**LAB MANUAL**

**of**

# OPERATING SYSTEM AND ASSEMBLY LANGUAGE PROGRAMMING LAB

**B. Tech II Year II Semester**

**R22 Regulation**


## Department of Computer Science and Engineering

## CVR COLLEGE OF ENGINEERING

(An UGC Autonomous Institution, Affiliated to JNTUH, Accredited by NBA, and NAAC)
Vastunagar, Mangalpalli (V), Ibrahimpatnam (M),
Ranga Reddy (Dist.) - 501510, Telangana State

# CVR COLLEGE OF ENGINEERING

(An UGC Autonomous Institution, Affiliated to JNTUH,
Accredited by NBA, and NAAC)
Vastunagar, Mangalpalli (V), Ibrahimpatnam (M),
Ranga Reddy (Dist.) - 501510, Telangana State.

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

### VISION

Towards a Global Knowledge Hub, striving continuously in pursuit of excellence in Education, Research, consultancy, and Technological services to society.

### MISSION

**M1:** To produce the best quality Computer Science professionals by imparting quality training, hands-on experience, and value education.

**M2:** To strengthen links with industry through partnerships and collaborative developmental works.

**M3:** To attain self-sustainability and overall development through Research, Consultancy, and Development activities.

**M4:** To extend technical expertise to other technical institutions of the region and play a lead role in imparting technical education.

**M5:** To inculcate work ethics and commitment in students for their future endeavors to servethe society.

Code: 22CS283

**OPERATING SYSTEM AND ASSEMBLY LANGUAGE PROGRAMMING LAB**

| | |
|---|---|
| Instruction: 2 Periods/week | Continuous Internal Evaluation: 40 Marks |
| Tutorial: - | Semester End Examination: 60 Marks |
| Credits:1 | Semester End Exam Duration: 3 Hours |

**Course Objectives:**

1. To provide an understanding of the design aspects of operating system concepts through simulation.
2. To introduce system call interface for process management, inter-process communication and I/O in Unix.
3. To enable students gain hands on experience on Assembly Language Programming on 8086.

**List of Experiments:**

1. Implement shell commands such as cp, ls, chmod, ls -ls using the I/O system calls of UNIX/LINUX operating system.(open, read, write, close, fcntl, seek, stat, opendir, readdir)
2. Write C programs to simulate the following CPU Scheduling algorithms:
   a) FCFS b) SJF.
3. Write C programs to simulate the following CPU Scheduling algorithms:
   a) Round Robin b) priority.
4. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.
5. Write C programs to illustrate the following IPC mechanisms.
   a) Pipes b) FIFOs
6. Write C program to illustrate the Message Queues IPC mechanism.
7. Write C program to illustrate the Shared Memory IPC mechanism.
8. Write a C program to implement the Producer–Consumer problem using semaphores using UNIX/LINUX system calls (pthread library API is optional).

9. Write an ALP in 8086 add, subtract and multiply two 16-bit unsigned numbers.
10. Write an ALP in 8086 to implement ASCII Adjust and decimal adjust instructions.
11. Write an ALP to pack two digits into a Byte.
12. Write an ALP to Count number of 1's and number of 0's present in the binary representation of a given number.
13. Implement the following string manipulation functions using appropriate registers.

    a) Copy a string b) Lower to upper case c) Reverse a string d) Palindrome.Write an ALP to Count no of even and odd numbers from the given array of numbers.
14. Write a program to check whether a given number is Positive or Negative number.
15. Write an ALP to sort the given array of numbers.
16. Write C programs to simulate Paging memory management techniques.
17. Write C programs to simulate Segmentation memory management techniques.

**Course Outcomes:** At the end of the course, the student should be able to

**CO1:** Write programs using I/O System calls for implementing file operations.

**CO2:** Simulate and implement operating system concepts such as scheduling, deadlock management, and memory management.

**CO3:** Implement and realize the semantics of synchronous and asynchronous Inter Process communication models.

**CO4:** Demonstrate the memory segmentation and implement the programming model of Intel 8086 processor.

**CO5:** Realize the data and string manipulation instructions.

## References:


1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th edition, John Wiley, 2006.
2. Advanced Programming in the Unix Environment, W.R.Stevens, 2nd edition, Pearson education, 2015.
3. Operating Systems – Internals and Design Principles, William Stallings, 5th edition, Pearson Education/PHI, 2005.
4. Advance Microprocessors and Peripherals – A.K.Ray and K.M.Bhurchandani, TMH, 3rd edition, 2013.
5. Microprocessors and Interfacing – D.V.Hall, TMGH, 2nd edition, 2006.

1. **Implement shell commands such as cp, ls, chmod, ls -ls using the I/O system calls of UNIX/LINUX operating system.(open, read, write, close, fcntl, seek, stat, opendir, readdir)**

**cp:**

```
#include <stdio.h>      //Provide printf(), scanf()
#include <unistd.h>     // Provide read(),write(), and close()
#include <fcntl.h>      //Provide open(), access modes O_RDONLY, O_WRONLY etc
void main()
{
    char buf;           //A single-character buffer to store data read from the source file.
    int fd_one, fd_two; //File descriptors for the source ( file1 ) and destination ( file2 ) files

    fd_one = open("file1", O_RDONLY);     //Opens file1 in read-only mode

    if (fd_one == -1)           //-1 , indicating an error occurred while opening file1
    {
        printf("Error opening first_file\n");
        close(fd_one);
        return;
    }
    fd_two = open("file2", O_WRONLY | O_CREAT,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    while(read(fd_one, &buf, 1))
    {
        write(fd_two, &buf, 1);
    }
    printf("Successful copy");
    close(fd_one);
    close(fd_two);
}
```

//Procedure
1. Opens the source file ( file1 ) in read-only mode.
2. Handles errors if the source file cannot be opened.
3. Opens or creates the destination file ( file2 ) in write-only mode.
4. Reads the source file one byte at a time and writes it to the destination file.
5. Prints a success message after copying the contents.
6. Closes both files to release system resources.

//If file2 doesn't exist, it creates it with the specified permissions.
File permissions:
S_IRUSR : Read permission for the owner.
S_IWUSR : Write permission for the owner.
S_IRGRP : Read permission for the group.
S_IROTH : Read permission for others.

**OUTPUT:**

File1:

Hi

After execution:

Successful copy

File2

Hi

**ls:**

```
#include<stdio.h>                //printf() and scanf() .
#include<dirent.h>               //Provide opendir() , readdir() , and struct dirent
main()
{
    char dirname[10];            //A character array (string) to store the name of the directory entered by the user
    DIR*p;                       //A pointer to a DIR structure that represents a directory stream
    struct dirent *d;            //A pointer to a struct dirent structure, which holds information
    printf("Enter directory name\n");
    scanf("%s",dirname);
    p=opendir(dirname);          //Opens the directory specified by the user.
    if(p==NULL)                  //returned NULL (indicating an error).
    {
        perror("Cannot find directory");
        exit(-1);
    }
    while(d=readdir(p))          //readdir(p) : Reads the next entry in the directory stream p .
        printf("%s\n",d->d_name);
  }
```

//Procedure
1. Prompt the user to enter the name of a directory.
2. Attempt to open the directory using opendir() .
3. Handle errors if the directory cannot be opened (e.g., it doesn't exist or lacks permissions).
4. Use readdir() in a loop to read each entry in the directory.
5. Print the name of each entry.
6. The program terminates when all entries have been read.

**OUTPUT:**

Enter Directory name:

CSE

cannot find directory: No such file or directory

Enter directory name:

Cse

.

..

