of processes: 3

Enter The Process Arrival Time & Burst Time
Enter Arrival time of process[1]:0
Enter Burst time of process[1]:24
Enter Arrival time of process[2]:0
Enter Burst time of process[2]:3
Enter Arrival time of process[3]:0
Enter Burst time of process[3]:3

Process      AT     BT     CT     TAT    WT



P1           0       24     24     24     0

P2           0       3      27     27     24

P3           0       3      30     30     27

Average TurnaroundTime:27.000000

Average Waiting Time:17.000000
```

RESULT:

FCFS CPU scheduling algorithm is implemented successfully and waiting time and turn around time are found out.

PROGRAM CODE:

```
#include<stdio.h>

int main() {
    int n, at[10], bt[10], wt[10], tat[10], ct[10], pr[10], p[10], sum, i, j, k, pos, temp;
    float totaltat = 0, totalwt = 0;
    printf("Enter the total number of processes: ");
    scanf("%d", &n);

    printf("\nEnter The Process Arrival Time, Burst Time and Priority\n");
    for(i = 0; i < n; i++) {
        printf("Enter Arrival time of process[%d]: ", i + 1);
        scanf("%d", &at[i]);
        printf("Enter Burst time of process[%d]: ", i + 1);
        scanf("%d", &bt[i]);
        printf("Enter priority of process[%d]: ", i + 1);
        scanf("%d", &pr[i]);
        p[i]=i+1;
        printf("\n");
    }


    // Sort processes based on arrival time
    for(i = 0; i < n; i++) {
        for(j = 0; j < n - i - 1; j++) {
            if(at[j] > at[j + 1]) {

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

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

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

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

    // Calculate completion time, turnaround time, and waiting time
    sum = at[0];
    for(i = 0; i < n; i++) {
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.1(ii)

PRIORITY CPU SCHEDULING

AIM:

Simulate Priority CPU scheduling algorithm to find turn around time and waiting time.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
        Number of processes, n.
        Arrival Time (AT), Burst Time (BT), and Priority (PR) of each process.
Step 2: Initialization:
        sum = 0 (to keep track of the total time passed or CPU time).
        totaltat = 0, totalwt = 0 (to accumulate total turnaround and waiting time).
        Arrays for at[], bt[], pr[], p[] (arrival time, burst time, priority, process IDs).
        Arrays for ct[], tat[], wt[] (completion time, turnaround time, waiting time).
Step 3: Process Selection:
        For each process i in the list:
                Set pos = i (initially select the current process as the one to be
                executed next).
                For each process j after i (from j = i + 1 to n-1):
                        If process j has arrived (at[j] <= sum) and has a higher priority
                        (pr[j] < pr[pos]), update pos = j (select the process with the
                        highest priority).
Step 4: Process Swapping:
        If the process selected (pos) is different from the current process (i):
                Swap the arrival time, burst time, and priority of the current process i
                with the process at pos.
                Swap the process IDs accordingly.
Step 5: CPU Execution:
        If the CPU has to wait for a process to arrive (sum < at[i]), set sum = at[i].
        Add the burst time of the selected process to sum to calculate the completion
        time:
                sum += bt[i].
                Set the completion time: ct[i] = sum.
                Calculate the turnaround time: tat[i] = ct[i] - at[i].
                Calculate the waiting time: wt[i] = tat[i] - bt[i].
Step 6: Accumulate Time:
        Add tat[i] to totaltat.
        Add wt[i] to totalwt.
    // Find the process with the highest priority (lowest priority number) among the
processes that have arrived
    pos = i;
    for(j = i + 1; j < n; j++) {
        if(at[j] <= sum && pr[j] < pr[pos]) {
            pos = j;
        }
    }

    // Swap to bring the selected process to the current position
    if(pos != i) {
        temp = at[i];
        at[i] = at[pos];
        at[pos] = temp;

        temp = bt[i];
        bt[i] = bt[pos];
        bt[pos] = temp;

        temp = pr[i];
        pr[i] = pr[pos];
        pr[pos] = temp;

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

    if (sum < at[i]) {
        sum = at[i];
    }

    sum += bt[i];
    ct[i] = sum;
    tat[i] = ct[i] - at[i];
    wt[i] = tat[i] - bt[i];
    totaltat += tat[i];
    totalwt += wt[i];
}

printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
for(i = 0; i < n; i++) {
    printf("\nP%d\t %d\t %d\t %d\t %d\t %d",p[i], at[i], bt[i], ct[i], tat[i], wt[i]);
}

printf("\n\nAverage Turnaround Time: %.2f", totaltat / n);
printf("\nAverage Waiting Time: %.2f", totalwt / n);

return 0;
}
```

```
Step 7: Output:
    Print the table of processes showing:
        Process ID, Arrival Time (AT), Burst Time (BT), Completion Time
        (CT), Turnaround Time (TAT), Waiting Time (WT).
    Calculate and print the average turnaround time:
        Average Turnaround Time = totaltat / n.
    Calculate and print the average waiting time:
        Average Waiting Time = totalwt / n.
Step 8: Stop
```

OUTPUT:

```
Enter the total number of processes: 5

Enter The Process Arrival Time, Burst Time and Priority
Enter Arrival time of process[1]: 0
Enter Burst time of process[1]: 3
Enter priority of process[1]: 3

Enter Arrival time of process[2]: 1
Enter Burst time of process[2]: 6
Enter priority of process[2]: 4

Enter Arrival time of process[3]: 3
Enter Burst time of process[3]: 1
Enter priority of process[3]: 9

Enter Arrival time of process[4]: 2
Enter Burst time of process[4]: 2
Enter priority of process[4]: 7

Enter Arrival time of process[5]: 4
Enter Burst time of process[5]: 4
Enter priority of process[5]: 8


Process     AT     BT     CT     TAT    WT

P1          0      3      3      3      0
P2          1      6      9      8      2
P4          2      2      11     9      7
P3          4      4      15     11     7
P5          3      1      16     13     12

Average Turnaround Time: 8.80
Average Waiting Time: 5.60
```

RESULT:

Priority CPU scheduling algorithm is implemented successfully and waiting time and turn around time are found out.

<u>PROGRAM CODE</u>:

```
#include <stdio.h>

int main() {
    int i,j, n, time=0, remain, flag = 0, ts;
    int at[10], bt[10], ct[10], wt[10], tat[10], rt[10];
    float totaltat = 0, totalwt = 0;
    printf("Enter the total number of processes: ");
    scanf("%d", &n);
    remain = n;

    printf("\nEnter The Process Arrival Time and Burst Time\n");
    for (i = 0; i < n; i++) {
        printf("Enter Arrival time of process[%d]: ", i + 1);
        scanf("%d", &at[i]);
        printf("Enter Burst time of process[%d]: ", i + 1);
        scanf("%d", &bt[i]);
        rt[i] = bt[i];
    }

    printf("\nEnter time slice: ");
    scanf("%d", &ts);

    int queue[50],rear=0,front=0;

    for(i=0;i<n;i++) {
        if(at[i]==0) {
            queue[rear++]=i;
        }
    }

    while (remain != 0) {

        i=queue[front++];

        if (rt[i] <= ts && rt[i] > 0) {
            time += rt[i];
            rt[i] = 0;
            flag = 1;
        } else if (rt[i] > 0) {
            rt[i] -= ts;
            time += ts;
        }

        if (rt[i] == 0 && flag == 1) {
            remain--;
            ct[i] = time;
            tat[i] = ct[i] - at[i];
            wt[i] = tat[i] - bt[i];
            totaltat += tat[i];
            totalwt += wt[i];
            flag = 0;
        }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.1(iii)

ROUND ROBIN CPU SCHEDULING

AIM:

Simulate Round Robin CPU scheduling algorithm to find turn around time and waiting time.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
        Number of processes n.
        Arrival Time (AT) and Burst Time (BT) for each process.
        Time Slice (ts).
Step 2: Initialization:
        time = 0 (to keep track of the total elapsed time).
        remain = n (remaining number of processes to be completed).
        rt[] (remaining time for each process initialized to burst time).
        Arrays at[], bt[], ct[], tat[], wt[] (arrival time, burst time, completion time,
        turnaround time, waiting time).
        Queue queue[] for keeping track of processes.
Step 3: Input Process Arrival, Burst Time and Time Slice:
        For each process i:
                Read arrival time (at[i]).
                Read burst time (bt[i]).
                Initialize rt[i] (remaining time) to bt[i].
        Read the time slice ts.
Step 4: Initial Queue Setup:
        Add the processes that have an arrival time of 0 to the queue.
Step 5: Round Robin Execution:
        While there are remaining processes (remain != 0):
        •   Process Selection:
                Select the process at the front of the queue.
        •   Execution:
                If the remaining time (rt[i]) of the selected process is less than or equal
                to the time slice (ts):
                        Add the remaining time to the total time.
                        Set rt[i] to 0 (process completes).
                        Set flag = 1 (indicating that the process is completed).
                Else, reduce the remaining time by the time slice (rt[i] -= ts) and
                increase time by ts.
        •   Completion:
                If the process is completed (rt[i] == 0 and flag == 1):
                        Decrease the remaining process count (remain--).
                        Set completion time ct[i] = time.
```

```
//processes that arrived during time slice
        for(j=0;j<n;j++) {
            if(at[j]>time-ts && at[j]<=time && rt[j]>0) {
                queue[rear++] = j;
            }
        }

        //if current process has not finished executing
        if(rt[i]>0) {
            queue[rear++] = i;
        }

    }

    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n");
    for (i = 0; i < n; i++) {
        printf("\nP%d\t %d\t %d\t %d\t %d\t %d", i + 1, at[i], bt[i], ct[i], tat[i], wt[i]);
    }
    printf("\n\nAverage Turnaround Time: %f", totaltat / n);
    printf("\nAverage Waiting Time: %f\n", totalwt / n);
    return 0;
}
```

```
                Calculate turnaround time: tat[i] = ct[i] - at[i].
                Calculate waiting time: wt[i] = tat[i] - bt[i].
                Accumulate tat[i] into totaltat and wt[i] into totalwt.
                Reset flag = 0.
      • Check for New Arrivals:
            For each process j:
                If it arrived during the current time slice (at[j] > time - ts && at[j]
                <= time), add it to the queue if it hasn’t finished execution.
      • Requeue the Process:
            If the current process has not finished execution (rt[i] > 0), add it back
            to the queue.
Step 6: Output:
        Print the table of processes showing:
            Process ID, Arrival Time (AT), Burst Time (BT), Completion Time
            (CT), Turnaround Time (TAT), Waiting Time (WT).
        Calculate and print the average turnaround time:
            Average Turnaround Time = totaltat / n.
        Calculate and print the average waiting time:
            Average Waiting Time = totalwt / n.
Step 7: Stop
```

<u>OUTPUT</u>:

```
Enter the total number of processes: 4

Enter The Process Arrival Time and Burst Time
Enter Arrival time of process[1]: 0
Enter Burst time of process[1]: 5
Enter Arrival time of process[2]: 1
Enter Burst time of process[2]: 4
Enter Arrival time of process[3]: 2
Enter Burst time of process[3]: 2
Enter Arrival time of process[4]: 4
Enter Burst time of process[4]: 1

Enter time slice: 2

Process         AT      BT      CT      TAT     WT

P1              0       5       12      12      7
P2              1       4       11      10      6
P3              2       2       6       4       2
P4              4       1       9       5       4

Average Turnaround Time: 7.750000
Average Waiting Time: 4.750000
```

RESULT:

Round Robin CPU scheduling algorithm is implemented successfully and waiting time and turn around time are found out.

PROGRAM CODE:

```
#include<stdio.h>
int main() {
    int n,at[10],bt[10],wt[10],tat[10],ct[10],p[10],sum,i,j,k,temp;
    float totaltat=0,totalwt=0;
    printf("Enter the total number of processes:");
    scanf("%d",&n);
    for(i=0;i<n;i++) {
        printf("Enter process id: P");
        scanf("%d",&p[i]);
        printf("Enter Arrival time of P[%d]: ",p[i]);
        scanf("%d",&at[i]);
        printf("Enter Burst time of P[%d]: ",p[i]);
        scanf("%d",&bt[i]);
        printf("\n");
    }
    /* Apply Bubble sort to sort according to arrival time */
    for(i=0;i<n;i++) {
        for(j=0;j<n-i-1;j++) {
            if(at[j]>at[j+1]) {
                temp=at[j];
                at[j]=at[j+1];
                at[j+1]=temp;

