**A.Y. 2021-2022**

**Name:** Meith Navlakha

**SAP ID:** 60004190068

**Branch:** Computer Engineering

**Subject:** Processor Organization and Architecture

**Academic Year:** 2021-2022

**Semester:** V

**Division:** B

**Batch:** B1

**EXPERIMENT 1**

**DATE OF PERFORMANCE:** 23/09/2021 **DATE OF SUBMISSSION:** 30/09/2021

**AIM:** Write a program to implement Booths Multiplication Algorithm

**THEORY:**

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. It generates 2n bit product and treats both positive and negative unbiasedly.

Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed.

Basic flow of the Booths Algorithm:

1. Multiplier and multiplicand are placed in the Q and M register respectively.
2. Result for this will be stored in the A and Q registers.
3. Initially, A and Q-1 register will be 0.
4. Multiplication of a number is done in a cycle.
5. A 1-bit register Q-1 is placed right of the least significant bit Q0 of the register Q.
6. In each of the cycle, Q0 and Q-1 bits will be checked.
   1. If Q0 and Q-1 are 11 or 00 then the bits of A, Q and Q-1 are shifted to the right by 1 bit.
   2. If the value is shown 01 then multiplicand is added to AC. After addition, A, Q0, Q-1 register are shifted to the right by 1 bit.
   3. If the value is shown 10 then multiplicand is subtracted from AC. After subtraction A, Q0, Q-1 register is shifted to the right by 1 bit.

**Best Case :** When there is a large block of consecutive 1’s and 0’s in the multipliers, so that there is minimum number of logical operations taking place, as in addition and subtraction.

**Worst case** : When there are pairs of alternate 0’s and 1’s, either 01 or 10 in the multipliers, so that maximum number of additions and subtractions are required.

**FLOWCHART:**

Diagram

Description automatically generated

**CODE:**

import java.util.\*;

import java.util.Arrays;

import java.lang.Integer;

import java.lang.Math;

public class booths {

public static String displayArray(int[] arr) {

String s = "";

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

s = s + " " + arr[i];

}

return s;

}

public static int[] Add(int[] A, int[] M) {

int c = 0;

for (int i = A.length - 1; i >= 0; i--) {

A[i] = A[i] + M[i] + c;

if (A[i] > 1) {

A[i] = A[i] % 2;

c = 1;

} else {

c = 0;

}

}

return A;

}

public static int[] twoCompliment(int[] arr, int len) {

//1's Compliment

for (int i = 0; i < len; i++) {

arr[i] = (arr[i] + 1) % 2;

}

int[] plus1 = new int[len];

plus1[len - 1] = 1;

// Add 1

arr = Add(arr, plus1);

// System.out.print("\n 2's Compliment : " + displayArray(arr)) ;

return arr;

}

public static int[] tobinary(int num) {

int m = Math.abs(num);

// int len =8 ;

int[] arr = new int[10]; // Extra for sign bit

int count = 0;

while (m > 0) {

arr[count] = m % 2;

count++;

m /= 2;

}

int[] a = new int[count + 1];

for (int i = 1; i <= count; i++) {

a[count + 1 - i] = arr[i - 1];

}

// if(num <0){

// a = twoCompliment(a , count+1) ;

// }

// System.out.print("\n Binary of "+num+" : "+ displayArray(a)) ;

return a;

}

public static int[][] rightShift(int[] A, int[] Q, int q0, int len) {

int temp = A[len - 1];

q0 = Q[len - 1];

int[][] res = new int[3][];

for (int k = len - 1; k > 0; k--) {

A[k] = A[k - 1];

Q[k] = Q[k - 1];

}

Q[0] = temp;

res[0] = A;

res[1] = Q;

res[2] = new int[] {

q0

};

System.out.print("\n ARS : \t\t" + displayArray(A) + "\t" + displayArray(Q) + "\t" + q0);

return res;

}

public static int toDecimal(int[] arr) {

int num = 0;

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

num = num \* 2 + arr[i];

}

return num;

}

public static void combine(int[] A, int[] Q) {

if (A[0] == 1) {

// Negative

int[] result = new int[2 \* A.length - 1];

System.arraycopy(A, 1, result, 0, A.length - 1);

System.arraycopy(Q, 0, result, A.length - 1, A.length);

result = twoCompliment(result, result.length);

System.out.print("\n\n RESULT : " + displayArray(A) + "" + displayArray(Q) + " => -" + toDecimal(result));