F1

F2

**Chmod:**

// Procedure

1. Argument Validation:
The program checks that the user provides exactly two arguments (mode and filename).
If not, it prints an error message and exits.
2. Parse and Convert Mode:
Converts the mode string (e.g., "755" ) to an octal value using strtol() .
3. Apply Permissions:
Uses the chmod() system call to apply the specified permissions to the file.
Handles errors if the operation fails.
4. Output Success:
Prints a success message if the permissions are changed successfully

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s <mode><filename>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *mode_str = argv[1];
    const char *filename = argv[2];
    // Convert the mode string to octal representation
    mode_t mode = strtol(mode_str, NULL, 8);
    // Change the file permissions using chmod
    if (chmod(filename, mode) == -1)
    {
        perror("chmod");
        exit(EXIT_FAILURE);
    }
    printf("File permissions changed successfully.\n");
```

//r functions like exit() and strtol() .

//For data types and function prototypes related to file operations, including chmod()

//int argc : The number of command-line arguments passed to the program.
char *argv[] : An array of strings containing the command-line arguments.

//1. argc != 3 : The program expects exactly two arguments:
argv[1] : The desired file permission mode (e.g., 755 ).
argv[2] : The name of the file whose permissions need to be changed.
2. fprintf(stderr, ...) : Prints an error message to the standard error ( stderr ) if the number
of arguments is incorrect.
3. exit(EXIT_FAILURE) : Terminates the program with a failure code.

//1. mode_str = argv[1] : Assigns the first argument (permission mode string) to mode_str .
2. filename = argv[2] : Assigns the second argument (file name) to filename .

strtol() function is used to convert a string to a long integer number according to the given base. which must be lies between 2 and 32

perror() function is designed to print a descriptive error message to the standard error stream (stderr), which helps in debugging

```
    return 0;
}
```

**OUTPUT:**

File permissions changed successfully.

Usage:./change_permissions<mode><filename>

Chmod: No such file or directory.

**ls -l:**

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
```

// <stdio.h>: Standard I/O functions like printf.
<stdlib.h>: Includes functions like exit and EXIT_FAILURE.
<sys/types.h>: Defines data types used in the file system APIs.
<dirent.h>: Provides directory handling functions (opendir, readdir).
<sys/stat.h>: Retrieves file attributes (size, permissions).
<unistd.h>: Provides access to the POSIX API.
<pwd.h>: Provides user account information.
<grp.h>: Provides group account information.
<time.h>: Handles time and date.

```
void print_permissions(mode_t mode)
{
    printf((S_ISDIR(mode)) ? "d" : "-");
    printf((mode & S_IRUSR) ? "r" : "-");
    printf((mode & S_IWUSR) ? "w" : "-");
    printf((mode & S_IXUSR) ? "x" : "-");
    printf((mode & S_IRGRP) ? "r" : "-");
    printf((mode & S_IWGRP) ? "w" : "-");
    printf((mode & S_IXGRP) ? "x" : "-");
    printf((mode & S_IROTH) ? "r" : "-");
    printf((mode & S_IWOTH) ? "w" : "-");
    printf((mode & S_IXOTH) ? "x" : "-");
```

//This function prints the file's permissions in the same format as ls -l

//S_ISDIR(mode): Checks if the file is a directory (d for directory, - for regular file).
Bitwise Operations (&):
S_IRUSR: Read permission for the owner.
S_IWUSR: Write permission for the owner.
S_IXUSR: Execute permission for the owner.
Similarly for GRP (group) and OTH (others).

```
}
void ls_l(const char *dirname)                //Opens a directory and iterates through its entries, printing details about each file.
{
        DIR *dir;
        struct dirent *entry;
        struct stat file_stat;

        if ((dir = opendir(dirname)) == NULL)        //Opens the directory specified by dirname
        {
                perror("opendir");                   //If it fails, perror prints an error message, and the program exits with EXIT_FAILURE
                exit(EXIT_FAILURE);
        }
        while ((entry = readdir(dir)) != NULL)       //readdir(dir): Reads entries (files/subdirectories) in the directory.
      {                                              //snprintf: Combines the directory name and file name into path.
                char path[PATH_MAX];
                snprintf(path, PATH_MAX, "%s/%s", dirname, entry->d_name);

                if (stat(path, &file_stat) == -1)    //stat(path, &file_stat): Retrieves the file's attributes and stores them in file_stat.
              {
                        perror("stat");
                        exit(EXIT_FAILURE);
                }
                struct passwd *user_info = getpwuid(file_stat.st_uid);   //getpwuid(file_stat.st_uid): Gets user name from user ID.
                struct group *group_info = getgrgid(file_stat.st_gid);   //getgrgid(file_stat.st_gid): Gets group name from group ID

                printf("%-20s %-8s %-8s %8lld ", entry->d_name, user_info-
                >pw_name,
                group_info->gr_name, (long long)file_stat.st_size);
                print_permissions(file_stat.st_mode);

                struct tm *time_info = localtime(&file_stat.st_mtime);
```

//localtime(&file_stat.st_mtime): Converts modification time into a human-readable format.
strftime: Formats the time (e.g., "Dec 25 15:30").
closedir(dir): Closes the directory.

```
                char time_str[80];
                strftime(time_str, sizeof(time_str), "%b %d %H:%M", time_info);

                printf(" %s\n", time_str);
        }
        closedir(dir);
}
int main(int argc, char *argv[])
{
        if (argc != 2)
       {
                fprintf(stderr, "Usage: %s <directory>\n", argv[0]);
                exit(EXIT_FAILURE);
        }
        ls_l(argv[1]);
        return 0;
}
```

**OUTPUT:**

```
File1.txt       user1   group1 1234   -rw—r—r--      Sep 07 15:45
File2.txt       user2   group2 5678   -rw—r—r--      Sep 07 15:50
File8.txt       user2   group2 1234   -rw—r—r--      Sep 07 16:45
```

**2. Write C programs to simulate the following CPU Scheduling algorithms:**

**FCFS:**

```
#include<stdio.h>
int main( )
{
        char p[10][10];
        int bt[10],wt[10],tat[10],i,n;
        float  avgwt,avgtat;
        printf("Enter number of processes:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter process %d name:\t",i+1);
                scanf("%s",p[i]);
                printf("Enter burst time\t");
               scanf("%d",&bt[i]);
        }
        wt[0]=avgwt=0;
        avgtat=tat[0]=bt[0];
        for(i=1;i<n;i++)
        {
                wt[i]=wt[i-1]+bt[i-1];
                tat[i]=wt[i]+bt[i];
                avgwt=avgwt+wt[i];
                avgtat=avgtat+tat[i];
        }
        printf("p_name\t B_time\t w_time\t turnarounftime\n");
        for(i=0;i<n;i++)
                printf("%s\t%d\t%d\t%d\n",p[i],bt[i],wt[i],tat[i]);
        printf("\navg waiting time=%f", avgwt/n);
        printf("\navgtat time=%f\n", avgtat/n);
```

```
    return 0;
}
```

**Output:**

```
Enter number of processes:4
Enter process 1 name: p1
Enter burst time        2
Enter process 2 name: p2
Enter burst time        4
Enter process 3 name: p3
Enter burst time        6
Enter process 4 name: p4
Enter burst time        8
```

| p_name | B_time | w_time | turnarounftime |
|---|---|---|---|
| p1 | 2 | 0 | 2 |
| p2 | 4 | 2 | 6 |
| p3 | 6 | 6 | 12 |
| p4 | 8 | 12 | 20 |

```
avg waiting time=5.000000
avgtat time=10.000000
```

**SJF:**

```
#include<stdio.h>
void main()
{
    int p[20], bt[20], wt[20], tat[20], i, k, n, temp; float wtavg,tatavg;
    //clrscr();
    printf("\nEnter the number of processes -- ");
    scanf("%d", &n);
    for(i=0;i<n;i++)
    {
```

```
            p[i]=i;
            printf("Enter Burst Time for Process %d -- ", i);
            scanf("%d", &bt[i]);
    }
    for(i=0;i<n;i++)
    for(k=i+1;k<n;k++)
    if(bt[i]>bt[k])
    {
          temp=bt[i];
          bt[i]=bt[k];
          bt[k]=temp;
          temp=p[i];
          p[i]=p[k];
          p[k]=temp;
  }
  wt[0] = wtavg = 0;
  tat[0] = tatavg = bt[0]; for(i=1;i<n;i++)
{
          wt[i] = wt[i-1] +bt[i-1];
          tat[i] = tat[i-1] +bt[i];
          wtavg = wtavg + wt[i];
          tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND
TIME\n");
for(i=0;i<n;i++)
          printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
          printf("\nAverage Turnaround Time -- %f", tatavg/n);
  getch();
}
```

**Output:**

Enter the number of processes -- 4

Enter Burst Time for Process 0 -- 2

Enter Burst Time for Process 1 -- 4

Enter Burst Time for Process 2 -- 6

Enter Burst Time for Process 3 – 8

| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
|---|---|---|---|
| P0 | 2 | 0 | 2 |
| P1 | 4 | 2 | 6 |
| P2 | 6 | 6 | 12 |
| P3 | 8 | 12 | 20 |