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

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


    // Calculate completion time, turnaround time and waiting time
    sum = at[0];
    for(i = 0; i < n; i++) {
        // Find the process with the minimum burst time among the processes that have
arrived
        int min = i;
        for(j = i + 1; j < n; j++) {
            if(at[j] <= sum && bt[j] < bt[min]) {
                min = j;
            }
        }

        // Swap to bring the selected process to the current position
        if(min != i) {
            temp = at[i];
            at[i] = at[min];
            at[min] = temp;
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.1(iv)

SJF CPU SCHEDULING

AIM:

Simulate SJF CPU scheduling algorithm to find turn around time and waiting time.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
    Number of processes n.
    Arrival Time (AT) and Burst Time (BT) for each process.
Step 2: Initialization:
    sum = 0 (to keep track of the current time).
    Arrays for at[], bt[], ct[], tat[], wt[] (arrival time, burst time, completion time,
    turnaround time, waiting time).
    totaltat = 0, totalwt = 0 (to accumulate total turnaround and waiting time).
Step 3: Process Selection:
    For each process i:
        Select the process with the shortest burst time (SJF) from the list of
        processes that have arrived and are not yet executed:
        o Set min = i (initially assume the current process has the shortest
          burst time).
        o For each process j from i+1 to n-1:
              If the process j has arrived (at[j] <= sum) and has a shorter
              burst time than the current shortest (bt[j] < bt[min]), update min
              = j (select process j as the process with the shortest burst
              time).
Step 4: Process Swapping:
    If the process selected (min) is not the current process (i):
        Swap the burst time bt[i] with bt[min].
        Swap the process ID p[i] with p[min] to maintain correct process
        tracking.
Step 5: CPU Execution:
    Add the burst time of the selected process to sum:
        sum += bt[i] (move forward in time by the burst time of the current
        process).
        Set completion time: ct[i] = sum.
        Calculate turnaround time: tat[i] = ct[i] - at[i].
        Calculate waiting time: wt[i] = tat[i] - bt[i].
Step 6: Accumulate Time:
    Add tat[i] to totaltat.
    Add wt[i] to totalwt.
```

```
            temp = bt[i];
            bt[i] = bt[min];
            bt[min] = temp;

            temp = p[i];
            p[i] = p[min];
            p[min] = temp;
        }

        sum += bt[i];
        ct[i] = sum;
        tat[i] = ct[i] - at[i];
        wt[i] = tat[i] - bt[i];
        totaltat += tat[i];
        totalwt += wt[i];
    }


    printf("\nProcess\tAT\tBT\tCT\tTAT\tWT\n\n\n");
    for(i=0;i<n;i++) {
        printf("\nP%d\t %d\t %d\t %d\t %d\t %d\n",p[i],at[i],bt[i],ct[i],tat[i],wt[i]);
    }

    printf("\nAverage TurnaroundTime:%f\n",totaltat/n);
    printf("\nAverage Waiting Time:%f",totalwt/n);

    return 0;
}
```

```
Step 7: Output:
        Print the table of processes showing:
                Process ID, Arrival Time (AT), Burst Time (BT), Completion Time
                (CT), Turnaround Time (TAT), Waiting Time (WT).
        Calculate and print the average turnaround time:
                Average Turnaround Time = totaltat / n.
        Calculate and print the average waiting time:
                Average Waiting Time = totalwt / n.
Step 8: Stop
```

OUTPUT:

```
Enter the total number of processes:5
Enter process id: P1
Enter Arrival time of P[1]: 1
Enter Burst time of P[1]: 7

Enter process id: P2
Enter Arrival time of P[2]: 3
Enter Burst time of P[2]: 3

Enter process id: P3
Enter Arrival time of P[3]: 6
Enter Burst time of P[3]: 2

Enter process id: P4
Enter Arrival time of P[4]: 7
Enter Burst time of P[4]: 10

Enter process id: P5
Enter Arrival time of P[5]: 9
Enter Burst time of P[5]: 8


Process      AT     BT     CT     TAT    WT



P1           1      7      8      7      0

P3           6      2      10     4      2

P2           3      3      13     10     7

P5           9      8      21     12     4

P4           7      10     31     24     14

Average TurnaroundTime:11.400000

Average Waiting Time:5.400000
```

RESULT:

SJF CPU scheduling algorithm is implemented successfully and waiting time and turn around time are found out.

PROGRAM CODE:

```
#include<stdio.h>

void main() {
    int i,j,k,f,pf=0,count=0,rs[25],m[10],n;
    printf("Enter the length of reference string: ");
    scanf("%d",&n);
    printf("Enter the reference string: ");
    for(i=0;i<n;i++) {
        scanf("%d",&rs[i]);
    }
    printf("Enter no of frames: ");
    scanf("%d",&f);
    for(i=0;i<f;i++) {
        m[i]=-1;
    }


    for(i=0;i<n;i++) {
        for(k=0;k<f;k++) {
            if(m[k]==rs[i]) {  // page hit
                break;
            }
        }
        if(k==f) {
            m[count++] = rs[i];
            pf++;
        }
        for(j=0;j<f;j++) {
            printf("\t%d",m[j]);
        }
        if(k==f) {
            printf("\tPF No. %d",pf);
        }
        printf("\n");
        if(count==f) {
            count=0;
        }
    }
    printf("\n The number of page faults using FIFO are %d",pf);
}
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.2(i)

FIFO PAGE REPLACEMENT

AIM:

Simulate FIFO page replacement algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
    n: Length of the reference string.
    rs[]: Reference string, an array containing the sequence of page requests.
    f: Number of frames in memory.
    pf: Page fault count, initialized to 0.
    count: Counter to keep track of the oldest page in frames, initialized to 0.
    m[]: Array to represent the frames, initialized to -1 indicating empty frames.
Step 2: Input the Reference String and Number of Frames:
    Read the reference string of length n.
    Read the number of frames f.
Step 3: Process Each Page Request in the Reference String:
    For each page rs[i] in the reference string:
        Check if the page is already in one of the frames:
            • If yes, it is a page hit; no page fault occurs.
            • If no, it is a page fault:
                    Load the page into the frame pointed to by count.
                    Increment the page fault count pf.
                    Increment count and reset to 0 if it reaches f (circular
                    queue behavior).
Step 4: Output the Frame State and Page Faults:
    After each page request, print the current state of frames.
    Print the total number of page faults at the end.
Step 5: Stop
```

OUTPUT:

```
Enter the length of reference string: 20
Enter the reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter no of frames: 4
          1       -1      -1      -1      PF No. 1
          1       2       -1      -1      PF No. 2
          1       2       3       -1      PF No. 3
          1       2       3       4       PF No. 4
          1       2       3       4
          1       2       3       4
          5       2       3       4       PF No. 5
          5       6       3       4       PF No. 6
          5       6       2       4       PF No. 7
          5       6       2       1       PF No. 8
          5       6       2       1
          3       6       2       1       PF No. 9
          3       7       2       1       PF No. 10
          3       7       6       1       PF No. 11
          3       7       6       1
          3       7       6       2       PF No. 12
          1       7       6       2       PF No. 13
          1       7       6       2
          1       3       6       2       PF No. 14
          1       3       6       2

 The number of page faults using FIFO are 14
```

RESULT:

FIFO page replacement algorithm is implemented successfully and page faults are found out.

PROGRAM CODE:

```
#include <stdio.h>
void main()
{
    int q[20], p[50], c = 0, c1, d, f, i, j, k = 0, n, r, t, b[20], c2[20];
    printf("Enter no of pages: ");
    scanf("%d", &n);
    printf("Enter the reference string: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &p[i]);
    }
    printf("Enter no of frames: ");
    scanf("%d", &f);
    q[k] = p[k];
    printf("\n\t%d\n", q[k]);
    c++;
    k++;
    for (i = 1; i < n; i++) {
        c1 = 0;
        for (j = 0; j < f; j++) {
            if (p[i] != q[j]) {
                c1++;
            }
        }
        if (c1 == f) {
            c++;
            if (k < f) {
                q[k] = p[i];
                k++;
                for (j = 0; j < k; j++) {
                    printf("\t%d", q[j]);
                }
                printf("\n");
            }
            else {
                for (r = 0; r < f; r++) {
                    c2[r] = 0;
                    for (j = i - 1; j < n; j--) {
                        if (q[r] != p[j]) {
                            c2[r]++;
                        }
                        else {
                            break;
                        }
                    }
                }
                for (r = 0; r < f; r++) {
                    b[r] = c2[r];
                }
                for (r = 0; r < f; r++) {
                    for (j = r; j < f; j++) {
                        if (b[r] < b[j]) {
                            t = b[r];
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.2(ii)

LFU PAGE REPLACEMENT

AIM:

Simulate LFU page replacement algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
    n: Number of pages in the reference string.
    p[]: Reference string, an array containing the sequence of page requests.
    f: Number of frames in memory.
    c: Page fault count, initialized to 0.
    k: Frame index, initialized to 0.
    q[]: Array representing the frames, initialized to store the pages.
    b[]: Array used to find the page with the highest count (used for sorting).
    c2[]: Array to keep the count of how many pages have been used before the
    current page.
Step 2: Input the Reference String and Number of Frames:
    Read the reference string of length n.
    Read the number of frames f.
Step 3: Process the First Page Request:
    Load the first page into the first frame, increment the page fault count.
Step 4: Process Each Page Request in the Reference String:
    For each page p[i] in the reference string starting from the second page:
        Check if the page is already in one of the frames:
            • If yes, it is a page hit; continue to the next page request.
            • If no, it is a page fault:
                ▪ Increment the page fault count c.
                ▪ If there are empty frames available, load the page into
                  the next available frame.
                ▪ If all frames are full:
                    o Calculate the frequency of each page in the
                      frames.
                    o Replace the page with the least frequency with
                      the new page.
Step 5: Output the Frame State and Page Faults:
        After each page request, print the current state of frames.
        Print the total number of page faults at the end.
Step 6: Stop
```

```
                b[r] = b[j];
                b[j] = t;
            }
        }
    }
    for (r = 0; r < f; r++) {
        if (c2[r] == b[0]) {
            q[r] = p[i];
        }
        printf("\t%d", q[r]);
    }
    printf("\n");
    }
  }
}
printf("\nThe no of page faults is %d", c);
}
```

OUTPUT:

```
Enter no of pages: 15
Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2
Enter no of frames: 3

        7
        7       1
        7       1       2
        0       1       2
        0       3       2
        0       3       4
        0       2       4
        3       2       4
        3       2       0
        3       2       1

The no of page faults is 10
```

RESULT:

LFU page replacement algorithm is implemented successfully and page faults are found out.

PROGRAM CODE:

```
#include<stdio.h>

void main() {
    int i,j,k,min,flag[25],count[10],rs[25],m[10],n,f,pf=0,next;
    printf("Enter the length of reference string: ");
    scanf("%d",&n);
    printf("Enter the reference string: ");
    for(i=0;i<n;i++) {
        scanf("%d",&rs[i]);
        flag[i]=0;
    }
    printf("Enter no of frames: ");
    scanf("%d",&f);
    for(i=0;i<f;i++) {
        count[i]=0;
        m[i]=-1;
    }


    for(i=0;i<n;i++) {
        for(j=0;j<f;j++) {
            if(m[j]==rs[i]) {
                flag[i]=1;
                count[j]=next;
                next++;
            }
        }
        if(flag[i]==0) {
            if(i<f) {
                m[i]=rs[i];
                count[i]=next;
                next++;
            }
            else {
                min=0;
                for(j=1;j<f;j++) {
                    if(count[min]>count[j]) {
                        min=j;
                    }
                }
                m[min]=rs[i];
                count[min]=next;
                next++;
            }
            pf++;
        }

        for(j=0;j<f;j++) {
            printf("%d\t",m[j]);
        }
        if(flag[i]==0) {
            printf("PF No. %d",pf);
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.2(iii)