} else {

int[] result = new int[2 \* A.length];

System.arraycopy(A, 0, result, 0, A.length);

System.arraycopy(Q, 0, result, A.length, A.length);

System.out.print("\n\n RESULT : " + displayArray(result) + " => " + toDecimal(result));

}

System.out.print("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

public static int[] padding(int[] arr, int len) {

int[] pad = new int[len];

int k = arr.length;

int i = 0;

while (i < len && k > 0) {

pad[len - arr.length + i] = arr[arr.length - k];

i++;

k--;

}

// System.out.print("\nPADDED : \t"+ displayArray(pad) );

return pad;

}

public static void boothsAlgo(int[] M, int[] minusM, int[] Q) {

int[] A = new int[M.length];

int q0 = 0;

int N = M.length;

System.out.print("\n Operation\t A\t Q \t q0");

System.out.print("\n INITIALISE : \t" + displayArray(A) + "\t" + displayArray(Q) + "\t" + q0);

System.out.print("\n\n N = " + N);

while (N > 0) {

if (q0 == 0 && Q[Q.length - 1] == 1) {

//10

A = Add(A, minusM);

System.out.print("\n A = A - M : \t" + displayArray(A) + "\t" + displayArray(Q) + "\t" + q0);

} else if (q0 == 1 && Q[Q.length - 1] == 0) {

// 01

A = Add(A, M);

System.out.print("\n A = A + M : \t" + displayArray(A) + "\t" + displayArray(Q) + "\t" + q0);

}

int[][] res = rightShift(A, Q, q0, A.length);

A = res[0];

Q = res[1];

q0 = res[2][0];

N--;

System.out.print("\n\n N = " + N);

}

// System.out.print("\n\n FINAL : "+ displayArray(A)+""+ displayArray(Q));

combine(A, Q);

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("\n Enter Multiplicand(M) : ");

int m = sc.nextInt();

System.out.print("\n Enter Multiplier(Q) : ");

int q = sc.nextInt();

int[] bin\_Q = tobinary(q);

int[] bin\_M = tobinary(m);

//int[] minusM = tobinary(-m) ;

int[] M;

int[] Q;

if (Math.abs(q) > Math.abs(m)) {

M = padding(bin\_M, bin\_Q.length);

Q = bin\_Q;

} else {

Q = padding(bin\_Q, bin\_M.length);

M = bin\_M;

}

int[] minusM = new int[M.length];

System.arraycopy(M, 0, minusM, 0, minusM.length);

if (q < 0) {

Q = twoCompliment(Q, Q.length);

}

if (m < 0) {

M = twoCompliment(M, M.length);

} else {

minusM = twoCompliment(minusM, minusM.length);;

}

System.out.print("\n M = " + m + " : " + displayArray(M));

System.out.print("\n -M = " + (-1 \* m) + " : " + displayArray(minusM));

System.out.print("\n Q = " + q + " : " + displayArray(Q));

System.out.print("\n\n \*\*\*\*\*\*BOOTHS ALGORITHM\*\*\*\*\*\n");

boothsAlgo(M, minusM, Q);

}

}

**OUTPUT:**

Text

Description automatically generated

Text

Description automatically generated

**CONCLUSION:**

In this experiment I implemented Booths Algorithm in Java. Booths Algorithm is used for signed multiplication of integers. Here, we use 2’s compliment for performing subtraction as processor executes addition much faster than subtraction also, Arithmetic Right Shift is used after every step. At the end we get 2n bit result where n is the number of bits of the highest value.

**EXPERIMENT 2**

**DATE OF PERFORMANCE:** 30/09/2021 **DATE OF SUBMISSSION:** 07/10/2021

**AIM:** To study and implement Restoring Division Algorithm.

**THEORY:**

**Restoring Division**

Restoring Division algorithm is a slow division algorithm. Here, register Q contains quotient and register A contains remainder. The n-bit dividend is loaded into Q and divisor is loaded into M. Value of Register is initially kept 0 and this is the register whose value is restored during iteration due to which it is named Restoring. Left Shift is performed in the algorithm and decision is taken which bit is to be added in the Q0 bit of Q. When all the iteration is over then the value in A is the remainder and value in Q is the quotient.