Average Waiting Time -- 5.000000

Average Turnaround Time -- 10.000000

**3. Write C programs to simulate the following CPU Scheduling algorithms:**

**a) Round Robin**

```
#include<stdio.h>
void main()
{
	int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
	float awt=0,att=0,temp=0;
	clrscr();
	printf("Enter the no of processes -- ");
	scanf("%d",&n);
	for(i=0;i<n;i++)
	{
		printf("\nEnter Burst Time for process %d -- ", i+1);
		scanf("%d",&bu[i]);
		ct[i]=bu[i];
	}
	printf("\nEnter the size of time slice -- ");
	scanf("%d",&t);
	max=bu[0];
	for(i=1;i<n;i++)
	if(max<bu[i])
		max=bu[i];
	for(j=0;j<(max/t)+1;j++)
	for(i=0;i<n;i++)
	if(bu[i]!=0)
		if(bu[i]<=t)
		{
			tat[i]=temp+bu[i];
			temp=temp+bu[i];
			bu[i]=0;
```

```
                }
                else
                {
                        bu[i]=bu[i]-t;
                        temp=temp+t;
                }
                for(i=0;i<n;i++)
                {
                        wa[i]=tat[i]-ct[i];
                        att+=tat[i];
                        awt+=wa[i];
                 }
                 printf("\nThe Average Turnaround time is -- %f",att/n);
                 printf("\nThe Average Waiting time is -- %f ",awt/n);
                 printf("\n\tPROCESS\t BURST TIME \t WAITING
                 TIME\tTURNAROUND TIME\n");
                for(i=0;i<n;i++)
                printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
                getch()
        }
```

**OUTPUT:**

Enter the no of processes -- 4

Enter Burst Time for process 1 -- 2

Enter Burst Time for process 2 -- 4

Enter Burst Time for process 3 -- 6

Enter Burst Time for process 4 -- 8

Enter the size of time slice -- 2

The Average Turnaround time is -- 12.000000

The Average Waiting time is -- 7.000000

| PROCESS | BURST TIME | WAITING TIME | TURNAROUND TIME |
|---|---|---|---|
| 1 | 2 | 0 | 2 |
| 2 | 4 | 6 | 10 |
| 3 | 6 | 10 | 16 |
| 4 | 8 | 12 | 20 |

**B) Priority Scheduling Algorithm:**

```
#include <stdio.h>

// Structure for each process
struct Process
{
        int pid;      // Process ID
        int burstTime; // Burst time
        int priority;  // Priority
        int waitingTime; // Waiting time
        int turnaroundTime; // Turnaround time
};

// Function to sort processes based on priority
void sortProcessesByPriority(struct Process proc[], int n)
{
struct Process temp;
        for (int i = 0; i< n - 1; i++)
        {
                for (int j = i + 1; j < n; j++)
                {
                        if (proc[i].priority> proc[j].priority)
                        {
```

```
                temp = proc[i];
                proc[i] = proc[j];
                proc[j] = temp;
            }
        }
    }
}

// Function to calculate waiting time and turnaround time
void calculateTimes(struct Process proc[], int n)
{
    proc[0].waitingTime = 0; // Waiting time for first process is 0
    proc[0].turnaroundTime = proc[0].burstTime; // Turnaround time = Burst
    time for first process

    // Calculate waiting time and turnaround time for remaining processes
    for (int i = 1; i< n; i++)
    {
        proc[i].waitingTime = proc[i - 1].waitingTime + proc[i -
        1].burstTime;
        proc[i].turnaroundTime = proc[i].waitingTime +
        proc[i].burstTime;
    }
}

// Function to display process details
void displayProcesses(struct Process proc[], int n)
{
    printf("PID\tPriority\tBurst Time\tWaiting Time\tTurnaround Time\n");
    for (int i = 0; i< n; i++)
    {
```

```
            printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", proc[i].pid, proc[i].priority,
            proc[i].burstTime, proc[i].waitingTime, proc[i].turnaroundTime);
        }
}

int main()
{
        int n;

        // Input number of processes
        printf("Enter the number of processes: ");
        scanf("%d", &n);

        struct Process proc[n]; // Array to store processes

        // Input process details
        for (int i = 0; i< n; i++) {
            printf("Enter details for process %d\n", i + 1);
            proc[i].pid = i + 1;
            printf("Burst Time: ");
            scanf("%d", &proc[i].burstTime);
            printf("Priority: ");
            scanf("%d", &proc[i].priority);
    }

// Sort processes based on priority
  sortProcessesByPriority(proc, n);

        // Calculate waiting time and turnaround time
        calculateTimes(proc, n);
```

```
    // Display process details
    printf("\nProcess details after scheduling:\n");
    displayProcesses(proc, n);

    return 0;
}
```

**OUTPUT:**

```
Enter the number of processes: 3
Enter details for process 1
Burst Time: 5
Priority: 2
Enter details for process 2
Burst Time: 8
Priority: 1
Enter details for process 3
Burst Time: 4
Priority: 3
Process details after scheduling:
PID    Priority    Burst Time    Waiting Time    Turnaround Time
2          1          8                0                8
1          2          5                8                13
3          3          4                13               17
```

**4. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        int alloc[10][10],max[10][10];
        int avail[10],work[10],total[10];
        int i,j,k,n,need[10][10];
        int m;
        int count=0,c=0;
        char finish[10];
        clrscr();
        printf("Enter the number of processes and resources:");
        scanf("%d%d",&n,&m);
        for(i=0;i<=n;i++)
                finish[i]='n';
        printf("Enter the claim matrix:\n");
        for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                scanf("%d",&max[i][j]);
                printf("Enter the allocation matrix:\n");
        for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                scanf("%d",&alloc[i][j]);
         printf("Resource vector:");
        for(i=0;i<m;i++)
                scanf("%d",&total[i]);
        for(i=0;i<m;i++)
```

```
        avail[i]=0;
for(i=0;i<n;i++)
for(j=0;j<m;j++)
avail[j]+=alloc[i][j];
for(i=0;i<m;i++)
        work[i]=avail[i];
for(j=0;j<m;j++)
        work[j]=total[j]-work[j];
for(i=0;i<n;i++)
for(j=0;j<m;j++)
        need[i][j]=max[i][j]-alloc[i][j];
A:
for(i=0;i<n;i++)
{
        c=0;
        for(j=0;j<m;j++)
        if((need[i][j]<=work[j])&&(finish[i]=='n'))
                c++;
        if(c==m)
        {
                printf("All the resources can be allocated to Process %d",
                i+1);
                printf("\n\nAvailable resources are:");
                for(k=0;k<m;k++)
                {
                        work[k]+=alloc[i][k];
                        printf("%4d",work[k]);
                }
        printf("\n");
        finish[i]='y';
        printf("\nProcess %d executed?:%c \n",i+1,finish[i]);
```

```
                count++;
            }
        }
        if(count!=n)
                goto A;
        else
        printf("\n System is in safe mode");
        printf("\n The given state is safe state");
        getch();
}
```

**OUTPUT:**

Enter the number of processes and resources: 4 3

Enter the claim matrix:

3 2 2

6 1 3

3 1 4

4 2 2

Enter the allocation matrix:

1 0 0

6 1 2

2 1 1

0 0 2

Resource vector:9 3 6

All the resources can be allocated to Process 2

Available resources are: 6 2 3

Process 2 executed?:y

All the resources can be allocated to Process 3 Available resourcesare: 8 3 4

Process 3 executed?:y

All the resources can be allocated to Process 4 Available resourcesare: 8 3 6

Process 4 executed?:y

All the resources can be allocated to Process 1

Available resources are: 9 3 6

Process 1 executed?:y

System is in safe mode

The given state is safe state

**5. Write C programs to illustrate the following IPC mechanisms.**

**Pipes:**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include<string.h>
#include<stdlib.h>

int main(void)
{
        int fd[2], nbytes;
        pid_tpid;
        char string[] = "Hello\n";
        charb[80];
        pipe(fd);
        pid=fork();
        if(pid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);
                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);
                /* Read in a string from the pipe */
                nbytes = read(fd[0], b, sizeof(b));
                printf("Received string: %s", b);
```

```
        }
        return(0);
}
```

**OUTPUT:**

Received string: Hello

**FIFO:**

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include<string.h>
#include<stdlib.h>
#include<fcntl.h>
#include<sys/ipc.h>

int main(void)
{
        intfd[2], nbytes,i,j,k;
        pid_tpid;
        charstring[] = "Hello\n";
        char b[80];
        i=mkfifo("/home/cvr/myfifo1",0666);
        pid=fork();
        if(pid == 0)
        {
                j=open("/home/cvr/myfifo1",O_RDWR);
                /* Send "string" through the output side of pipe */
                write(j, string, (strlen(string)+1));
                exit(0);
        }
```