LRU PAGE REPLACEMENT

AIM:

Simulate LRU page replacement algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
    n: Length of the reference string.
    rs[]: Reference string, an array containing the sequence of page requests.
    f: Number of frames in memory.
    pf: Page fault count, initialized to 0.
    next: A counter used to keep track of the sequence of page references.
    count[]: Array to keep track of the usage order (timestamp) of pages in
    frames.
    m[]: Array to represent the frames, initialized to -1 indicating empty frames.
    flag[]: Array to indicate if a page hit occurs for a reference.
Step 2: Input the Reference String and Number of Frames:
    Read the reference string of length n.
    Read the number of frames f.
Step 3: Process Each Page Request in the Reference String:
    For each page rs[i] in the reference string:
        Check if the page is already in one of the frames:
            • If yes, it is a page hit; update the usage order (count[]).
            • If no, it is a page fault:
                ▪ If there are empty frames, load the page into the next
                  available frame and update count[].
                ▪ If all frames are full, find the frame with the least recent
                  usage (minimum count[]), replace it, and update
                  count[].
                ▪ Increment the page fault count pf.
Step 4: Output the Frame State and Page Faults:
    After each page request, print the current state of frames.
    Print the total number of page faults at the end.
Step 5: Stop
```

```
        }
        printf("\n");
    }
    printf("\nThe number of page faults using LRU are %d",pf);
}
```

OUTPUT:

```
Enter the length of reference string: 20
Enter the reference string: 1 2 3 4 2 1 5 6 2 1 2 3 7 6 3 2 1 2 3 6
Enter no of frames: 4
1       -1      -1      -1      PF No. 1
1       2       -1      -1      PF No. 2
1       2       3       -1      PF No. 3
1       2       3       4       PF No. 4
1       2       3       4
1       2       3       4
1       2       5       4       PF No. 5
1       2       5       6       PF No. 6
1       2       5       6
1       2       5       6
1       2       5       6
1       2       3       6       PF No. 7
1       2       3       7       PF No. 8
6       2       3       7       PF No. 9
6       2       3       7
6       2       3       7
6       2       3       1       PF No. 10
6       2       3       1
6       2       3       1
6       2       3       1

The number of page faults using LRU are 10
```

<u>RESULT</u>:

FIFO page replacement algorithm is implemented successfully and page faults are found out.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>

void main() {
   int RQ[100],i,n,TotalHeadMoment=0,initial;
   printf("Enter the no of requests: ");
   scanf("%d",&n);
   printf("Enter the requests sequence: ");
   for(i=0;i<n;i++) {
      scanf("%d",&RQ[i]);
   }
   printf("Enter initial head position: ");
   scanf("%d",&initial);

   for(i=0;i<n;i++) {
      TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
      initial=RQ[i];
   }

   printf("Total head moment is %d",TotalHeadMoment);
}
```

OUTPUT:

```
Enter the no of requests: 8
Enter the requests sequence: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Total head moment is 640
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.3(i)

FCFS DISK SCHEDULING

AIM:

Simulate FCFS disk scheduling algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
            n: Number of disk requests.
            RQ[]: Sequence of disk requests
        TotalHeadMovement to 0, which will store the total distance the disk head
        moves.
        initial to store the initial position of the disk head.
Step 2: Input the Number of Requests and the Sequence of Requests:
        Read the number of disk requests n.
        Read the sequence of disk requests RQ[].
Step 3: Input the Initial Head Position:
        Read the initial position of the disk head initial.
Step 4: Process Each Disk Request:
        For each request RQ[i] in the sequence:
            •   Calculate the absolute distance the disk head moves from its current
                position to the request position.
            •   Add this distance to TotalHeadMovement.
            •   Update the disk head's current position to the request position.
Step 5: Output the Total Head Movement:
        Print the total head movement.
Step 6: Stop
```

RESULT:

FCFS disk scheduling algorithm is implemented successfully.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>

void main() {
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move,temp,index;
    printf("Enter the no of requests: ");
    scanf("%d",&n);
    printf("Enter the requests sequence: ");
    for(i=0;i<n;i++) {
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position: ");
    scanf("%d",&initial);
    printf("Enter total disk size: ");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and low 0: ");
    scanf("%d",&move);

    //Sort the request array
    for(i=0;i<n;i++) {
        for(j=0;j<n-i-1;j++) {
            if(RQ[j]>RQ[j+1]) {
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }
        }
    }

    for(i=0;i<n;i++) {
        if(initial<RQ[i]) {
            index=i;
            break;
        }
    }

    //if movement is towards high value
    if(move==1) {
        for(i=index;i<n;i++) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        //last movement for max size
        TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
        initial=size-1;
        for(i=index-1;i>=0;i--) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }

    //if movement is towards low value
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.3(ii)

SCAN DISK SCHEDULING

AIM:

Simulate SCAN disk scheduling algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
    n: Number of disk requests.
    RQ[]: Sequence of disk requests.
    TotalHeadMovement to 0.
    initial to store the initial position of the disk head.
    size to store the total size of the disk.
    move to indicate the direction of head movement (1 for high, 0 for low).
Step 2: Input the Number of Requests and the Sequence of Requests:
    Read the number of disk requests n.
    Read the sequence of disk requests RQ[].

Step 3: Input the Initial Head Position, Disk Size, and Head Movement Direction:
    Read the initial position of the disk head initial.
    Read the total disk size size.
    Read the head movement direction move.
Step 4: Sort the Request Array:
    Sort the array RQ[] in ascending order.
Step 5: Find the Index of the First Request Greater Than the Initial Position:
    Determine the index index where the first request greater than the initial
    position is located in RQ[].
Step 6: Service Requests Based on Head Movement Direction:
    If move is towards high values (1):
        • Service requests from index to n-1.
        • Move the head to the end of the disk (size - 1).
        • Service the remaining requests from index-1 to 0.
    If move is towards low values (0):
        • Service requests from index-1 to 0.
        • Move the head to the start of the disk (0).
        • Service the remaining requests from index to n-1.
Step 7: Output the Total Head Movement:
    Print the total head movement.
Step 8: Stop
```

```
    else{
        for(i=index-1;i>=0;i--) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        //last movement for min size
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        initial=0;
        for(i=index;i<n;i++) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }

    printf("Total head movement is %d",TotalHeadMoment);
}
```

OUTPUT:

```
Enter the no of requests: 8
Enter the requests sequence: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Enter total disk size: 200
Enter the head movement direction for high 1 and low 0: 0
Total head movement is 236
```

RESULT:

SCAN disk scheduling algorithm is implemented successfully.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int RQ[100],i,j,n,TotalHeadMoment=0,initial,size,move,index,temp;
    printf("Enter the number of Requests: ");
    scanf("%d",&n);
    printf("Enter the Requests sequence: ");
    for(i=0;i<n;i++) {
        scanf("%d",&RQ[i]);
    }
    printf("Enter initial head position: ");
    scanf("%d",&initial);
    printf("Enter total disk size: ");
    scanf("%d",&size);
    printf("Enter the head movement direction for high 1 and for low 0: ");
    scanf("%d",&move);
    /* logic for C-Scan disk scheduling */
    /*logic for sort the request array */
    for(i=0;i<n;i++) {
        for( j=0;j<n-i-1;j++) {
            if(RQ[j]>RQ[j+1]) {
                temp=RQ[j];
                RQ[j]=RQ[j+1];
                RQ[j+1]=temp;
            }
        }
    }
    for(i=0;i<n;i++) {
        if(initial<RQ[i]) {
            index=i;
            break;
        }
    }

    /* if movement is towards high value */
    if(move==1) {
        for(i=index;i<n;i++) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        /* last movement for max size*/
        TotalHeadMoment=TotalHeadMoment+abs(size-RQ[i-1]-1);
        /*movement max to min disk */
        TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
        initial=0;
        for( i=0;i<index;i++) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.3(iii)

C-SCAN DISK SCHEDULING

AIM:

Simulate C-SCAN disk scheduling algorithm.

ALGORITHM:

```
Step 0: Start
Step 1: Initialize Variables:
        n: Number of disk requests.
        RQ[]: Sequence of disk requests.
        TotalHeadMovement to 0.
        initial to store the initial position of the disk head.
        size to store the total size of the disk.
        move to indicate the direction of head movement (1 for high, 0 for low).
Step 2: Input the Number of Requests and the Sequence of Requests:
        Read the number of disk requests n.
        Read the sequence of disk requests RQ[].
Step 3: Input the Initial Head Position, Disk Size, and Head Movement Direction:
        Read the initial position of the disk head initial.
        Read the total disk size size.
        Read the head movement direction move.
Step 4: Sort the Request Array:
        Sort the array RQ[] in ascending order using bubble sort.
Step 5: Find the Index of the First Request Greater Than the Initial Position:
        Determine the index index where the first request greater than the initial
        position is located in RQ[].
Step 6: Service Requests Based on Head Movement Direction:
        If move is towards high values (1):
            • Service requests from index to n-1.
            • Move the head to the end of the disk (size - 1).
            • Jump to the beginning of the disk (0).
            • Service the remaining requests from 0 to index-1.
        If move is towards low values (0):
            • Service requests from index-1 to 0.
            • Move the head to the beginning of the disk (0).
            • Jump to the end of the disk (size - 1).
            • Service the remaining requests from n-1 to index.
Step 7: Output the Total Head Movement:
        Print the total head movement.
Step 8: Stop
```

```
    /* if movement is towards low value */
    else {
        for(i=index-1;i>=0;i--) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
        /* last movement for min size */
        TotalHeadMoment=TotalHeadMoment+abs(RQ[i+1]-0);
        /*movement min to max disk */
        TotalHeadMoment=TotalHeadMoment+abs(size-1-0);
        initial =size-1;
        for(i=n-1;i>=index;i--) {
            TotalHeadMoment=TotalHeadMoment+abs(RQ[i]-initial);
            initial=RQ[i];
        }
    }

    printf("Total head movement is %d",TotalHeadMoment);
    return 0;
}
```

OUTPUT:

```
Enter the number of Requests: 8
Enter the Requests sequence: 98 183 37 122 14 124 65 67
Enter initial head position: 53
Enter total disk size: 200
Enter the head movement direction for high 1 and for low 0: 1
Total head movement is 382
```

RESULT:

C-SCAN disk scheduling algorithm is implemented successfully.

PROGRAM CODE:

```
#include<stdio.h>
struct file {
    int name;
    int sb,length;
}
f[30];

int main() {
    int i,j,n;
    char s[20];
    printf("Enter no of files :");
    scanf("%d",&n);
    for(i=0;i<n;i++) {
        printf("\nEnter file name : ");
        scanf("%d",&f[i].name);
        printf("Enter starting block of file %d :",f[i].name);
        scanf("%d",&f[i].sb);
        printf("Enter the length of the file %d :",f[i].name);
        scanf("%d",&f[i].length);
        printf("\n");
    }

    for(i=0;i<n;i++) {
        printf("\nFile Name: %d",f[i].name);
        printf("\nStart Block: %d",f[i].sb);
        printf("\nLength: %d",f[i].length);
        printf("\nFile is allocated from block %d to block %d",f[i].sb,f[i].sb+f[i].length-1);
        printf("\n");

        /*for(j=0;j<f[i].length;j++)
            printf("%d, ",f[i].sb+j);
        */
    }

    return 0;
}
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.4(i)

SEQUENTIAL FILE ALLOCATION

AIM:

Simulate Sequential file allocation strategies.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
        Number of files n.
        For each file, input:
                File name.
                Starting block (sb) of the file.
                Length of the file (number of blocks).
Step 2: Initialization:
        Create a structure file to store information about each file:
                name: Name of the file.
                sb: Starting block of the file.
                length: Length of the file (in number of blocks).
        Declare an array f[] to store details of up to 30 files.
Step 3: Input File Details:
        For each file i (from i = 0 to n-1):
                Input the file name (f[i].name).
                Input the starting block (f[i].sb).
                Input the length of the file (f[i].length).
Step 4: File Allocation Information:
        For each file i (from i = 0 to n-1):
                Print the file name.
                Print the starting block of the file.
                Print the length of the file.
                Calculate and print the range of blocks allocated to the file:
                        Start block: f[i].sb.
                        End block: f[i].sb + f[i].length - 1.
        For each block allocated to the file, print the block numbers from f[i].sb to
        f[i].sb + f[i].length - 1.
Step 5: Output:
        Print the allocation details for each file:
                File name.
                Starting block.
                Length of the file.
                Block range allocated to the file (from start block to end block).
Step 6: Stop
```

OUTPUT:

```
Enter no of files :3

Enter file name : 1
Enter starting block of file 1 :0
Enter the length of the file 1 :10


Enter file name : 2
Enter starting block of file 2 :12
Enter the length of the file 2 :5


Enter file name : 3
Enter starting block of file 3 :20
Enter the length of the file 3 :8


File Name: 1
Start Block: 0
Length: 10
File is allocated from block 0 to block 9

File Name: 2
Start Block: 12
Length: 5
File is allocated from block 12 to block 16

File Name: 3
Start Block: 20
Length: 8
File is allocated from block 20 to block 27
```

<u>RESULT</u>:

Sequential file allocation strategy is implemented successfully.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>

struct file
{
    int nob, blocks[30], start, length;
}
f[30];

int main() {
    int i, j, n, s, ch;
    printf("Enter no of files : ");
    scanf("%d",&n);
    for(i=0;i<n;i++) {
        printf("Enter the start address of file %d: ",i+1);
        scanf("%d",&f[i].start);
        printf("Enter the length of file %d: ",i+1);
        scanf("%d",&f[i].length);
        printf("Enter no of blocks in file %d: ",i+1);
        scanf("%d",&f[i].nob);
        printf("Enter the blocks of the file: ");
        for(j=0;j<f[i].nob;j++)
            scanf("%d",&f[i].blocks[j]);
        printf("\n");
    }

    for(i=0;i<n;i++) {
        printf("File no: %d",i+1);
        printf("\tNo of blocks: %d",f[i].nob);
        printf("\t\tLength: %d",f[i].length);
        printf("\tStart address: %d",f[i].start);
        printf("\n");

    }


    printf("1. Get the blocks\t2. Exit\n");
    scanf("%d",&ch);
    while(ch!=2) {
        if(ch==1) {
            printf("\nEnter the file number: ");
            scanf("%d",&s);
            printf("Blocks allocated : \n");
            for(j=0;j<f[s-1].nob;j++)
                printf("%d\t",f[s-1].blocks[j]);
            printf("\n");
        }
        else if(ch==2) {
            exit(0);
        }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.4(ii)

INDEXED FILE ALLOCATION

AIM:

Simulate Indexed file allocation strategies.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
        Number of files n.
        For each file, input:
        Start address.
        Length of the file.
        Number of blocks in the file (nob).
        The list of blocks allocated to the file.
Step 2: Initialization:
        Define a structure file with the following fields:
                start: Starting address of the file.
                length: Length of the file.
                nob: Number of blocks.
                blocks[]: Array to store the blocks allocated to the file.
        Declare an array f[] to store the information for up to 30 files.
Step 3: Input File Details:
        For each file i (from i = 0 to n-1):
                Input the starting address of the file (f[i].start).
                Input the length of the file (f[i].length).
                Input the number of blocks allocated to the file (f[i].nob).
                Input the block numbers that are allocated to the file, and store them
                in f[i].blocks[].
Step 4: Display File Information:
        For each file i (from i = 0 to n-1):
                Print:
                The file number i+1.
                The number of blocks allocated to the file (f[i].nob).
                The length of the file (f[i].length).
                The starting address of the file (f[i].start).
Step 5: Option to Retrieve Block Information:
        Present the user with two choices:
            • Get the blocks:
                Input the file number s (selected by the user).
                Display the list of blocks allocated to the selected file by accessing f[s-
                1].blocks[].
            • Exit:
                Exit the program.
        else {
            printf("Invalid choice");
        }
        printf("\n1. Get the blocks\t2. Exit\n");
        scanf("%d",&ch);
    }

    return 0;
}
```

OUTPUT:

```
Enter no of files : 3
Enter the start address of file 1: 5
Enter the length of file 1: 7
Enter no of blocks in file 1: 4
Enter the blocks of the file: 3 4 5 6

Enter the start address of file 2: 12
Enter the length of file 2: 6
Enter no of blocks in file 2: 3
Enter the blocks of the file: 7 8 9

Enter the start address of file 3: 22
Enter the length of file 3: 3
Enter no of blocks in file 3: 2
Enter the blocks of the file: 10 11

File no: 1      No of blocks: 4              Length: 7      Start address: 5
File no: 2      No of blocks: 3              Length: 6      Start address: 12
File no: 3      No of blocks: 2              Length: 3      Start address: 22
1. Get the blocks       2. Exit
1

Enter the file number: 1
Blocks allocated :
3       4       5       6

1. Get the blocks       2. Exit
1

Enter the file number: 2
Blocks allocated :
7       8       9

1. Get the blocks       2. Exit
2
```

```
Step 6: Handle User Choices:
    If the user selects option 1 (to retrieve the blocks):
        Input the file number s and display the blocks allocated to the file.
        Loop back to the menu.
    If the user selects option 2 (to exit), terminate the program.
    If the user provides an invalid choice, display an error message and ask for
    input again.
Step 7: Stop
```

RESULT:

Indexed file allocation strategy is implemented successfully.

PROGRAM CODE:

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int f[50],p,a,st,len,n,c,ch;
    for(int i=0;i<50;i++)
        f[i]=0;     //unallocated
    printf("Enter the number of blocks which are already allocated: ");
    scanf("%d",&p);
    if(p!=0) {
        printf("\nEnter the blocks which are already allocated: ");
        for(int i=0;i<p;i++) {
            scanf("%d",&a);
            f[a]=1; //allocate the blocks
        }
    }

    printf("\n1. Add file \t2. Exit\n");
    scanf("%d",&ch);
    while(ch!=2) {
        if(ch==1) {
            printf("Enter the starting index block & length: ");
            scanf("%d%d",&st,&len);
            int k=len;
            for(int j=st;j<(k+st);j++) {
                if(f[j]==0) {
                    f[j]=1;
                    printf("\n%d->%d",j,f[j]);
                }
                else {
                    printf("\n%d->file is already allocated",j);
                    k++;
                }
            }
            printf("\n");
        }
        else if(ch==2) {
            exit(0);
        }
        else {
            printf("Invalid choice");
        }
        printf("\n1. Get the blocks\t2. Exit\n");
        scanf("%d",&ch);
    }
    return 0;
}
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 1.4(iii)

LINKED FILE ALLOCATION

AIM:

Simulate Linked file allocation strategies.

ALGORITHM:

```
Step 0: Start
Step 1: Input:
        The number of blocks p that are already allocated.
        The list of block numbers that are already allocated.
Step 2: Initialization:
        Declare an array f[50] to represent the file system with 50 blocks. Initialize all
        blocks to 0 (unallocated).
        If p != 0, input the block numbers that are already allocated, and set f[a] = 1
        for each allocated block a.
Step 3: Menu Options:
        Display a menu with two choices:
                Add a file.
                Exit.
Step 4: Add File:
        If the user chooses to add a file:
                Input the starting block st and the length of the file len (number of
                blocks required).
                Initialize k = len to keep track of the number of blocks left to allocate.
Step 5: Block Allocation:
        For each block j from st to st + len - 1:
                If the block f[j] == 0 (unallocated), allocate it by setting f[j] = 1 and print
                the allocation: j -> 1 (block j is now allocated).
                If the block f[j] == 1 (already allocated), print that the block is already
                allocated and increment k to attempt allocating the next block in the
                file, ensuring the required number of blocks are allocated.
Step 6: Exit Program:
        If the user chooses the exit option, terminate the program.
        If the user provides an invalid choice, display an error message and prompt
        for input again.
Step 7: Repeat:
        After processing the file allocation, display the menu again for further action.
Step 8: Stop
```

<u>OUTPUT</u>:

```
Enter the number of blocks which are already allocated: 3

Enter the blocks which are already allocated: 2 5 8

1. Add file        2. Exit
1
Enter the starting index block & length: 4 3

4->1
5->file is already allocated
6->1
7->1

1. Get the blocks        2. Exit
2
```

RESULT:

Linked file allocation strategy is implemented successfully.

EXPERIMENTS FROM SYSTEM SOFTWARE

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void passOne(char label[10], char opcode[10], char operand[10], char code[10], char
mnemonic[3]);
void display();

int main()
{
   // for reading from input
   char label[10], opcode[10], operand[10];
   // for reading from optab
   char code[10], mnemonic[3];
   // call the function
   passOne(label, opcode, operand, code, mnemonic);

   return 0;
}

void passOne(char label[10], char opcode[10], char operand[10], char code[10], char
mnemonic[3])
{
   int locctr, start, length;

   FILE *fp1, *fp2, *fp3, *fp4, *fp5;                          // file pointers

   // read mode
   fp1 = fopen("input.txt", "r");
   fp2 = fopen("optab.txt", "r");
   // write mode
   fp3 = fopen("symtab.txt", "w");
   fp4 = fopen("intermediate.txt", "w");
   fp5 = fopen("length.txt", "w");

   fscanf(fp1, "%s\t%s\t%s", label, opcode, operand);          // read first line

   if (strcmp(opcode, "START") == 0) {
      // atoi() requires stdlib.h header file , it converts ASCII to integer
      start = atoi(operand);                          // convert operand value from
string to integer and assign to start
      locctr = start;
      fprintf(fp4, "\t%s\t%s\t%s\n", label, opcode, operand);    // write to output file
(additional tab space as start will not have any locctr)
      fscanf(fp1, "%s\t%s\t%s", label, opcode, operand);       // read next line
   }
   else {
      locctr = 0;
   }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 2.1

PASS 1 OF TWO PASS ASSEMBLER

AIM:

Implement pass one of a two pass assembler.