Restoring Division Algorithm:

* **Step-1:** First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)
* **Step-2:** Then the content of register A and Q is shifted left as if they are a single unit
* **Step-3:** Then content of register M is subtracted from A and result is stored in A
* **Step-4:** Then the most significant bit of the A is checked if it is 0 the least significant bit of Q is set to 1 otherwise if it is 1 the least significant bit of Q is set to 0 and value of register A is restored i.e. the value of A before the subtraction with M
* **Step-5:** The value of counter n is decremented
* **Step-6:** If the value of n becomes zero we get of the loop otherwise we repeat from step 2
* **Step-7:** Finally, the register Q contain the quotient and A contain remainder

**Disadvantages:** Slower, as requires time because of restoration in each cycle.

**FLOWCHART**

Diagram

Description automatically generated

**CODE:**

import java.util.\*;

import java.util.Arrays;

import java.lang.Integer;

import java.lang.Math;

public class RestoringDiv {

public static int[] Add(int[] A, int[] M) {

int c = 0;

for (int i = A.length - 1; i >= 0; i--) {

A[i] = A[i] + M[i] + c;

if (A[i] > 1) {

A[i] = A[i] % 2;

c = 1;

} else {

c = 0;

}

}

return A;

}

public static String displayArray(int[] arr) {

String s = "";

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

s = s + " " + arr[i];

}

return s;

}

public static int[] twoCompliment(int[] arr, int len) {

//1's Compliment

for (int i = 0; i < len; i++) {

arr[i] = (arr[i] + 1) % 2;

}

int[] plus1 = new int[len];

plus1[len - 1] = 1;

// Add 1

arr = Add(arr, plus1);

// System.out.print("\n 2's Compliment : " + displayArray(arr)) ;

return arr;

}

public static int[] tobinary(int num) {

int m = Math.abs(num);

int[] arr = new int[10];

int count = 0;

while (m > 0) {

arr[count] = m % 2;

count++;

m /= 2;

}

int[] a = new int[count];

for (int i = 0; i < count; i++) {

a[count - i - 1] = arr[i];

}

return a;

}

public static int[][] leftShift(int[] A, int[] Q, int N) {

int temp = Q[0];

int[][] res = new int[2][];

A[0] = A[1];

for (int k = 0; k < N - 1; k++) {

A[k + 1] = A[k + 2];

Q[k] = Q[k + 1];

}

A[N] = temp;

Q[N - 1] = -1;

res[0] = A;

res[1] = Q;

System.out.print("\n LS : \t\t" + displayArray(A) + "\t" + displayArray(Q));

return res;

}

public static int toDecimal(int[] arr) {

int num = 0;

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

num = num \* 2 + arr[i];

}

return num;

}

public static void restoringDiv(int[] M, int[] minusM, int[] Q, int N) {

int[] A = new int[N + 1];

int count = N;

System.out.print("\n Operation\t A\t Q ");

System.out.print("\n INITIALISE : \t" + displayArray(A) + "\t" + displayArray(Q));

System.out.print("\n\n N = " + count);

while (count > 0) {

int[][] res = leftShift(A, Q, N);

A = res[0];

Q = res[1];

A = Add(A, minusM);

System.out.print("\n A= A-M : \t" + displayArray(A) + "\t" + displayArray(Q));

if (A[0] == 1) {

// Negative

Q[N - 1] = 0;

System.out.print("\n Q[0] = 0 : \t" + displayArray(A) + "\t" + displayArray(Q));

A = Add(A, M);

System.out.print("\n A = A+M : \t" + displayArray(A) + "\t" + displayArray(Q));

} else {

Q[N - 1] = 1;

System.out.print("\n Q[0] = 1 : \t" + displayArray(A) + "\t" + displayArray(Q));

}

count--;

System.out.print("\n\n N = " + count);

}

System.out.print("\n\n\*\*\*\*\*RESULT\*\*\*\*\*");

System.out.print("\n QUOTIENT : " + displayArray(Q) + " => " + toDecimal(Q));

System.out.print("\n REMAINDER : " + displayArray(A) + " => " + toDecimal(A));

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("\n Enter Dividend(Q) : ");

int q = sc.nextInt();

System.out.print("\n Enter Divisor(M) : ");

int m = sc.nextInt();

int[] Q = tobinary(q);

int N = Q.length;

int[] M = new int[N + 1];

int[] minusM = new int[N + 1];

int[] bin\_M = tobinary(m);

// Padding

int index = bin\_M.length;

while (index > 0) {

M[N - bin\_M.length + index] = bin\_M[bin\_M.length - index];

index--;

}

System.arraycopy(M, 0, minusM, 0, minusM.length);

minusM = twoCompliment(minusM, minusM.length);