```
        else
{
                k=open("/home/cvr/myfifo1",O_RDWR);
                /* Read in a string from the pipe */
                nbytes = read(k, b, sizeof(b));
                printf("Received string: %s", b);
        }
        return(0);
}
```

**OUTPUT:**

Received string: Hello

**6. Write C program to illustrate the Message Queues IPC mechanism.**

Message queues:

Msgsnd.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// message queue structure
struct mesg_buffer
{
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;
                // generate unique key
        key = ftok("somefile", 65);
                // create a message queue and return identifier
        msgid = msgget(key, 0666 | IPC_CREAT);
        message.mesg_type = 1;
        printf("Insert message  : ");
        gets(message.mesg_text);
                // send message
        msgsnd(msgid, &message, sizeof(message), 0);
                // display the message
        printf("Message sent to server : %s\n", message.mesg_text);
                return 0;
```

```
}
```

**OUTPUT:**

Insert message: hello

Message sent to server: hello

**Msgrcv.c**

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer
{
        long mesg_type;
        char mesg_text[100];
} message;

int main()
{
        key_t key;
        int msgid;
                // generate unique key
        key = ftok("somefile", 65);
                // create a message queue and return identifier
        msgid = msgget(key, 0666 | IPC_CREAT);
        printf("Waiting for a message from client...\n");
        // receive message
        msgrcv(msgid, &message, sizeof(message), 1, 0);
                // display the message
        printf("Message received from client : %s\n",message.mesg_text);
```

```
        // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);
        return 0;
}
```

**OUTPUT:**

Waiting for a message from client…

Message received from client: Hello,world!

7. **Write a C program to illustrate the Shared Memory IPC mechanism.**

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
    int i;
    void *shared_memory;
    char buff[100];
    int shmid;
    shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
    printf("Key of shared memory is %d\n",shmid);
    shared_memory=shmat(shmid,NULL,0);
    printf("Process attached at %p\n",shared_memory);
    printf("Enter some data to write to shared memory\n");
    read(0,buff,100); //get some input from user
    strcpy(shared_memory,buff); //data written to shared memory
    printf("You wrote : %s\n",(char *)shared_memory);
}
```

# OUTPUT:

```
Key of shared memory is 0
Process attached at 0x7ff28b5e4000
Enter some data to write to shared memory
hi
You wrote : hi
```

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
        int i;
        void *shared_memory;
        char buff[100];
        int shmid;
        shmid=shmget((key_t)2345, 1024, 0666);
        printf("Key of shared memory is %d\n",shmid);
        shared_memory=shmat(shmid,NULL,0); //process attached to shared
        memory segment
        printf("Process attached at %p\n",shared_memory);
        printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

**OUTPUT:**

```
Key of shared memory is 0
Process attached at 0x7f7a81689000
Data read from shared memory is : hi
```

**8. Write a C program to implement the Producer–Consumer problem using semaphores using UNIX/LINUX system calls.**

```
#include<stdio.h>
#include<stdlib.h>
int mutex=1,full=0,empty=3,x=0;
int main()
{
        int n;
        void producer();
        void consumer();
        int wait(int);
        int signal(int);
        printf("\n1.Producer\n2.Consumer\n3.Exit");
        while(1)
        {
                printf("\nEnter your choice:");
                scanf("%d",&n);
                switch(n)
                {
                        case 1: if((mutex==1)&&(empty!=0))
                                        producer();
                                else
                                        printf("Buffer is full!!");
                        break;

                        case 2: if((mutex==1)&&(full!=0))
                                        consumer();
                                else
                                        printf("Buffer is empty!!");
                        break;
```

```
                        case 3: exit(0);
                        break;
                }
        }
        return 0;
}
int wait(int s)
{
        return (--s);
}
int signal(int s)
{
        return(++s);
}
void producer()
{
        mutex=wait(mutex);
        full=signal(full);
        empty=wait(empty);
        x++;
        printf("\nProducer produces the item %d",x);
        mutex=signal(mutex);
}

void consumer()
{
        mutex=wait(mutex);
        full=wait(full);
        empty=signal(empty);
        printf("\nConsumer consumes item %d",x);
        x--;
```

```
        mutex=signal(mutex);

}
```

**OUTPUT:**

1.Producer

2.Consumer

3.Exit

Enter your choice:1

Producer produces the item 1

Enter your choice:2

Consumer consumes item 1

Enter your choice:3

**8086 Architecture and Registers:**

The **8086 microprocessor** is a 16-bit processor designed by Intel in 1978, which laid the foundation for the x86 architecture. Below is a detailed explanation of its architecture and registers:

## 8086 Architecture

The 8086 uses a **CISC (Complex Instruction Set Computer)** architecture with a segmented memory model. It consists of two primary units: the**Bus Interface Unit (BIU)** and the **Execution Unit (EU)**.

1. **Bus Interface Unit (BIU)**

- Responsible for interfacing with memory and I/O devices.
- Contains the **instruction queue** to support pipelining, allowing the pre-fetching of up to 6 bytes of instructions.
- Manages **address calculation** for memory access using **segmented memory**. The effective address is calculated using the segment registers and offset values.
- Controls the **data bus** and **address bus**.

2. **Execution Unit (EU)**

- Executes instructions fetched by the BIU.
- Contains the **arithmetic and logic unit (ALU)**, which performs arithmetic and logical operations.
- Accesses **general-purpose registers** for data manipulation.
- Manages **flags** and their updates based on the results of operations.

## Registers in 8086

8086 contains several types of registers categorized as general-purpose, segment, pointer, index, and flag registers.

1. **General-Purpose Registers**

These 16-bit registers can be used for data storage and manipulation. They can also be split into two 8-bit registers (high and low).

- **AX (Accumulator Register)**: Used for arithmetic operations, I/O, and string operations.
  - Split into: **AH** (high byte) and **AL** (low byte).
- **BX (Base Register)**: Often used as a pointer to data in memory.
  - Split into: **BH** (high byte) and **BL** (low byte).
- **CX (Count Register)**: Primarily used as a loop counter in instructions like `LOOP` and for shift/rotate operations.
  - Split into: **CH** and **CL**.
- **DX (Data Register)**: Used in I/O operations, multiplication, and division.
  - Split into: **DH** and **DL**.

2. **Segment Registers**

Segment registers hold the base address of the different memory segments, allowing the 8086 to access more than 64 KB of memory.

- **CS (Code Segment)**: Holds the base address of the code (program) currently being executed.
- **DS (Data Segment)**: Holds the base address of the data used by the program.
- **SS (Stack Segment)**: Points to the base of the stack, used in stack operations.
- **ES (Extra Segment)**: Used for extra data, particularly for string operations.

3. **Pointer and Index Registers**

Used to point to memory locations or for string and stack operations.

- **SP (Stack Pointer)**: Points to the current location within the stack (in SS segment).

- **BP (Base Pointer)**: Used primarily for accessing data on the stack in stack-related instructions.
- **SI (Source Index)**: Used as a pointer in string operations, relative to the DS segment.
- **DI (Destination Index)**: Used in string operations as the destination pointer, relative to the ES segment.

4. **Instruction Pointer (IP)**

- **IP (Instruction Pointer)**: Holds the offset address of the next instruction to be executed, relative to the CS (code segment) register.

5. **Flag Register (FLAGS)**

The **FLAGS** register is a 16-bit register containing various flags that indicate the result of operations or control certain aspects of the processor's operation.

- **Carry Flag (CF)**: Set if there's a carry out from the most significant bit in an arithmetic operation.
- **Zero Flag (ZF)**: Set if the result of an operation is zero.
- **Sign Flag (SF)**: Set if the result of an operation is negative.
- **Overflow Flag (OF)**: Set if there is an overflow in a signed arithmetic operation.
- **Parity Flag (PF)**: Set if the number of set bits in the result is even.
- **Auxiliary Carry Flag (AF)**: Used in binary-coded decimal (BCD) arithmetic.
- **Direction Flag (DF)**: Determines the direction of string processing (increment or decrement).
- **Interrupt Flag (IF)**: Controls the enabling and disabling of interrupts.

## Memory Segmentation

- **The 8086 processor uses a segmented memory model to access 1 MB of memory space. Each** memory address is calculated as:
- Physical Address = (Segment Address × 16) + Offset

- Segments are defined by segment registers, and the offset comes from registers like IP, SI, DI, etc.