ALGORITHM:

```
BEGIN
   READ first input line
   IF OPCODE = 'START' THEN
   BEGIN
      SAVE #[OPERAND] as starting address
      INITIALIZE LOCCTR to starting address
      WRITE line to intermediate file
      READ next input line
   END (if START)
   ELSE
      INITIALIZE LOCCTR to 0

   WHILE OPCODE != 'END' DO
   BEGIN
      IF this is not a comment line THEN
      BEGIN
         IF there is a symbol in the LABEL field THEN
         BEGIN
            SEARCH SYMTAB for LABEL
            IF found THEN
               SET error flag (duplicate symbol)
            ELSE
               INSERT (LABEL, LOCCTR) into SYMTAB
         END (if symbol)

         SEARCH OPTAB for OPCODE
         IF found THEN
            ADD 3 (instruction length) to LOCCTR
         ELSE IF OPCODE = 'WORD' THEN
            ADD 3 to LOCCTR
         ELSE IF OPCODE = 'RESW' THEN
            ADD 3 * #[OPERAND] to LOCCTR
         ELSE IF OPCODE = 'RESB' THEN
            ADD #[OPERAND] to LOCCTR
         ELSE IF OPCODE = 'BYTE' THEN
         BEGIN
```

```
//iterate till end
   while (strcmp(opcode, "END") != 0) {

      // 1. transfer address and read line to output file
      fprintf(fp4, "%d\t%s\t%s\t%s\n", locctr, label, opcode, operand);

      // 2. make symtab file with values not starting with **
      if (strcmp(label, "**") != 0) {
         fprintf(fp3, "%s\t%d\n", label, locctr);
      }

      // 3. read from optab (code and mnemonic value)
      fscanf(fp2, "%s\t%s", code, mnemonic);

      // 4. traverse till the end of optab file
      while (strcmp(code, "END") != 0) {
         if (strcmp(opcode, code) == 0) {                    // if opcode in input matches
the one in optab, increment locctr by 3
            locctr += 3;
            break;
         }
         // read next line
         fscanf(fp2, "%s\t%s", code, mnemonic);
      }

      // 5. Searching opcode for WORD, RESW, BYTE, RESB keywords and updating
locctr

            // WORD -> add 3 to locctr
      if (strcmp(opcode, "WORD") == 0) {
         locctr += 3;
      }
            // RESW -> add 3*operand to locctr
      else if (strcmp(opcode, "RESW") == 0) {
         locctr += (3 * (atoi(operand)));                    // convert operand to integer and
multiply with 3
      }
            // BYTE -> add 1 to locctr
      else if (strcmp(opcode, "BYTE") == 0) {
         ++locctr;
      }
            // RESB -> add operand to locctr
      else if (strcmp(opcode, "RESB") == 0) {
         locctr += atoi(operand);
      }

      // read next line
      fscanf(fp1, "%s\t%s\t%s", label, opcode, operand);
   }
   // 6. transfer last line to file
   fprintf(fp4, "%d\t%s\t%s\t%s\n", locctr, label, opcode, operand);

   // 7. Close all files
   fclose(fp4);
   fclose(fp3);
```

```
            FIND length of constant in bytes
            ADD length to LOCCTR
        END (if BYTE)
        ELSE
            SET error flag (invalid operation code)
    END (if not a comment)

    WRITE line to intermediate file
    READ next input line
END (while not END)

WRITE last line to intermediate file
SAVE (LOCCTR - starting address) as program length
END (Pass 1)
```

```
    fclose(fp2);
    fclose(fp1);

    // 8. display outputs
    display();

    // 9. calculate length of program
    length = locctr - start;
    fprintf(fp5, "%d", length);
    fclose(fp5);
    printf("\nThe length of the code : %d\n", length);
}

void display() {

    char str;
    FILE *fp1, *fp2, *fp3;

    // 1. Input Table
    printf("\nThe contents of Input Table :\n\n");
    fp1 = fopen("input.txt", "r");
    str = fgetc(fp1);
    while (str != EOF) {
        printf("%c", str);
        str = fgetc(fp1);
    }
    fclose(fp1);

    //2. Output Table
    printf("\n\nThe contents of Output Table :\n\n");
    fp2 = fopen("intermediate.txt", "r");
    str = fgetc(fp2);
    while (str != EOF) {
        printf("%c", str);
        str = fgetc(fp2);
    }
    fclose(fp2);

    // 3. Symtable
    printf("\n\nThe contents of Symbol Table :\n\n");
    fp3 = fopen("symtab.txt", "r");
    str = fgetc(fp3);
    while (str != EOF) {
        printf("%c", str);
        str = fgetc(fp3);
    }
    fclose(fp3);
}
```

<u>OUTPUT</u>:

input.txt :

```
        **      START   2000
        **      LDA     FIVE
        **      STA     ALPHA
        **      LDCH    CHARZ
        **      STCH    C1
        ALPHA           RESW    2
        FIVE    WORD            5
        CHARZ           BYTE    C'Z'
        C1      RESB    1
        **      END     **
```

optab.txt :

```
        LDA 03
        STA     0f
        LDCH    53
        STCH    57
        END     *
```

intermediate.txt :

```
                **      START           2000
        2000    **      LDA     FIVE
        2003    **      STA     ALPHA
        2006    **      LDCH    CHARZ
        2009    **      STCH    C1
        2012    ALPHA           RESW 2
        2018    FIVE    WORD5
        2021    CHARZ           BYTE    C'Z'
        2022    C1      RESB    1
        2023    **      END     **
```

symtab.txt :

```
        ALPHA           2012
        FIVE    2018
        CHARZ           2021
        C1      2022
```

length.txt :

```
        23
```

RESULT:

Pass 1 of one pass assembler implemented successfully.

PROGRAM CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void display();

// itoa manual implementation as its not ANSI Standard
// start of itoa block
// Function to swap two numbers
void swap(char *x, char *y)
{
    char t = *x;
    *x = *y;
    *y = t;
}

// Function to reverse `buffer[i…j]`
char *reverse(char *buffer, int i, int j)
{
    while (i < j)
    {
        swap(&buffer[i++], &buffer[j--]);
    }

    return buffer;
}

// Iterative function to implement `itoa()` function in C
char *itoa(int value, char *buffer, int base)
{
    // invalid input
    if (base < 2 || base > 32)
    {
        return buffer;
    }

    // consider the absolute value of the number
    int n = abs(value);

    int i = 0;
    while (n)
    {
        int r = n % base;

        if (r >= 10)
        {
            buffer[i++] = 65 + (r - 10);
        }
        else
        {
            buffer[i++] = 48 + r;
        }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 2.2

PASS 2 OF TWO PASS ASSEMBLER

AIM:

Implement pass two of a two pass assembler.

ALGORITHM:

```
BEGIN
  READ first input line (from intermediate file)
  IF OPCODE = 'START' THEN
  BEGIN
    WRITE listing line
    READ next input line
  END (if START)

  WRITE Header record to object program
  INITIALIZE first Text record

  WHILE OPCODE != 'END' DO
  BEGIN
    IF this is not a comment line THEN
    BEGIN
      SEARCH OPTAB for OPCODE
      IF found THEN
      BEGIN
        IF there is a symbol in the OPERAND field THEN
        BEGIN
          SEARCH SYMTAB for OPERAND
          IF found THEN
            STORE symbol value as operand address
          ELSE
          BEGIN
            STORE 0 as operand address
            SET error flag (undefined symbol)
          END
        END (if symbol)
        ELSE
          STORE 0 as operand address

        ASSEMBLE the object code instruction
      END (if opcode found)
      ELSE IF OPCODE = 'BYTE' or 'WORD' THEN
        CONVERT constant to object code
```

```
n = n / base;
    }

    // if the number is 0
    if (i == 0)
    {
        buffer[i++] = '0';
    }

    // If the base is 10 and the value is negative, the resulting string
    // is preceded with a minus sign (-)
    // With any other base, value is always considered unsigned
    if (value < 0 && base == 10)
    {
        buffer[i++] = '-';
    }

    buffer[i] = '\0'; // null terminate string

    // reverse the string and return it
    return reverse(buffer, 0, i - 1);
}
// end of itoa block

int main()
{
    char a[10], ad[10], label[10], opcode[10], operand[10], symbol[10];
    int start, diff, i, address, add, len, actual_len, finaddr, prevaddr, j = 0;
    char mnemonic[15][15] = {"LDA", "STA", "LDCH", "STCH"};
    char code[15][15] = {"33", "44", "53", "57"};

    FILE *fp1, *fp2, *fp3, *fp4;
    fp1 = fopen("output.txt", "w");
    fp2 = fopen("symtab.txt", "r");
    fp3 = fopen("intermediate.txt", "r");
    fp4 = fopen("objcode.txt", "w");

    fscanf(fp3, "%s\t%s\t%s", label, opcode, operand);

    while (strcmp(opcode, "END") != 0)
    {
        prevaddr = address;
        fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
    }
    finaddr = address;

    fclose(fp3);
    fp3 = fopen("intermediate.txt", "r");

    fscanf(fp3, "\t%s\t%s\t%s", label, opcode, operand);
    if (strcmp(opcode, "START") == 0)
    {
        fprintf(fp1, "\t%s\t%s\t%s\n", label, opcode, operand);
        fprintf(fp4, "H^%s^00%s^00%d\n", label, operand, finaddr);
        fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
```

```
         IF object code will not fit into the current Text record THEN
         BEGIN
            WRITE Text record to object program
            INITIALIZE new Text record
         END

         ADD object code to Text record
      END (if not comment)

      WRITE listing line
      READ next input line
   END (while not END)

   WRITE last Text record to object program
   WRITE End record to object program
   WRITE last listing line
END (Pass 2)
```

```
        start = address;
        diff = prevaddr - start;
        fprintf(fp4, "T^00%d^%d", address, diff);
    }

    while (strcmp(opcode, "END") != 0)
    {
        if (strcmp(opcode, "BYTE") == 0)
        {
            fprintf(fp1, "%d\t%s\t%s\t%s\t", address, label, opcode, operand);
            len = strlen(operand);
            actual_len = len - 3;
            fprintf(fp4, "^");
            for (i = 2; i < (actual_len + 2); i++)
            {
                itoa(operand[i], ad, 16);
                fprintf(fp1, "%s", ad);
                fprintf(fp4, "%s", ad);
            }
            fprintf(fp1, "\n");
        }

        else if (strcmp(opcode, "WORD") == 0)
        {
            len = strlen(operand);
            itoa(atoi(operand), a, 10);
            fprintf(fp1, "%d\t%s\t%s\t%s\t00000%s\n", address, label, opcode, operand,
a);
            fprintf(fp4, "^00000%s", a);
        }

        else if ((strcmp(opcode, "RESB") == 0) || (strcmp(opcode, "RESW") == 0))
        {
            fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
        }

        else
        {
            while (strcmp(opcode, mnemonic[j]) != 0)
                j++;
            if (strcmp(operand, "COPY") == 0)
                fprintf(fp1, "%d\t%s\t%s\t%s\t%s0000\n", address, label, opcode, operand,
code[j]);
            else
            {
                rewind(fp2);
                fscanf(fp2, "%s%d", symbol, &add);
                while (strcmp(operand, symbol) != 0)
                    fscanf(fp2, "%s%d", symbol, &add);
                fprintf(fp1, "%d\t%s\t%s\t%s\t%s%d\n", address, label, opcode, operand,
code[j], add);
                fprintf(fp4, "^%s%d", code[j], add);
            }
        }
```

```
        fscanf(fp3, "%d%s%s%s", &address, label, opcode, operand);
    }

    fprintf(fp1, "%d\t%s\t%s\t%s\n", address, label, opcode, operand);
    fprintf(fp4, "\nE^00%d", start);

    fclose(fp4);
    fclose(fp3);
    fclose(fp2);
    fclose(fp1);

    display();

    return 0;
}

void display()
{
    char ch;
    FILE *fp1, *fp2, *fp3, *fp4;

    printf("\nIntermediate file is converted into object code");

    printf("\n\nThe contents of Intermediate file:\n\n");
    fp3 = fopen("intermediate.txt", "r");
    ch = fgetc(fp3);
    while (ch != EOF)
    {
        printf("%c", ch);
        ch = fgetc(fp3);
    }
    fclose(fp3);

    printf("\n\nThe contents of Symbol Table :\n\n");
    fp2 = fopen("symtab.txt", "r");
    ch = fgetc(fp2);
    while (ch != EOF)
    {
        printf("%c", ch);
        ch = fgetc(fp2);
    }
    fclose(fp2);

    printf("\n\nThe contents of Output file :\n\n");
    fp1 = fopen("output.txt", "r");
    ch = fgetc(fp1);
    while (ch != EOF)
    {
        printf("%c", ch);
        ch = fgetc(fp1);
    }
    fclose(fp1);

    printf("\n\nThe contents of Object code file :\n\n");
    fp4 = fopen("objcode.txt", "r");
```

```
        ch = fgetc(fp4);
        while (ch != EOF)
        {
            printf("%c", ch);
            ch = fgetc(fp4);
        }
        fclose(fp4);
        printf("\n\n");
    }
```

OUTPUT:

intermediate.txt :

```
             **      START          2000
      2000   **      LDA    FIVE
      2003   **      STA    ALPHA
      2006   **      LDCH   CHARZ
      2009   **      STCH   C1
      2012   ALPHA          RESW 2
      2018   FIVE    WORD5
      2021   CHARZ          BYTE   C'Z'
      2022   C1      RESB   1
      2023   **      END    **
```

output.txt :

```
             **      START          2000
      2000   **      LDA    FIVE    332018
      2003   **      STA    ALPHA          442012
      2006   **      LDCH   CHARZ          532021
      2009   **      STCH   C1      572022
      2012   ALPHA          RESW 2
      2018   FIVE    WORD5          000005
      2021   CHARZ          BYTE   C'Z'     5A
      2022   C1      RESB   1
      2023   **      END    **
```

symtab.txt :

```
      ALPHA         2012
      FIVE   2018
      CHARZ         2021
      C1     2022
```

objcode.txt :

```
      H^***^002000^002023
      T^002000^22^332018^442012^532021^572022^000005^5A
      E^002000
```

RESULT:

Pass 2 of two pass assembler implemented successfully.