System.out.print("\n Q = " + q + " : " + displayArray(Q));

System.out.print("\n M = " + m + " : " + displayArray(M));

System.out.print("\n -M = " + (-1 \* m) + " : " + displayArray(minusM));

System.out.print("\n\n \*\*\*\*\*\*RESTORING DIVISION ALGORITHM\*\*\*\*\*\n");

restoringDiv(M, minusM, Q, N);

}

}

**OUTPUT:**

Text

Description automatically generated

**CONCLUSION:**

In this experiment I implemented Restoring Division Algorithm in Java. Here we take two inputs: Divisor(Q) and Dividend(M) and convert them into binary. Iterating, accumulator(A) and Q till N(the number of bits in binary of Q). Left Shift and Subtraction by 2’s Compliment is used during the process. Finally, value of Q is the quotient and value of A is the remainder. Newton–Raphson and Goldschmidt are faster than Restoring Division Algorithm.

**EXPERIMENT 3**

**DATE OF PERFORMANCE:** 07/10/2021 **DATE OF SUBMISSSION:** 14/10/2021

**AIM:** To implement Best Fit Memory Allocation

**THEORY:**

Memory allocation is the process of assigning blocks of memory to process on request. Typically the allocator receives memory from the operating system and it must assign it to the appropriate process to satisfy the request. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses.

In this experiement we will implement **Best Fit Memory Allocation**.

**Best-Fit Allocation**

The best fit deals with allocating the smallest free partition which meets the requirement of the requesting process. This algorithm first searches the entire list of free partitions and considers the free block that gives the least internal fragmentation. Allocates the process to that memory block.

It is used for Partition Memory allocation where the memory is pre partitioned into equal or unequal blocks.

**Advantages:**

* Memory Efficient, as the operating system allocates the job to the minimum possible space in the memory.
* Minimizes internal Fragmentation
* Memory utilization is much better than first fit as it searches the smallest free partition first available

**Disadvantages:**

* It is slower because it scans through the entire memory list every time and tries to find out the smallest block big enough to hold the process.
* Leads to Internal fragmentation
* Cannot be used for where fast execution of processes is expected.

**CODE:**

import java.util.\*;

import java.util.Arrays;

import java.lang.Math;

import java.lang.Integer;

public class BestFit {

public static int findBlock(int curr\_process, int[] curr\_mem, int[] busy\_mem, int num\_p, int num\_b) {

int index = -1;

int lowest = Integer.MAX\_VALUE;

for (int i = 0; i < num\_b; i++) {

//Empty

if (busy\_mem[i] == 0 && (curr\_mem[i] - curr\_process) >= 0 && lowest > (curr\_mem[i] - curr\_process)) {

lowest = curr\_mem[i] - curr\_process;

index = i;

}

}

return index;

}

public static void bestFit(int[] process, int[] memory, int num\_p, int num\_b) {

int[] busy\_mem = new int[num\_b];

int[] internal\_fragment = new int[num\_b];

int[] exe\_process = new int[num\_b];

// int[]

System.out.print("\n Memory Block\tJob\tJob Size\tStatus\tInternal Frag");

for (int i = 0; i < num\_p; i++) {

int index = findBlock(process[i], memory, busy\_mem, num\_p, num\_b);

if (index == -1) {

// No Space

} else{

busy\_mem[index] = 1;

exe\_process[index] = i;

internal\_fragment[index] = memory[index] - process[i];

}

} //closes for

// Display

int total = 0, total\_frag = 0;

for (int i = 0; i < num\_b; i++) {

if (busy\_mem[i] == 1) {

System.out.print("\n " + memory[i] + "\t\tP" + (exe\_process[i] + 1) + "\t" + (process[exe\_process[i]]) + "\t\tBusy\t" + internal\_fragment[i]);

total += process[exe\_process[i]];

total\_frag += internal\_fragment[i];

} else {

System.out.print("\n " + memory[i] + "\t\tNone" + "\t-\t" + "\tFree");

}

}

System.out.print("\n Total Used : \t\t" + total + "\t\t\t" + total\_frag);

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

System.out.print("\n Enter the number of Memory Blocks : ");

int num\_block = sc.nextInt();

int[] memory = new int[num\_block];

System.out.print("\n\n Enter the Sizes of Memory Blocks : \n");

for (int i = 0; i < num\_block; i++) {

System.out.print(" Size of Memory Block B" + (i + 1) + " : ");

memory[i] = sc.nextInt();