## 8086 Addressing Modes:

The **Intel 8086** microprocessor supports various addressing modes to specify how the operand (data) for an instruction is accessed. These modes are broadly categorized into **register-based**, **memory-based**, and **immediate** addressing modes. Below is a summary of each addressing mode:

### 1. Immediate Addressing Mode

- The operand is a constant value specified directly in the instruction.
- Example: `MOV AX, 1234H` (The value 1234H is moved into AX).

### 2. Register Addressing Mode

- The operand is located in one of the CPU's general-purpose registers.
- Example: `MOV AX, BX` (The value in the BX register is moved into AX).

### 3. Direct Addressing Mode

- The effective address of the operand is provided directly in the instruction.
- Example: `MOV AX, [1234H]` (The value at memory address 1234H is moved into AX).

### 4. Register Indirect Addressing Mode

- The effective address of the operand is held in a register (BX, BP, SI, DI).
- Example: `MOV AX, [BX]` (The value at the memory address stored in BX is moved into AX).

### 5. Based Addressing Mode

- The effective address is the sum of a **base register** (BX or BP) and a displacement.
- Example: `MOV AX, [BX + 1234H]` (The content at the address formed by BX + 1234H is moved into AX).

**6. Indexed Addressing Mode**

- The effective address is the sum of an **index register** (SI or DI) and a displacement.
- Example: MOV AX, [SI + 1234H] (The content at the address formed by SI + 1234H is moved into AX).

**7. Based-Indexed Addressing Mode**

- The effective address is the sum of a **base register** (BX or BP) and an **index register** (SI or DI).
- Example: `MOV AX, [BX + SI]` (The content at the address formed by BX + SI is moved into AX).

**8. Based-Indexed with Displacement Addressing Mode**

- The effective address is the sum of a **base register**, an **index register**, and a displacement.
- Example: `MOV AX, [BX + SI + 1234H]` (The content at the address formed by BX + SI + 1234H is moved into AX).

**9. Relative Addressing Mode**

- The effective address is calculated relative to the **Instruction Pointer (IP)**. Mainly used in jump instructions.
- Example: `JMP LABEL` (Jump to a location relative to the current IP).

**10. Implied Addressing Mode**

- The operand is implied by the instruction itself, and no additional operand is specified.
- Example: `CLC` (Clear the carry flag; no operand is needed).

## 8086 Instruction Set:

The **8086 instruction set** consists of a wide variety of instructions that enable the processor to perform data transfer, arithmetic, logical, control, string, and machine control operations. Below is a summary of the key categories of instructions in the 8086 instruction set:

### 1. Data Transfer Instructions

These instructions are used to transfer data between registers, memory, or I/O devices.

- **MOV**: Move data from one location to another.
  Example: `MOV AX, BX` (Move content of BX into AX).
- **PUSH**: Push register/memory content onto the stack.
- **POP**: Pop content from the stack into a register/memory.
- **IN**: Read data from a port.
- **OUT**: Write data to a port.
- **XCHG**: Exchange the contents of two registers.
- **LEA**: Load effective address of a memory operand into a register.
- **LDS/LES**: Load DS/ES segment register and pointer with memory contents.

### 2. Arithmetic Instructions

These instructions perform arithmetic operations like addition, subtraction, and multiplication.

- **ADD**: Add two operands.
  Example: `ADD AX, BX` (AX = AX + BX).
- **SUB**: Subtract the second operand from the first.

- **INC**: Increment the operand by 1.
- **DEC**: Decrement the operand by 1.
- **MUL:** Unsigned multiply (AX or DX= AX * operand).
- **IMUL:** Signed multiply.
- **DIV:** Unsigned division (AX = dividend, operand = divisor).
- **IDIV:** Signed division.
- **ADC:** Add with carry.
- **SBB:** Subtract with borrow.

### 3. Logical Instructions

These instructions perform bitwise operations on operands.

- **AND**: Perform bitwise AND on two operands.
  Example: AND AX, BX (AX = AX AND BX).
- **OR**: Perform bitwise OR.
- **XOR**: Perform bitwise XOR.
- **NOT**: Perform bitwise NOT (complement).
- **TEST**: Perform bitwise AND, but the result is not stored (only sets flags).

### 4. Shift and Rotate Instructions

These instructions shift or rotate bits within a register or memory.

- **SHL (SAL)**: Shift bits to the left, filling with zeros.
- **SHR**: Shift bits to the right, filling with zeros.
- **SAR**: Arithmetic right shift (preserves sign).
- **ROL**: Rotate bits to the left.
- **ROR**: Rotate bits to the right.
- **RCL**: Rotate through carry flag to the left.
- **RCR**: Rotate through carry flag to the right.

### 5. Control Transfer Instructions

These instructions alter the flow of program execution.

- **JMP**: Unconditional jump to a specified address.
  Example: `JMP LABEL` (Jump to the instruction at LABEL).
- **CALL**: Call a procedure (subroutine).
- **RET**: Return from procedure.
- **JZ/JNZ**: Jump if zero/non-zero flag is set/clear.
- **JC/JNC**: Jump if carry flag is set/clear.
- **JE/JNE**: Jump if equal/not equal.
- **LOOP**: Loop a block of code a specified number of times.

### 6. String Manipulation Instructions

These instructions are used to perform operations on strings of data.

- **MOVSB/MOVSW**: Move byte/word from source to destination.
- **CMPSB/CMPSW**: Compare byte/word strings.
- **SCASB/SCASW**: Scan a string for a byte/word.
- **LODSB/LODSW**: Load byte/word from string into AL/AX.
- **STOSB/STOSW**: Store byte/word from AL/AX into string.

### 7. Flag Manipulation Instructions

These instructions control the flags within the flag register.

- **STC**: Set carry flag.
- **CLC**: Clear carry flag.
- **CMC**: Complement carry flag.
- **STD**: Set direction flag (for string operations).
- **CLD**: Clear direction flag.
- **STI**: Set interrupt flag.
- **CLI**: Clear interrupt flag.

### 8. Processor Control Instructions

These instructions control the processor's internal operations, typically for interrupts or halting the CPU.

- **HLT**: Halt the processor.
- **NOP**: No operation (do nothing).
- **WAIT**: Wait for the Test pin to become active.
- **ESC**: Escape to external device (used for coprocessors like 8087).
- **LOCK**: Assert the bus lock signal (used in multi-processor environments).

**9. Miscellaneous Instructions**

These instructions don't fit neatly into other categories but provide various utility functions.

- **INT**: Trigger a software interrupt.
- **IRET**: Return from interrupt.
- **XLAT**: Translate a byte in AL using a table in memory

**9. Write an ALP in 8086 add, subtract and multiply two 16-bit unsigned numbers.**

**Algorithm for the 8086 Program (16-bit Addition with Carry)**

This algorithm performs the addition of two 16-bit numbers (`n1` and `n2`), stores the result in `sum`, and accounts for any carry generated during the addition.

**Step-by-Step Algorithm:**

1. **Initialize the Data Segment**:
   - Set up the data segment to access the variables (n1, n2, and sum) from memory.
2. **Clear the Carry Register:**
   - Clear the BX register, which will be used to handle any carry that may occur during the addition.
3. **Load the First Number (n1) into AX:**
   Load the first 16-bit number (n1) from memory into the AX register.
4. **Add the Second Number (n2) to AX:**
   - Perform the addition of the second number (n2) with the value in the AX register.
5. **Store the Result in sum:**
   Store the result of the addition in the first two bytes of the sum variable.
6. **Check for Carry and Adjust:**
   - Check if a carry occurred during the addition using the ADC (Add with Carry) instruction. Add 0 to BX along with the carry, effectively storing the carry bit in BX.
7. **Store the Carry (if any) in sum[2]:**
   - If a carry occurred, store the value of BX (which contains the carry bit) in the next 2 bytes of sum.
8. **Exit the Program:**
   - Terminate the program and return control to DOS.