<u>PROGRAM CODE</u>:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define max 100
void swap(char *x, char *y)
{
    char t = *x;
    *x = *y;
    *y = t;
}

// Function to reverse `buffer[i…j]`
char *reverse(char *buffer, int i, int j)
{
    while (i < j)
    {
        swap(&buffer[i++], &buffer[j--]);
    }

    return buffer;
}

// Iterative function to implement `itoa()` function in C
char *itoa(int value, char *buffer, int base)
{
    // invalid input
    if (base < 2 || base > 32)
    {
        return buffer;
    }

    // consider the absolute value of the number
    int n = abs(value);

    int i = 0;
    while (n)
    {
        int r = n % base;

        if (r >= 10)
        {
            buffer[i++] = 65 + (r - 10);
        }
        else
        {
            buffer[i++] = 48 + r;
        }

        n = n / base;
    }

    // if the number is 0
```

Name: Keerthana J　　　　　　　　　　　　Date:
Roll No: 35
Class: CS5A

EXPERIMENT 2.3

MACRO PROCESSOR

AIM:

Implement a single pass macro processor.

ALGORITHM:

```
begin (macro processor)
    EXPANDING := FALSE
    while OPCODE *'END' do
        begin
        GETLINE
        PROCESSLINE
    end (while)
end (macro processor)

procedure PROCESSLINE
begin
    search NAMTAB for OPCODE
    if found then
        EXPAND
    else if OPCODE = 'MACRO' then
        DEFINE
    else write source line to expanded file
end (PROCESSLINE)

procedure DEFINE
begin
    enter macro name into NAMTAB
    enter macro prototype into DEFTAB
    LEVEL := 1
    while LEVEL > 0 do
        begin
        GETLINE
        if this is not a comment line then
            begin
            substitute positional notation for parameters
            enter line into DEFTAB
            if OPCODE = 'MACRO' then
                LEVEL := LEVEL + 1
    if (i == 0)
    {
        buffer[i++] = '0';
    }

    // If the base is 10 and the value is negative, the resulting string
    // is preceded with a minus sign (-)
    // With any other base, value is always considered unsigned
    if (value < 0 && base == 10)
    {
        buffer[i++] = '-';
    }

    buffer[i] = '\0'; // null terminate string

    // reverse the string and return it
    return reverse(buffer, 0, i - 1);
}
// end of itoa block

int main()
{
    FILE *f1, *f2, *f3, *f4, *f5;
    char label[max], opcode[max], operand[max], name[max], opcode1[max],
operand1[max], arg[max];
    int pos = 1, len;
    char pos1[max], pos2[max];
    f1 = fopen("input.txt", "r");
    f2 = fopen("namtab.txt", "w+");
    f3 = fopen("deftab.txt", "w+");
    f4 = fopen("argtab.txt", "w+");
    f5 = fopen("output.txt", "w+");
    fscanf(f1, "%s%s%s", label, opcode, operand);
    while (strcmp(opcode, "END") != 0)
    {
        if (strcmp(opcode, "MACRO") == 0)
        {
            fprintf(f2, "%s", label);
            fseek(f2, SEEK_SET, 0);
            fprintf(f3, "%s\t%s\n", label, operand);
            fscanf(f1, "%s%s%s", label, opcode, operand);
            while (strcmp(opcode, "MEND") != 0)
            {
                if (operand[0] == '&')
                {
                    itoa(pos, pos1, 5);
                    strcpy(pos2, "?");
                    strcpy(operand, strcat(pos2, pos1));
                    pos = pos + 1;
                }
                fprintf(f3, "%s\t%s\n", opcode, operand);
                fscanf(f1, "%s%s%s", label, opcode, operand);
            }
            fprintf(f3, "%s", opcode);
        }
```

```
                    else if OPCODE = 'MEND' then
                          LEVEL := LEVEL - I
              end (if not comment)
        end (while)
store in NAMTAB pointers to beginning and end of definition
end (DEFINE)


PROCEDURE EXPAND
BEGIN
  EXPANDING  := TRUE
  get first line of macro definition (prototype) from DEFTAB
  set up arguments from macro invocation in ARGTAB
  write macro invocation to expanded file as a comment
  WHILE not end of macro definition DO
  BEGIN
    GETLINE
    PROCESSLINE
  END (while)
  EXPANDING := FALSE
END (EXPAND)


PROCEDURE GETLINE
BEGIN
  IF EXPANDING THEN
  BEGIN
    get next line of macro definition from DEFTAB
    substitute arguments from ARGTAB for positional notation
  END (if)
  ELSE
    read next line from input file
END (GETLINE)
```

```
        else
        {
            fscanf(f2, "%s", name);
            if (strcmp(name, opcode) == 0)
            {
                len = strlen(operand);
                for (int i = 0; i < len; i++)
                {
                    if (operand[i] != ',')
                    {
                        fprintf(f4, "%c", operand[i]);
                    }
                    else
                    {
                        fprintf(f4, "\n");
                    }
                }
                fseek(f3, SEEK_SET, 0);
                fseek(f4, SEEK_SET, 0);
                fscanf(f3, "%s%s", opcode1, operand1);
                fprintf(f5, ".\t%s\t%s\n", opcode1, operand1);
                fscanf(f3, "%s%s", opcode1, operand1);
                while (strcmp(opcode1, "MEND") != 0)
                {
                    if (operand1[0] == '?')
                    {
                        fscanf(f4, "%s", arg);
                        fprintf(f5, "-\t%s\t%s", opcode1, arg);
                    }
                    else
                    {
                        fprintf(f5, "-\t%s\t%s", opcode1, operand1);
                    }
                    fscanf(f3, "%s%s", opcode1, operand1);
                }
            }
            else
            {
                fprintf(f5, "%s\t%s\t%s\n", label, opcode, operand);
            }
        }
        fscanf(f1, "%s%s%s", label, opcode, operand);
    }
    fprintf(f5, "%s\t%s\t%s\n", label, opcode, operand);
}
```

<u>OUTPUT</u>:

input.txt :

```
        PGM START 0
        ABC MACRO &A,&B
        -   STA &A
        -   STB &B
        -   MEND    -
        -   ABC    P,Q
        -   ABC    R,S
        P   RESW    1
        Q   RESW    1
        R   RESW    1
        S   RESW    1
        -   END –
```

namtab.txt :

```
        ABC
```

argtab.txt :

```
        P
        QR
        S
```

deftab.txt :

```
        ABC     &A,&B
        STA     ?1
        STB     ?2
        MEND
```

output.txt :

```
        PGM     START           0
        .       ABC     &A,&B
        -       STA     P-      STB     Q.      ABC     &A,&B
        -       STA     P-      STB     QRP     RESW 1
        Q       RESW 1
        R       RESW 1
        S       RESW 1
        -       END     -
```

RESULT:

Single pass macro processor implemented successfully.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main()
{
    FILE *fp, *fp1;
    int i, addr1, l, j, staddr1;
    char name[10], line[50], name1[10], addr[10], rec[10], ch, staddr[10];
    printf("Enter Program Name : ");
    scanf("%s", name);
    fp = fopen("input.txt", "r");
    fp1 = fopen("output.txt", "w");
    fscanf(fp, "%s", line);
    for (i = 2, j = 0; i < 8, j < 6; i++, j++)
        name1[j] = line[i];
    name1[j] = '\0';
    printf("Name from Input File : %s\n", name1);
    if (strcmp(name, name1) == 0)
    {
        do
        {
            fscanf(fp, "%s", line);
            if (line[0] == 'T')
            {
                for (i = 2, j = 0; i < 8, j < 6; i++, j++)
                    staddr[j] = line[i];
                staddr[j] = '\0';
                staddr1 = atoi(staddr);
                i = 12;
                while (line[i] != '$')
                {
                    if (line[i] != '^')
                    {
                        printf("00%d \t %c%c\n",
                               staddr1, line[i], line[i + 1]);
                        fprintf(fp1, "00%d \t %c%c\n",
                                staddr1, line[i], line[i + 1]);
                        staddr1++;
                        i = i + 2;
                    }
                    else
                        i++;
                }
            }
            else if (line[0] = 'E')
            {
                fclose(fp);
                exit(0);
            }
        } while (!feof(fp));
    }
}
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 2.4

ABSOLUTE LOADER

AIM:

Implement an absolute loader.

ALGORITHM:

```
BEGIN
    read Header record
    verify program name and length
    read first Text record
    while record type != 'E' do
        begin
            (if object code is in character form, convert into internal
            representation)
            move object code to specified location in memory
            read next object program record
        end
    jump to address specified in End record
END
```

<u>OUTPUT</u>:

input.txt :

```
        H^SAMPLE^001000^0035
        T^001000^0C^001003^071009$
        T^002000^03^111111$
        E^001000
```

output.txt :

```
        001000          00
        001001          10
        001002          03
        001003          07
        001004          10
        001005          09
        002000          11
        002001          11
        002002          11
```

RESULT:

Absolute loader implemented successfully.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void convert(char h[12]);
char bitmask[12];
char bit[12] = {0};
void main()
{
    char add[6], length[10], input[10], binary[12], relocbit, ch, pn[5];
    int start, inp, len, i, address, opcode, addr, actualadd, tlen;
    FILE *fp1, *fp2;
    printf("\n\n Enter the actual starting address : ");
    scanf("%x", &start);
    fp1 = fopen("input.txt", "r");
    fp2 = fopen("output.txt", "w");
    fscanf(fp1, "%s", input);
    fprintf(fp2, " ----------------------------\n");
    fprintf(fp2, " ADDRESS\tCONTENT\n");
    fprintf(fp2, " ----------------------------\n");
    while (strcmp(input, "E") != 0)
    {
        if (strcmp(input, "H") == 0)
        {
            fscanf(fp1, "%s", pn);
            fscanf(fp1, "%x", add);
            fscanf(fp1, "%x", length);
            fscanf(fp1, "%s", input);
        }
        if (strcmp(input, "T") == 0)
        {
            fscanf(fp1, "%x", &address);
            fscanf(fp1, "%x", &tlen);
            fscanf(fp1, "%s", bitmask);
            address += start;
            convert(bitmask);
            len = strlen(bit);
            if (len >= 11)
                len = 10;
            for (i = 0; i < len; i++)
            {
                fscanf(fp1, "%x", &opcode);
                fscanf(fp1, "%x", &addr);
                relocbit = bit[i];
                if (relocbit == '0')
                    actualadd = addr;
                else
                    actualadd = addr + start;
                fprintf(fp2, "\n  %x\t\t%x%x\n", address, opcode, actualadd);
                address += 3;
            }
            fscanf(fp1, "%s", input);
        }
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 2.5

RELOCATING LOADER

AIM:

Implement a relocating loader.

ALGORITHM:

```
START
   Prompt user to enter the actual starting address
   Read starting address (start) from user
   Open "input.txt" in read mode as fp1
   Open "output.txt" in write mode as fp2
   Write header information to output file
   READ first line of input from fp1 into variable 'input'

   WHILE 'input' is not "E" (End record) DO
      IF 'input' is "H" (Header record) THEN
         READ program name (pn), starting address (add), and program length
(length) from fp1
         READ next line of input from fp1 into 'input'
      END IF

      IF 'input' is "T" (Text record) THEN
         READ address (address) and text length (tlen) from fp1
         READ bitmask (bitmask) from fp1
         Adjust 'address' by adding 'start' to it

         CONVERT bitmask to binary format and store in 'bit'
         SET length of 'bit' to len (limit to 10 if greater than 10)

         FOR i = 0 to len - 1 DO
            READ opcode and address from fp1
            SET relocbit = bit[i]

            IF relocbit is '0' THEN
               actualadd = address (no relocation needed)
            ELSE
               actualadd = address + start (relocation needed)

            WRITE address, opcode, and actualadd to fp2
            Increment address by 3 (for next instruction slot)
         END FOR
```