}

System.out.print("\n\n Enter the number of Processes : ");

int num\_process = sc.nextInt();

int[] process = new int[num\_process];

System.out.print("\n\n Enter the Sizes of Processes : \n");

for (int i = 0; i < num\_process; i++) {

System.out.print(" Size Requested by P" + (i + 1) + " : ");

process[i] = sc.nextInt();

}

bestFit(process, memory, num\_process, num\_block);

}

}

**OUTPUT:**

Text

Description automatically generated

**CONCLUSION:**

In this experiment I implemented Best Fit Memory Allocation in Java. Here, the OS allocates the free memory block to the process with the least internal fragmentation. The advantages of Best Fit are efficient memory utilization as it allocates the most appropriate memory block. But it is very slow as it has to scan through the entire free/allotted block list.

**EXPERIMENT 4**

**DATE OF PERFORMANCE:** 14/10/2021 **DATE OF SUBMISSSION:** 21/10/2021

**AIM:** Implement Sequential Memory Organization with the given details: Processor can access one word at a time (Word accessible memory), L1 cache can store max 32 words, L2 cache can store 128 words, main memory has the capacity of 2048 bytes. Consider TL1=20 ns, TL2=60ns, TMM=120ns.

**THEORY:**

Cache memory is used to reduce the average time to access data from the Main memory. The cache is a smaller and faster memory which stores copies of the data from frequently used main memory locations.

Thus, in order to access the data from the secondary memory, load it into the L1 and L2 cache and make it available for the processor, a linking or mapping technique is needed for accessing the data and passing it through the memory hierarchy. This is explained by **mapping functions**.

**Mapping Functions:**

The mapping functions are used to map a particular block of main memory to a particular block of memory of cache. This mapping is used to transfer the block from the main memory to cache memory. Three different mapping functions are available:

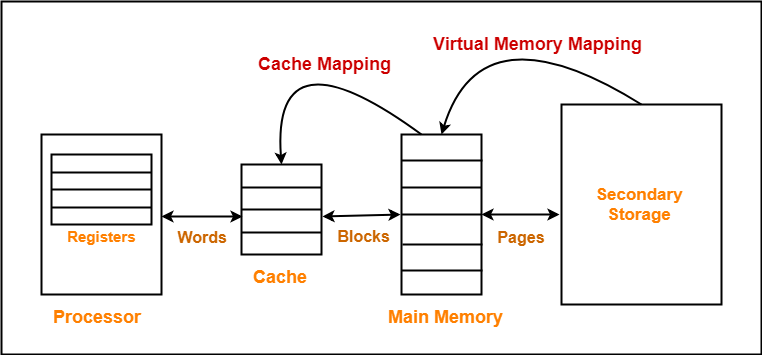
1**) Direct Mapping:** In this technique, block k of main memory maps into k modulo m of the cache, where m is the total number of blocks in cache. A particular block of memory can be brought to a particular block of cache memory. So, it is not flexible.

2) **Associative Mapping:** In this mapping function any block of main memory can potentially reside in any cache block position. In this case, the main memory is divided into two groups, low-order bits identifies the location within a block and high-order bits identifies the block. This is much more flexible mapping method.

3) **Block Set-Associative Mapping:** In this method, blocks of cache are grouped into sets, and the mapping allows a block of main memory to reside in any block of a specific set. From the flexibility point of view, it is in between the other two methods. This method reduces the searching overhead, because the search is restricted to number of sets, instead of blocks.

All the above 3 mapping techniques require **Page Replacement Policy** when the cache is full. Th property of locality of reference gives some clue to design a good replacement policy. Some types:

* LRU
* FIFO



**CODE:**

import java.util.\*;

import java.util.Arrays;

public class SeqMem {

public static int MAX\_PAGES = 64;

public static int MAX\_BLOCKS = 64;

public static int MAX\_WORDS = 32;

public static int[][][] mm = new int[MAX\_PAGES][4][2];

public static int[][] L2 = new int[MAX\_BLOCKS][2];

public static int[] L1 = new int[MAX\_WORDS];

public static int processor = -1;

public static int TL1 = 20;

public static int TL2 = 60;

public static int TMM = 120;

public static int in\_index\_L1 = -1;

public static int in\_index\_L2 = -1;

public static int[] searchMM(int elem) {

int p = (elem / (mm[0].length \* mm[0][0].length) % MAX\_PAGES);

int b = (elem / 2) % 4;

int w = elem % 2;