**Algorithm for 16-bit Subtraction Program in 8086 Assembly Language**

This algorithm explains the steps involved in subtracting two 16-bit numbers (`n1` and `n2`) and storing the result in memory, along with handling the borrow flag if the result is negative.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small to allocate 64KB each for code and data.
   - Allocate 256 bytes for the stack segment using .stack 100h.
2. **Define Variables in Data Segment:**
   - Declare the two 16-bit numbers (n1 and n2), and a 32-bit space (diff) to store the result of the subtraction.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Perform the Subtraction:**
   - Load the first 16-bit number (n1) into the AX register.
   - Subtract the second number (n2) from AX.
   - Store the result of the subtraction in the diff memory location.
5. **Handle Borrow (if Necessary):**
   - Clear the BX register to handle any potential borrow.
   - Use the SBB (Subtract with Borrow) instruction to check if a borrow occurred. If the result was negative (i.e., borrow), subtract 1 from BX.
   - Store the BX register value (borrow) in the high part of the diff variable.
6. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
7. **End the Program:**
   - Mark the end of the program with the end directive.

**Algorithm for 16-bit Multiplication Program in 8086 Assembly Language**

This algorithm explains the steps involved in multiplying two 16-bit numbers (n1 and n2) and storing the result (which can be up to 32 bits) in memory using the MUL instruction.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for code and 64KB for data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare the two 16-bit numbers (n1 and n2) and a 32-bit space (product) to store the result of the multiplication.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Perform 16-bit Multiplication:**
   - Load the first 16-bit number (n1) into the AX register.
   - Use the MUL instruction to multiply n1 by n2. The result of the multiplication will be stored in DX:AX (high 16 bits in DX, low 16 bits in AX).
   - Store the lower 16 bits of the result (AX) in the product variable.
   - Store the higher 16 bits of the result (DX) in the next part of the product variable.
5. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
6. **End the Program:**
   - Mark the end of the program with the end directive.

**10. Write an ALP in 8086 to implement ASCII Adjust and decimal adjust instructions.**

**Algorithm for the Program: ASCII Adjust After Addition (AAA) in 8086 Assembly Language**

This algorithm explains the steps for performing ASCII adjustment after adding two ASCII values in 8086 assembly language using the AAA instruction. The goal is to convert the result of a hexadecimal addition into a valid decimal ASCII code.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, allocating 64KB each for code and data.
   - Allocate a stack size of 256 bytes.
2. **Define Variables in Data Segment:**
   - Declare two ASCII characters (n1 and n2) as bytes.
   - Declare a word variable (ascadj) to store the result after performing ASCII adjustment.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Clear AX Register:**
   - Clear the AX register by performing an exclusive OR operation on AX to ensure there are no previous values.
5. **Load First ASCII Character into AL:**
   - Load the first ASCII character (n1) into the AL register.
6. **Add Second ASCII Character to AL:**
   - Add the second ASCII character (n2) to the value in AL. At this point, the sum will be in hexadecimal.
7. **Perform ASCII Adjustment After Addition (AAA):**
   - Use the AAA (ASCII Adjust After Addition) instruction to adjust the hexadecimal result of the addition into valid decimal ASCII codes. This instruction adjusts the contents of AL so that it contains a valid unpacked

decimal value in ASCII format. If the adjustment requires, the higher nibble of AL will be set to zero, and AH will increment by 1 (if needed).

8. **Store the Result:**
   - Store the adjusted result in the ascadj variable.
9. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
10. **End the Program:**
    - Mark the end of the program with the end directive.

**Algorithm for the Program: Decimal Adjust After Addition (DAA) in 8086 Assembly Language**

This algorithm explains the steps for performing a decimal adjustment after adding two hexadecimal numbers using the DAA (Decimal Adjust After Addition) instruction in 8086 assembly language. The goal is to convert the hexadecimal sum into a valid Binary-Coded Decimal (BCD) result.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB each for code and data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare two 8-bit hexadecimal numbers (n1 and n2).
   - Declare a word variable (decans) to store the result after performing decimal adjustment.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Clear AX Register:**
   - Clear the AX register by performing an exclusive OR operation on AX to

ensure it holds no previous values.

5. **Load First Number into AL:**
   - Load the first 8-bit hexadecimal number (n1) into the AL register.
6. **Add Second Number to AL:**
   - Add the second 8-bit hexadecimal number (n2) to the value in AL. The result of this addition will initially be in hexadecimal format.
7. **Perform Decimal Adjustment After Addition (DAA):**
   - Use the DAA (Decimal Adjust After Addition) instruction to adjust the result in AL so that it becomes a valid Binary-Coded Decimal (BCD) value. The DAA instruction modifies the contents of AL based on the flags to convert the result into a valid BCD form.
8. **Handle Carry:**
   - Use the ADC (Add with Carry) instruction to adjust the AH register if there is a carry after performing the decimal adjustment. This ensures any overflow or carry is correctly accounted for in the higher byte of AX.
9. **Store the Result:**
   - Store the adjusted result in the decans variable.
10. **Terminate the Program:**
    - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.

**11. Write an ALP to pack two digits into a Byte.**

**Algorithm for the Program:** Packing Two Digits into a Byte in 8086 Assembly Language

This algorithm explains how to pack two 4-bit digits (from two different bytes) into a single byte using 8086 assembly language. The goal is to pack the two digits such that the first digit is in the most significant nibble, and the second digit is in the least significant nibble.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare two 4-bit hexadecimal numbers (num1 and num2) that need to be packed into a single byte.
   - Declare a byte variable (packdnum) to store the packed result.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Clear AX Register:**
   - Clear the AX register by performing an exclusive OR operation on AX to ensure it holds no previous values.
5. **Load Most Significant Nibble:**
   - Load the first number (num1) into the AL register. This number will be treated as the most significant nibble (upper 4 bits).
6. **Load Least Significant Nibble:**
   - Load the second number (num2) into the BL register. This number will be treated as the least significant nibble (lower 4 bits).

7. **Rotate AL to Align Most Significant Nibble:**
    - Perform a left rotate operation on the AL register by 4 bits, which shifts the lower nibble of AL to the upper nibble. This positions the most significant nibble for packing.
8. **Pack Both Nibbles:**
    - Use the logical OR (OR) instruction to combine the contents of BL (least significant nibble) with the contents of AL (most significant nibble). This operation packs both digits into a single byte in AL.
9. **Store the Packed Byte:**
    - Store the packed byte from the AL register into the packdnum variable.
10. **Terminate the Program:**
    - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
11. **End the Program:**
    - Mark the end of the program with the end directive.

**12. Write an ALP to Count number of 1's and number of 0's present in the binary representation of a given number.**

**Algorithm for Counting the Number of Ones in a Number Using 8086 Assembly Language**

This algorithm outlines the steps to count the number of 1s in the binary representation of an 8-bit number using 8086 assembly language. The process involves shifting the bits of the number and checking if a carry is generated, which indicates a 1 in the most significant bit (MSB).

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare a variable n1 to store the 8-bit number (0FEh) in which we will count the number of ones.
   - Declare a variable count to store the result, which is the number of 1s in the binary form of n1.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Initialize Registers:**
   - Clear the AH register and move the value of n1 (the number in which to count ones) into the AL register.
   - Set CX to 8 (since the number is 8 bits long) to act as the loop counter.
   - Clear the BL register to use it as the counter for the number of 1s.
5. **Loop to Check Each Bit:**
   - Shift the contents of AL one bit to the left using the SHL instruction. The

most significant bit will be moved into the carry flag (CF).

- Use the JNC (jump if no carry) instruction to check if the carry flag is set. If there is no carry (i.e., the bit was 0), skip the increment.
- If the carry flag is set (i.e., the bit was 1), increment the BL register, which keeps track of the number of 1s.
- Decrement the loop counter CX and repeat the process until all 8 bits have been checked.

6. **Store the Result:**
   - Once the loop is complete, move the count of 1s from the BL register into the count variable.
7. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
8. **End the Program:**
   - Mark the end of the program with the end directive.

**Algorithm for Counting the Number of Zeros in a Number Using 8086 Assembly Language**

This algorithm counts the number of 0s in the binary representation of an 8-bit number using 8086 assembly language. The procedure involves shifting the bits of the number and checking whether a carry is generated. If no carry is generated, the bit is a 0, and the zero count is incremented.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare a variable n1 to store the 8-bit number (87h) in which the number of

zeros will be counted.

- Declare a variable count to store the result, which is the number of 0s in the binary form of n1.