```
    }
    fprintf(fp2, " ----------------------------\n");
    fclose(fp1);
    fclose(fp2);
    printf("\n\n The contents of output file(output.txt) \n\n");
    fp2 = fopen("output.txt", "r");
    ch = fgetc(fp2);
    while (ch != EOF)
    {
        printf("%c", ch);
        ch = fgetc(fp2);
    }
    fclose(fp2);
}
void convert(char h[12])
{
    int i, l;
    strcpy(bit, "");
    l = strlen(h);
    for (i = 0; i < l; i++)
    {
        switch (h[i])
        {
        case '0':
            strcat(bit, "0");
            break;
        case '1':
            strcat(bit, "1");
            break;
        case '2':
            strcat(bit, "10");
            break;
        case '3':
            strcat(bit, "11");
            break;
        case '4':
            strcat(bit, "100");
            break;
        case '5':
            strcat(bit, "101");
            break;
        case '6':
            strcat(bit, "110");
            break;
        case '7':
            strcat(bit, "111");
            break;
        case '8':
            strcat(bit, "1000");
            break;
        case '9':
            strcat(bit, "1001");
            break;
        case 'A':
            strcat(bit, "1010");
            READ next line from fp1 into 'input'
        END IF
    END WHILE

    Write footer line to output file
    Close files fp1 and fp2
    Print contents of "output.txt" to console:
        Open "output.txt" in read mode as fp2
        WHILE not end of file (EOF) in fp2
            Read each character from fp2 and print to console
        END WHILE
        Close fp2
END


PROCEDURE CONVERT(h)
    Initialize 'bit' as an empty string
    FOR each character in 'h'
        Convert hexadecimal character to its binary equivalent and append to 'bit'
    END FOR
END PROCEDURE
```

```
                    break;
                case 'B':
                    strcat(bit, "1011");
                    break;
                case 'C':
                    strcat(bit, "1100");
                    break;
                case 'D':
                    strcat(bit, "1101");
                    break;
                case 'E':
                    strcat(bit, "1110");
                    break;
                case 'F':
                    strcat(bit, "1111");
                    break;
                }
            }
        }
```

OUTPUT:

input.txt :

```
H COPY 000000 00107A
T 000000 1E FFC 14 0033 48 1039 10 0036 28 0030 30 0015 48 1061 3C 0003 20
002A 1C 0039 30 002D
T 002500 15 E00 1D 0036 48 1061 18 0033 4C 1000 80 1000 60 1003
E 000000
```

output.txt :

```
----------------------------
 ADDRESS    CONTENT
 ----------------------------

  4000          144033

  4003          485039

  4006          104036

  4009          284030

  400c          304015

  400f          485061

  4012          3c4003

  4015          20402a

  4018          1c4039
```

```
 401b           30402d
 6500           1d4036
 6503           485061
 6506           184033
 6509           4c1000
 650c           801000
 650f           601003
-----------------------------
```

<u>RESULT</u>:

Relocating loader implemented successfully.

EXPERIMENTS FROM MICROPROCESSORS AND MICROCONTROLLERS

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.1

STUDY OF ASSEMBLER AND DEBUGGING COMMANDS

AIM:

To study about the assembler and different debugging commands.

THEORY:

A program called 'assembler' is used to convert an assembly input file (source file) to an object file that is further converted to machine code using linker. The chances of error being committed are less (than hand coding) as mnemonics are used instead of opcodes.

MASM (Microsoft Macro Assembler) is one of the popular assemblers used along with the LINK program to structure the code generated by MASM in the form of an executable file. LINK accepts object code generated by MASM as input and produces an EXE file.

DEBUG.COM is a DOS utility that facilitates debugging and trouble-shooting of assembly language programs. DEBUG enables you to have control on resources of personal computers to an extent. The program DEBUG may be used to debug a source program or to observe the result of execution of an .EXE file with the help of .LST. DEBUG is able to troubleshoot only.EXE files. DEBUG offers a good platform for troubleshooting, executing and observing results of assembly language programs.

RESULT:

Successfully studied about assembler and different debugging commands.

PROGRAM CODE:

```
2000    MOV    S1, 3000
2003    MOV    D1, 4000
2006    MOV    AX, [SI]
2008    INC    SI
2009    INC    SI
200A    MOV    BX, [SI]
200C    ADD    AX, BX
200E    JNC    2015
2010    MOV    [DI], 0001
2013    JMP    2018
2015    MOV    [DI], 0000
2018    INC    DI
2019    MOV    [DI],AX
2018    HLT
```

INPUT:

Adress	Data
3003:	F0
3002:	02
3001:	F0
3000:	02

OUTPUT:

Adress	Data
4003:	00
4002:	01
4001:	E0
4000:	04

Name: Keerthana J   Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.2(i)

ADDITION OF 16 BIT NUMBERS

AIM:

Implementation of addition of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

1. Start the program
2. Set the Source Index (SI) register to memory address 3000
3. Set the Destination Index (DI) register to point to memory address 4000
4. Load the value stored at the memory address pointed to by SI into the AX register
5. Increment the SI register by 2 bytes (assuming SI is pointing to a word or 2 bytes)
6. Load the value stored at the updated memory address pointed to by SI into the BX register
7. Add the values stored in AX and BX registers and store the result in the AX register.
8. Check if there was no carry generated during the addition
    1. If no carry occurred, store the value 0001 at the memory address pointed to by DI.
    2. If a carry occurred, store the value 0000 at the memory address pointed to by DI.
9. Increment the DI register to point to the next memory address.
10. Store the contents of the AX register at the memory address pointed to by DI.
11. Halt the execution of the program

RESULT:

Executed addition of 16 bit numbers using kit.

PROGRAM CODE:

```
2000   MOV   S1, 3000
2003   MOV   D1, 4000
2006   MOV   AX, [SI]
2008   INC   SI
2009   INC   SI
200A   MOV   BX,[S0]
200C   CMP   AX,BX
200E   JC    2017
2010   SUB   AX,BX
2012   MOV   [DI],0000
2015   JMP   2022
2017   MOV   CX,AX
2019   MOV   AX,BX
201B   MOV   BX,CX
```

INPUT:

Adress	Data
3003:	00
3002:	60
3001:	00
3000:	50

OUTPUT:

Adress	Data
4003:	00
4002:	01
4001:	00
4000:	00

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.2(ii)

SUBTRACTION OF 16 BIT NUMBERS

AIM:

Implementation of subtraction of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

```
1.Start the program
2.Set SI register to the memory address 3000
3.Set DI register to the memory address 4000
4.Load the value at the memory address stored in SI into the AX
5.Increment the SI register by 2 (assuming SI points to a word or 2 bytes)
6.Load the value at the updated memory address stored in SI into the BX register
7.Compare the values in AX and BX.
    1. If AX is less than BX, jump to memory address 2017.
    2. Otherwise, continue to the next instruction.
8.Subtract BX from AX.
9.Store the value 0000 at the memory address stored in DI.
10.Jump to memory address 2022.
11.If AX was less than BX:
    1.Move the value in AX to the CX register
    2.Move the value in BX to AX.
    3.Move the value in CX to BX.
    4.Subtract BX from AX
    5.Store the value 0001 at the memory address stored in DI
12. Increment the DI register by 1
13.Store the value in AX at the memory address stored in DI
14.Halt the program
```

RESULT:

Executed subtraction of 16 bit numbers using kit.

PROGRAM CODE:

```
2000  MOV   SI,3000
2003  MOV   DI,4000
2006  MOV   AX, [SI]
2008  INC   SI
2009  INC   SI
200A  MOV   BX, [SI]
200C  MUL   BX
200E  MOV   [DI], AX
2010  INC   DI
2011  INC   DI
2012  MOV   [DI], DX
2014  HLT
```

INPUT:

Adress	Data
3003:	07
3002:	08
3001:	04
3000:	03

OUTPUT:

Adress	Data
4003:	00
4002:	1C
4001:	35
4000:	18

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.2(iii)

MULTIPLICATION OF 16 BIT NUMBERS

AIM:

Implementation of multiplication of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

1.Start the program
2. Set the source index register SI to point to memory address 3000
3.Set the destination index register DI to point to memory address 4000
4.Load the value at the memory address pointed by SI into the AX register
5.Increment SI by 2 (assuming SI points to word or 2 bytes in memory)
6.Load the value at the updated memory address pointed by SI into the BX register
7.Multiply the value in AX by the value in BX, storing the result in AX (product) and DX (high-order bits of the product)
8.Store the value in AX at the memory address pointed by DI
9.Increment DI by 2 (assuming DI points to a word or 2 bytes in memory)
10.Store the value in DX at the updated memory address pointed by DI
11.Halt the program

RESULT:

Assembly program to perform multiplication of 16-bit numbers have been executed successfully.

PROGRAM CODE:

```
2000   MOV   SI,3000
2003   MOV   DI,4000
2006   MOV   AX, [SI]
2008   INC   SI
2009   INC   SI
200A   MOV   BX, [SI]
200C   DIV   BX
200E   MOV   [DI], AX
2010   INC   DI
2011   INC   DI
2012   MOV   [DI], DX
2014   HLT
```

INPUT:

Adress	Data
3002:	24
3001:	CF
3000:	2D

OUTPUT:

Adress	Data
4000:	D1

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.2(iv)

DIVISION OF 16 BIT NUMBERS

AIM:

Implementation of division of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

1.Start the program
2. Set the source index register SI to point to memory address 3000
3.Set the destination index register DI to point to memory address 4000
4.Load the value at the memory address pointed by SI into the AX register
5.Increment SI by 2 (assuming SI points to word or 2 bytes in memory)
6.Load the value at the updated memory address pointed by SI into the BX register
7.Divide the value in AX by the value in BX, storing the quotient in AX and the remainder in DX
8.Store the value in AX at the memory address pointed by DI
9.Increment DI by 2 (assuming DI points to a word or 2 bytes in memory)
10.Store the value in DX at the updated memory address pointed by DI
11.Halt the program

RESULT:

Assembly programs to perform division of 16-bit numbers have been executed successfully.

<u>PROGRAM CODE</u>:

```
2000   MOV   SI,4000
2003   MOV   BX,(SI]
2005   DEC   BX
2007   MOV   SI,4000
200A   MOV   CX,[SI]
200C   DEC   CX
200E   INC   SI
2010   MOV   AX,[SI]
2012   INC   SI
2013   INC   SI
2814   CMP   AX,[SI]
2016   JC    201E
2018   XCHG  AX,(SI]
201A   DEC   SI
201B   DEC   SI
201C   XCHG  AX,[SI]
201E   INC   SI
201F   INC   SI
2020   DEC   CX
2022   JNZ   2010
2024   DEC   BX
2026   JNZ   2007
2028   HLT
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.3(i)

SORTING OF 16 BIT NUMBERS

AIM:

Implementation of searching and sorting of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

1.Initialize registers: SI, BX, CX, AX.
2. Load SI with the address 4000
3. Load BX with the value at the memory address pointed by SI ([SI])
4. Decrement the value in BX
5. Store the decremented value back in the memory address pointed by SI ((SI)).
6.Load SI with the address 4000
7.Load CX with the value at the memory address pointed by SI ([SI]).
8. Decrement the value in CX.
9. Increment the value in SI.
10. Load AX with the value at the memory address pointed by SI ([SI])
11. Increment SI twice to point to the next memory address.
12. Compare AX with the value at the memory address pointed by SI ([SI]).
13. If AX is less than the value at the memory address. jump to label 201E
14. Exchange the values of AX and the memory address pointed by SI.
15. Decrement SI twice
16. Exchange the values of AX and the memory address pointed by SI.
17. Increment SI twice.
18. Increment SI twice
19. Decrement CX.
20. If CX is not zero, jump to label 2010
21. Decrement BX.
22. If BX is not zero, jump to label 2007
23. Halt the execution

INPUT:

Adress	Value
4000:	05
4001:	10
4002:	15
4003:	23
4004:	22
4005:	40
4006:	80
4007:	00
4008:	30
4009:	00
400A:	10

OUTPUT:

Adress	Value
4001:	00
4002:	10
4003:	10
4004:	15
4005:	23
4006:	22
4007:	00
4008:	30
4009:	40
400A:	80

<u>RESULT</u>:

Assembly programs to perform sorting of 16-bit numbers have been executed successfully.