3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
4. **Initialize Registers:**
   - Clear the AH register and move the value of n1 (the number in which to count zeros) into the AL register.
   - Set CX to 8 (since the number is 8 bits long) to act as the loop counter.
   - Clear the BL register to use it as the counter for the number of 0s.
5. **Loop to Check Each Bit:**
   - Shift the contents of AL one bit to the left using the SHL instruction. The most significant bit will be moved into the carry flag (CF).
   - Use the JC (jump if carry) instruction to check if the carry flag is set. If the carry flag is set (i.e., the bit was 1), skip the increment.
   - If there is no carry (i.e., the bit was 0), increment the BL register, which keeps track of the number of 0s.
   - Decrement the loop counter CX and repeat the process until all 8 bits have been checked.
6. **Store the Result:**
   - Once the loop is complete, move the count of 0s from the BL register into the count variable.
7. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
8. **End the Program:**
   - Mark the end of the program with the end directive.

**13. Implement the following string manipulation functions using appropriate registers.**

**a) Copy a string**

**Algorithm for Copying a Given String in 8086 Assembly Language**

This algorithm copies a given string (text1) into another string (text2) using 8086 assembly language. It uses the MOVSB instruction to copy each byte from the source string to the destination string, employing registers and memory pointers for handling the string manipulation.

**Algorithm Steps:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Variables in Data Segment:**
   - Declare a string text1 containing the message "hello welcome".
   - Declare count as the length of the string (13 characters).
   - Declare text2 with a space to hold the copied string, using the DUP directive to allocate 13 uninitialized bytes.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to initialize the data segment.
   - Also, move the same address into the ES register, as both the source (DS) and destination (ES) strings reside in the same data segment.
4. **Set Up Pointers and Counters:**
   - Load the offset of text1 (source string) into the SI register.
   - Load the offset of text2 (destination string) into the DI register.
   - Set the CX register with the value of count (the number of bytes to copy).

5. **Set the Direction Flag (CLD):**
   - Use the CLD instruction to ensure that the string copy happens in auto-increment mode, meaning the pointers (SI and DI) will be automatically incremented after each byte is copied.
6. **Copy the String Using REP MOVSB:**
   - Use the REP MOVSB instruction, which repeatedly moves bytes from the source (pointed to by SI) to the destination (pointed to by DI) until the counter CX becomes zero.
7. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
8. **End the Program:**
   - Mark the end of the program with the end directive.

**b) Lower to upper case**

**Algorithm for Converting Upper Case to Lower Case in 8086 Assembly**

This algorithm takes an array of lowercase characters (text) and converts them into uppercase characters by adjusting the ASCII values. In the ASCII table, the lowercase letters and their corresponding uppercase counterparts differ by 32 (20h). The algorithm modifies each character by subtracting 20h to convert it from lowercase to uppercase.

**Steps in the Algorithm:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Data Segment:**
   - Declare the string text with the lowercase alphabet.
   - Declare count to store the length of the string, which is 26 characters.

3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to both DS and ES registers to initialize the data segment. This ensures that both source and destination are in the same memory area.
4. **Set Up the SI Register and Counter:**
   - Set the SI register to 0 to point to the first character of the string (text).
   - Load the value of count (26) into the CX register to define how many characters to process.
5. **Convert Characters to Upper Case:**
   - The loop will read each character in the text string using SI as the pointer.
   - For each character:
   - Load the current character from text[si] into the AL register.
   - Subtract 20h from AL to convert the lowercase letter to uppercase (the ASCII value of uppercase letters is 32 units lower than lowercase letters).
   - Store the modified character back into text[si].
   - Increment SI to move to the next character.
   - Loop until all characters are processed.
6. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
7. **End the Program:**
   - Mark the end of the program with the end directive.

**c) Reverse of a String**

**Algorithm for Reversing a Given String in 8086 Assembly**

This algorithm reverses a given string (text1) and stores the reversed string into another string (text2). It uses the MOV instruction to transfer characters from the source to the destination, but in reverse order. The algorithm uses two pointers (SI for source and DI for destination) to achieve this.

**Steps in the Algorithm:**

1. **Select Memory Model and Stack:**
   - Define the memory model as small, which allocates 64KB for both code and data.
   - Allocate 256 bytes for the stack.
2. **Define Data Segment:**
   - Declare the source string text1 with the value "hello welcome".
   - Declare count to store the length of the string (13 characters).
   - Declare text2 with space for 13 bytes to hold the reversed string.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to both DS and ES registers to initialize the data segment.
4. **Set Up Pointers and Counter:**
   - Set the SI register to 0 to start from the beginning of text1.
   - Set the DI register to the length of the string (count), which will be used to point to the end of text2.
   - Decrement DI by 1 because the index is zero-based.
5. **Reverse the String:**
   - Loop through the string using CX as the counter:
   - Load a character from text1[SI] into the AL register.
   - Store the character into text2[DI].
   - Increment SI to move to the next character in text1.
   - Decrement DI to move to the previous position in text2.
   - Continue the loop until all characters are processed.
6. **Terminate the Program:**
   - Use the DOS interrupt INT 21h with service number AH = 4Ch to exit the program and return control to the operating system.
7. **End the Program:**
   - Mark the end of the program with the end directive.

**d) Palindrome**

**Algorithm for Checking if a String is a Palindrome in 8086 Assembly**

This algorithm checks whether the given string is a palindrome. A palindrome is a word or phrase that reads the same forward and backward. The program compares characters from the start and end of the string until the middle is reached.

**Steps in the Algorithm:**

1. **Initialize Memory Model:**
   - The memory model is set to small to allow for a 64KB code and data segment.
2. **Define Data Segment:**
   - Define the string STRING DB 'madam$', where $ is used to terminate the string for DOS interrupt.
   - Define S_LENTH to calculate the length of the string (excluding the terminating $).
   - Define two messages MSG1 and MSG2 to display whether the string is a palindrome or not.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to set up the data segment.
4. **Set Up Registers for String Comparison:**
   - Load the address of MSG2 (the "not a palindrome" message) into the DX register as the default.
   - Initialize the CX register with the length of the string (S_LENTH) to use as a counter for the loop.
   - Set SI to point to the start of the string (STRING).
   - Set DI to point to the last character of the string (STRING + S_LENTH - 1).
   - Divide CX by 2 (SHR CX, 1) to compare only half of the string.

5. **String Comparison Loop:**
   - Start the loop to compare characters from both ends of the string.
   - In each iteration, compare the characters at SI (start of the string) and DI (end of the string).
   - If the characters are not equal (CMP AL, [DI] and JNZ NEXT), jump to the NEXT label to display the "not a palindrome" message.
   - Otherwise, increment SI to move forward and decrement DI to move backward.
   - Continue the loop until half of the string is compared (LOOP BACK).
6. **Palindrome Check:**
   - If the entire loop completes without finding a mismatch, load the address of MSG1 (the "palindrome" message) into DX.
7. **Display the Result:**
   - Use DOS interrupt INT 21H with service number AH = 09H to display the appropriate message stored in DX.
8. **Exit the Program:**
   - Use DOS interrupt INT 21H with service number AH = 4CH to exit the program.
9. **End the Program:**
   - Mark the end of the program using the END directive.

**14. Write an ALP to Count no of even and odd numbers from the given array of numbers.**

**Algorithm for finding the Number of Odd and Even Numbers in an Array Using 8086 Assembly.**

This program checks whether each number in an array is odd or even by analyzing the least significant bit (LSB) of each element. If the LSB is 1, the number is odd; if it's 0, the number is even. It then counts how many odd and even numbers exist in the array.

**Steps in the Algorithm:**

1. **Initialize Memory Model:**
   - The memory model is set to small to allow for a 64KB code and data segment.
2. **Define Data Segment:**
   - The array n contains 10 numbers in hexadecimal format.
   - Count holds the total number of elements in the array.
   - Odd count and even count are variables to store the number of odd and even
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to set up the data segment.
4. **Initialize Registers:**
   - Set SI (source index) to 0 to start reading the array from the first element.
   - Load the count of numbers (count) into the CX register, which will be used as a loop counter.
   - Clear BX to store the count of odd numbers and DX to store the count of even numbers.
5. **Loop Through the Array:**
   - Start the loop to check each element in the array.
   - Load each number from the array into the AL register (MOV AL, n[SI]).
   - Use the ROR instruction to rotate the AL register right by one bit, moving the

LSB into the carry flag (CF).

- If the CF is set (JC odd), the number is odd, so increment BX (odd count).
- If the CF is not set, the number is even, so increment DX (even count).
- Update SI to point to the next number, decrement CX, and repeat until all numbers are processed.

6. **Store the Results:**
   - Once the loop is complete, store the values of BX (odd count) in odd count and DX (even count) in even count.
7. **Exit the Program:**
   - Use DOS interrupt INT 21H with service number AH = 4CH to exit the program.
8. **End the Program:**
   - Mark the end of the program using the END directive.

**15. Write a program to check whether a given number is Positive or Negative number.**

**Algorithm for Finding the Number of Positive and Negative Numbers in an Array Using 8086 Assembly**

This program checks if each number in an array is positive or negative by analyzing the most significant bit (MSB). If the MSB is 1, the number is negative; if it's 0, the number is positive. It counts how many positive and negative numbers are in the array.

**Steps in the Algorithm:**

1. **Initialize Memory Model:**
   - Set the memory model to small to allow for 64KB of data and code.
2. **Define Data Segment:**
   - The array n contains 10 numbers.
   - count holds the total number of elements in the array.
   - poscount and negcount are variables to store the count of positive and negative numbers, respectively.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to set up the data segment.
4. **Initialize Registers:**
   - Set SI (source index) to 0 to start reading the array from the first element.
   - Load the count of numbers (count) into the CX register, which will be used as a loop counter.
   - Clear BX to store the count of negative numbers and DX to store the count of positive numbers.
5. **Loop Through the Array:**
   - Start the loop to check each element in the array.
   - Load each number from the array into the AL register (MOV AL, n[SI]).

- Use the ROL (rotate left) instruction to rotate the AL register left by one bit, moving the MSB into the carry flag (CF).
- If the CF is set (JC neg), the number is negative, so increment BX (negative count).
- If the CF is not set, the number is positive, so increment DX (positive count).
- Update SI to point to the next number, decrement CX, and repeat until all numbers are processed.

6. **Store the Results:**
   - Once the loop is complete, store the values of BX (negative count) in negcount and DX (positive count) in poscount.
7. **Exit the Program:**
   - Use DOS interrupt INT 21H with service number AH = 4CH to exit the program
8. **End the Program:**
   - Mark the end of the program using the END directive.

**16. Write an ALP to sort the given array of numbers.**

**Algorithm for Sorting an Array in Ascending Order Using 8086 Assembly**

This algorithm implements **bubble sort** to sort an array of 10 numbers in ascending order. It compares adjacent elements and swaps them if the previous element is greater than the next. The process repeats until the entire array is sorted.

**Steps in the Algorithm:**

1. **Initialize Memory Model:**
   - The memory model is set to small for 64KB of data and code, which is suitable for small programs.
2. **Define Data Segment:**
   - The array n contains 10 hexadecimal numbers.
   - count holds the total number of elements in the array, which is 10 in this case.
3. **Initialize Data Segment:**
   - Load the data segment address into the AX register and move it to the DS register to set up access to the data segment.
4. **Initialize Registers:**
   - Clear the AX register (precaution).
   - Load the total count of numbers (count) into the CX register for loop control.
   - Decrement CX by 1 because bubble sort involves n-1 comparisons.
5. **Outer Loop (Next Scan):**
   - The outer loop (nxtscan) iterates through the array to perform multiple passes until all numbers are sorted.
   - Save the current count in BX (as a local loop counter for inner comparisons).
   - Clear SI to start from the first element in the array.
6. **Inner Loop (Next Comparison):**
   - The inner loop (nxtcomp) compares adjacent elements.
   - Load the current element into AL and the next element into DL.
   - Compare the two elements using the CMP instruction. If AL is less than or

equal to DL (i.e., no carry is generated), no swap is needed.

- If AL is greater than DL, swap the two elements.

7. **Move to Next Pair:**
   - Increment the SI register to point to the next pair of numbers.
   - Decrement BX (the local comparison counter) and continue the inner loop until all pairs in the current pass are compared.
8. **Repeat the Outer Loop:**
   - Use the LOOP instruction to repeat the outer loop for the remaining passes.
   - Continue the outer loop until the entire array is sorted.
9. **Exit the Program:**
   - Use DOS interrupt INT 21H with service number AH = 4CH to exit the program.
10. **End the Program:**
    - Mark the end of the program using the END directive.

**17. Write C programs to simulate Paging memory management techniques.**

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

#define FRAME_SIZE 4   // Size of each frame (in KB)
#define MEMORY_SIZE 16 // Total memory size (in KB)

int main()
{
int numPages, pageTable[10], pageNumber, offset;
        int memory[MEMORY_SIZE / FRAME_SIZE];
// Initialize memory frames as empty (-1)
for (int i = 0; i< MEMORY_SIZE / FRAME_SIZE; i++)
        {
        memory[i] = -1;
}
// Input the number of pages in the process
        printf("Enter the number of pages for the process: ");
        scanf("%d", &numPages);
// Allocate each page to a frame
for (int i = 0; i<numPages; i++)
        {
                printf("Enter frame number for page %d: ", i);
                scanf("%d", &pageTable[i]);

                if (pageTable[i] >= 0 &&pageTable[i] < MEMORY_SIZE /
                FRAME_SIZE)
                {
        memory[pageTable[i]] = i;  // Load page into frame
        }
```

```
                else
                {
                        printf("Invalid frame number!\n");
        return 1;
        }
}

// Input a logical address and translate to physical address
        printf("\nEnter a logical address (page number and offset):\n");
        printf("Page number: ");
        scanf("%d", &pageNumber);
        printf("Offset: ");
        scanf("%d", &offset);

       if (pageNumber>= numPages || pageTable[pageNumber] == -1)
       {
                printf("Invalid page number!\n");
        return 1;
}

if (offset >= FRAME_SIZE)
        {
                printf("Invalid offset (exceeds frame size)!\n");
        return 1;
}

// Calculate physical address
int frameNumber = pageTable[pageNumber];
int physicalAddress = frameNumber * FRAME_SIZE + offset;

        printf("\nLogical Address -> Page Number: %d, Offset: %d\n",
```

```
        pageNumber, offset);
        printf("Physical Address -> Frame Number: %d, Physical Address: %d\n",
        frameNumber, physicalAddress);

return 0;
        }
}
```

**OUTPUT:**

```
Enter the number of pages for the process: 3
Enter frame number for page 0: 1
Enter frame number for page 1: 3
Enter frame number for page 2: 2

Enter a logical address (page number and offset):
Page number: 1
Offset: 2

Logical Address -> Page Number: 1, Offset: 2
Physical Address -> Frame Number: 3, Physical Address: 14
```

**18. Write C programs to simulate Segmentation memory management techniques.**

```
#include <stdio.h>

// Structure to represent a segment

struct Segment

{

        int base;   // Base address of the segment

        int limit;  // Limit of the segment (size)

};

int main()

{

        int numSegments, segmentNumber, offset;

        // Input the number of segments in the process

        printf("Enter the number of segments: ");

        scanf("%d", &numSegments);

        // Array to store segment details

        struct Segment segments[numSegments];

        // Input segment base and limit for each segment

        for (int i = 0; i<numSegments; i++)
```

```
{
        printf("Enter base address and limit for segment %d:\n", i);
        printf("Base Address: ");
        scanf("%d", &segments[i].base);
        printf("Limit (Size): ");
        scanf("%d", &segments[i].limit);
}
// Input a logical address (segment number and offset)
printf("\nEnter a logical address (segment number and offset):\n");
printf("Segment Number: ");
scanf("%d", &segmentNumber);
printf("Offset: ");
scanf("%d", &offset);
// Check for valid segment number and offset
if (segmentNumber>= numSegments || segmentNumber< 0)
{
        printf("Invalid segment number!\n");
return 1;
}
```

```
    if (offset >= segments[segmentNumber].limit || offset < 0)
    {
        printf("Invalid offset! Offset exceeds segment limit.\n");
    return 1;
    }
    // Calculate physical address
    int physicalAddress = segments[segmentNumber].base + offset;
    // Output the physical address
    printf("\nLogical Address -> Segment Number: %d, Offset: %d\n", segmentNumber, offset);
    printf("Physical Address -> Base Address: %d, Physical Address: %d\n", segments[segmentNumber].base, physicalAddress);
     return 0;
}
```

**OUTPUT:**

```
Enter the number of segments: 3
Enter base address and limit for segment 0:
Base Address: 1000
Limit (Size): 300
Enter base address and limit for segment 1:
Base Address: 2000
Limit (Size): 400
Enter base address and limit for segment 2:
```

Base Address: 3000

Limit (Size): 500

Enter a logical address (segment number and offset):

Segment Number: 1

Offset: 250

Logical Address -> Segment Number: 1, Offset: 250

Physical Address -> Base Address: 2000, Physical Address: 2250