PROGRAM CODE:

```
0400   MOV   SI, 2000
0403   MOV   DI, 4000
0406   MOV   CX, 0005
0409   MOV   BX, [SI]
040B   INC   SI
040C   INC   SI
040D   MOV   AX, [SI]
040F   INC   SI
0410   INC   SI
0411   DEC   CX
0412   CMP   AX, BX
0414   JE    0421
0416   CMP   CX,0000
041A   JG    040D
041C   MOV   [DI], FF
041F   JMP   0424
0421   MOV   [DI], 0A
0424   HLT
```

INPUT:

Address	Value
2000:	34
2001:	12
2002:	12
2003:	00
2004:	13
2005:	00
2006:	34
2007:	12
2008:	28
2009:	00
200A:	34
200B:	00

OUTPUT:

Address	Value
4000:	0A

Name: Keerthana J   Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.3(ii)

SEARCHING OF 16 BIT NUMBERS

AIM:

Implementation of searching and sorting of 16-bit numbers using 8086 trainer kit.

ALGORITHM:

```
1. Set SI to the starting address of the list (2000)
2. Set DI to the address where the result will be stored (4000)
3. Set CX to the number of elements to search (5 elements)
4. While CX > 0, repeat steps 5 to 10
5. Load the next two-byte number from the address pointed by SI
      1. Load the current 16-bit number from [SI] into BX
      2. Increment SI to point to the next element
6. Load the following two-byte number for comparison
      1. Load the next 16-bit number from [SI] into AX
      2. Increment SI to point to the next element
7. Decrement CX to track the remaining comparisons
8. Compare AX and BX
9. If AX = BX then
      1. Store 0A at [DI] (match found)
      2. Jump to End of Program (0424)
10. If CX > 0 then, jump back to the comparison loop (040D)
    Else, store FF at [DI] to indicate no match
11. Halt the program
```

RESULT:

Assembly programs to perform searching of 16-bit numbers have been executed successfully.

PROGRAM CODE:

```
data SEGMENT
        MSG1 DB 10,13,'ENTER THE STRING:$’
        MSG2 DB 10,13, ‘STRING IS PALINDROME$’
        MSG3 DB 10,13, 'STRING IS NOT PALINDROME$’
        STR1 DB 50 DUP(0)
data ENDS

code SEGMENT
        ASSUME CS:code, DS:data
        START:
                MOV AX, DATA
                MOV DS, AX
                LEA DX, MSG1
                MOV AH, 09H
                INT 21H
                LEA SI, STR1
                LEA DI, STR1
                MOV AH, 01H

        NEXT:
                INT 21H
                CMP AL, ODH
                JE TERMINATE
                MOV [DI], AL
                INC DI
                JMP NEXT

        TERMINATE:
                MOV AL, '$'
                MOV [DI], AL

        DOTHIS:
                DEC DI
                MOV AL, [SI]
                CMP (DI], AL
                JNE NOTPALINDROME
                INC SI
                CMP SI, DI
                JL DOTHIS

        PALINDROME:
                MOV AH, 09H
                LEA DX, MSG2
                INT 21H
                JMP XX

        NOTPALINDROME:
                MOV AH, 09H
                LEA DX, MSG1
                INT 21H
```

Name: Keerthana J　　　　　　　　　　Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.4(i)

STRING MANIPULATION - PALINDROME

AIM:

To write an assembly program to check whether a string is palindrome or not using 8086 emulator.

ALGORITHM:

1.Define the DATA segment:
    1. MSG1: Message prompting the user to enter a string
    2. MSG2: Message indicating that the entered string is a palindrome
    3. MSG3: Message indicating that the entered string is not a palindrome.
    4. STR1: Buffer to store the input string, initialized with 50 bytes of zeros.
2. Define the CODE segment and assume segment associations (CS:code , DS:data).
3. Define a label START:
    1. Move the address of the DATA segment into AX.
    2. Move the value in AX to the DS register.
    3. Load the effective address of MSG1 into DX.
    4. Set AH to 09H (print string function)
    5. Trigger interrupt 21H (INT 21H) to display MSG1 to prompt the user for input
    6. Load the effective address of STR1 into SI and DI.
    7. Set AH to 01H (input character function).
4. Start a loop labeled NEXT:
    1. Trigger interrupt 21H to input a character from the user
    2. Compare the input character with carriage return (0DH).
    3. If it's a carriage return, jump to label TERMINATE.
    4. Store the input character at the memory location pointed by DI.
    Increment DI and continue the loop NEXT.
5. When a carriage return is encountered (label TERMINATE):
    1. Store '$' (string terminator) at the memory location pointed by DI.
6. Start a loop labeled DOTHIS:
    1. Decrement DI to point to the end of the entered string
    2. Load a character from the beginning of the string using SI into AL
    3. Compare the character at DI with AL
    4. If they are not equal, Jump to label NOTPALINDROME.
    5. Increment SI, compare SI with DI, and 1f SI is less than DI, continue the loop
7. If the string is a palindrome (1abel PALINDROME):
    1. Display MSG2 indicating that the entered string is a palindrome

```
        XX:
                MOV AH, 4CH
                INT 21H
code ENDS
END START
```

OUTPUT:

```
ENTER THE STRING: ABCD
STRING IS NOT PALINDROME

ENTER THE STRING: ABBA
STRING IS PALINDROME
```

    2. Jump to label XX.
8. If the string is not a palindrome (label NOTPALINDROME):
    1. Display MSG3 indicating that the entered string is not a palindrome.
9. Label XX:
    1. Set AH to 4CH (exit program function)
    2. Trigger interrupt 21H (INT 21H) to terminate the program.

<u>RESULT</u>:

Assembly program to check whether the input string is palindrome or not has been successfully executed using 8086 emulator

<u>PROGRAM CODE</u>:

```
data SEGMENT
        MSG1 DB 10,13,'ENTER THE STRING:$'
        STR1 DB 50 DUP(0)
        MSG2 DB 10,13,'REVERSED STRING:$'
data ENDS

code SEGMENT
        ASSUME CS:code, DS:data
        START:
                MOV AX, DATA
                MOV DS, AX
                LEA DX, MSG1
                MOV AH, 09H
                INT 21H
                LEA SI, STR1
                LEA DI, STR1
                MOV AH, 01H

        NEXT:
                INT 21H
                CMP AL, ODH
                JE TERMINATE
                MOV [DI], AL
                INC DI
                JMP NEXT

        TERMINATE:
                MOV BYTE PTR[DI],'$'
                DEC DI

        REVERSE:
                CMP SI, DI
                JAE PRINT
                MOV AL, [SI]
                MOV BL, [DI]
                MOD [DI], AL
                MOV [SI], BL
                INC SI
                DEC DI
                JMP REVERSE

        PRINT:
                LEA DX, MSG2
                MOV AH, 09H
                INT 21H
                LEA DX, STR1
                MOV AH, 09H
                INT 21H
                MOV AH, 4CH
                INT 21H
code ENDS
END START
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.4(i)

STRING MANIPULATION - REVERSE

AIM:

To write an assembly program to find the reverse of a string using 8086 emulator.

ALGORITHM:

```
1.Define the DATA segment:
        1. MSG1: Message prompting the user to enter a string
        2. STR1: Buffer to store the input string, initialized with 50 bytes of zeros.
        3. MSG2: Message to print reversed string
2. Define the CODE segment and assume segment associations (CS:code ,
   DS:data).
3. Define a label START:
        1. Move the address of the DATA segment into AX.
        2. Move the value in AX to the DS register.
        3. Load the effective address of MSG1 into DX.
        4. Set AH to 09H (print string function)
        5. Trigger interrupt 21H (INT 21H) to display MSG1 to prompt the user for
           input
        6. Load the effective address of STR1 into SI and DI.
        7. Set AH to 01H (input character function).
4. Start a loop labeled NEXT:
        1. Trigger interrupt 21H to input a character from the user
        2. Compare the input character with carriage return (0DH).
        3. If it's a carriage return, jump to label TERMINATE.
        4. Store the input character at the memory location pointed by DI.
        Increment DI and continue the loop NEXT.
5.  When a carriage return is encountered (label TERMINATE):
        1. Store '$' (string terminator) at the memory location pointed by DI.
6. Compare pointers SI and DI .
    If SI >= DI, string is fully reversed, go to PRINT
7. Load character at SI into AL.
   Load character at DI into BL.
8. Move character from AL to position DI
   Move character from BL to position SI
9. Move SI forward and move DI backward
10. Continue reversing
11. To display reversed string
        1. Load address of MSG2 into DX
        2. Set AH for display string function
        3. Call interrupt to display "REVERSED STRING:"
        4. Load address of reversed string into DX
        5. Set AH for display string function
```

OUTPUT:

```
ENTER THE STRING: PROGRAM
REVERSED STRING: MARGORP
```

        6. Call interrupt to display reversed string
    12. Exit the program
        1. Set AH for program termination
        2. Call interrupt to terminate program

RESULT:

Assembly program to print the reverse of a string has been successfully executed using 8086 emulator.

PROGRAM CODE:

```
0400:  MOV      AL,80
0402:  OUT      46, AL
0404:  MOV      CL, 04
0406:  MOV      BX,0500
0409:  MOV      AL, [BX]
040B:  OUT      40,AL
040D:  CALL     0415
0410:  INC      BX
0411:  LOOPNZ   0409
0413:  JMP      0404
0415:  PUSH     CX
0416:  MOV      CX, FFFF
0419:  LOOPNZ   0419
041B:  POP      CX
041C:  RET
```

INPUT:

Memory :

```
0500:  0A
0501:  06
0502:  05
0503:  09
0500:  09
0501:  05
0502:  06
0503:  0A
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.5

STEPPER MOTOR

AIM:

Interfacing a stepper motor with 8086 which rotates through any given sequence.

ALGORITHM:

1.Set AL register to 80 (MOV AL, 80)
2.Output the value in AL to port 46 (OUT 46, AL)
3. Set CL register to 04 (MOV CL, 04)
4. Set BX register to the memory location 0500 (MOV BX, 0500)
5. Load the value from the memory location pointed by BX into AL (MOV AL, [BX])
6. Output the value in AL to port 40 (OUT 40, AL)
7. Call the subroutine at address 0415 (CALL 0415)
8. Increment the value in BX by 1 (INC BX)
9. Loop back to 5 if the zero flag is not set (LOOPNZ 0409)
10. Jump to the instruction at address 0404 (JMP 0404)
11. Subroutine at address 0415:
    a. Push the value in CX onto the stack (PUSH CX)
    b. Set CX register to FFFF (MOV CX, FFFF)
    c. Loop back to step b if the zero flag is not set (LOOPNZ 0419)
    d. Pop the value from the stack into CX (POP CX)
    e. Return from the subroutine (RET)

RESULT:

The program was executed and output was verified.

PROGRAM CODE:

```
0400:  MOV        AL, 00
0402:  OUT        22, AL
0404:  MOV        AL, 2D
0406:  OUT        22, AL
0408:  MOV        AL, 90
040A:  OUT        22, AL
040C:  MOV        BX, 041E
040F:  MOV        SI, 0000
0412:  MOV        CX, 0007
0415:  MOV        AL, [BX+SI]
0419:  OUT        20, AL
0419:  INC        SI
041A:  LOOPNZ     0415
041C:  HLT
```

INPUT:

```
401E:  77
401F:  7F
4020:  39
4021:  3F
4022:  79
4023:  71
4024:  7D
```

Name: Keerthana J Date:
Roll No: 35
Class: CS5A

EXPERIMENT 3.6

INTERFACING WITH 8257

AIM:

Interfacing 8086 with 8257 (Static display implementation).

ALGORITHM:

```
1.Set the value of register AL to 00
2.Output the value in register AL to the I/0 port 22.
3.Set the value of register AL to 2D
4.Output the value in register AL to the I/0 port 22.
5.Set the value of register AL to 90.
6.Output the value in register AL to the I/0 port 22.
7. Load the memory address 041E into register BX.
8. Set the value of register SI to 0000.
9. Set the value of register CX to 0007
10.Enter a loop :
    a. Load the byte at the memory address (BX + SI) into register AL.
    b. Output the value in register AL to the I/0 port 20.
    c. Increment the value in register SI
    d. Decrement the value in register CX.
    e. Repeat the loop if CX is not zero
11.Halt the processor.
```

RESULT:

The program was executed and output was verified.