System.out.print(" W" + elem + " Found in Main Memory --> PAGE " + p);

System.out.print("\n TOTAL TIME TAKEN : " + (TL1 + TL2 + TMM) + " ns\n");

return mm[p][b];

}

public static int searchL2(int elem) {

if (in\_index\_L2 == -1) {

// Empty

int[] b = searchMM(elem);

L2[++in\_index\_L2] = b;

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

if (b[i] == elem) {

return b[i];

}

}

} else {

for (int i = 0; i <= in\_index\_L2; i++) {

for (int k = 0; k < L2[0].length; k++) {

if (L2[i][k] == elem) {

System.out.print(" W" + elem + " Found in L2 --> BLOCK " + i);

System.out.print("\n TOTAL TIME TAKEN : " + (TL1 + TL2) + " ns\n");

return elem;

}

}

}

int[] b = searchMM(elem);

in\_index\_L2++;

L2[in\_index\_L2 % MAX\_BLOCKS] = b;

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

if (b[i] == elem) {

return b[i];

}

}

}

return 0;

}

public static int searchL1(int elem) {

if (in\_index\_L1 == -1) {

// Empty

int w = searchL2(elem);

in\_index\_L1++;

L1[in\_index\_L1] = w;

return w;

} else {

for (int i = 0; i <= in\_index\_L1; i++) {

if (L1[i] == elem) {

System.out.print(" W" + elem + " Found in L1 WORD --> " + i);

System.out.print("\n TOTAL TIME TAKEN : " + (TL1) + " ns\n");

// break;

return elem;

}

}

int w = searchL2(elem);

in\_index\_L1++;

L1[in\_index\_L1 % MAX\_WORDS] = w;

return w;

}

}

public static void SeqMemSearch(int elem) {

if (processor != elem) {

int w = searchL1(elem);

processor = w;

} else {

System.out.print("W" + elem + " Found in Processor.");

System.out.print("\n TIME TAKEN : 0ns\n");

}

}

public static void main(String args[]) {

Scanner sc = new Scanner(System.in);

// Data in MM

int data = 0;

for (int i = 0; i < MAX\_PAGES; i++) {

for (int j = 0; j < mm[0].length; j++) {

for (int k = 0; k < mm[0][0].length; k++) {

mm[i][j][k] = data;

data++;

}

}

}

int elem = 0;

while (elem != -1) {

System.out.print("\n Enter the word to Search : ");

elem = sc.nextInt();

if (elem == -1) {

break;

} else {

SeqMemSearch(elem);

}

}

}

}

**OUTPUT:**

Text

Description automatically generated

**CONCLUSION:**

In this experiment, I implemented sequential memory organization having Main Memory of size 2048 bytes, L2 having capacity of 128 words, L1 having capacity of 32 words and processor which can store 1 word, where 1word is equal to 8 bytes. User requests for a word, the system will find it in the processor, if not there then finds in the L1 cache, if not there then finds in the L2 cache and if not there finally into main memory. While traversing back the corresponding block and the word will be stored in L2 and L1 respectively (all the higher levels of memory) so that it is present in the cache incase of future request for the same word. FIFO is used as a page replacement policy. The final output also calculates the total time taken to fetch the word from the memory storage hierarchy.

**EXPERIMENT 5**

**DATE OF PERFORMANCE:** 21/10/2021 **DATE OF SUBMISSSION:** 28/10/2021

**AIM:** Implement Direct Memory Mapping using Given set of data and links. Use UNC miniMIPS Simulator and C Program.

**THEORY:**

Direct mapping is a procedure used to assign each memory block in the main memory to a particular line in the cache. If a line is already filled with a memory block and a new block needs to be loaded, then the old block is discarded from the cache.

Direct mapping divides an address into three parts:

* t tag bits
* l line bits
* w word bits.

The word bits are the least significant bits that identify the specific word within a block of memory. The line bits are the next least significant bits that identify the line of the cache in which the block is stored. The remaining bits are stored along with the block as the tag which locates the block’s position in the main memory.

Cache line number = (Main Memory Block Address) Modulo (Number of lines in Cache)

Diagram

Description automatically generated with medium confidence

After CPU generates a memory request,

* The line number field of the address is used to access the particular line of the cache.
* The tag field of the CPU address is then compared with the tag of the line.
* If the two tags match, a **cache hit** occurs and the desired word is found in the cache.
* If the two tags do not match, a **cache miss** occurs. In case of a cache miss, the required word has to be brought from the main memory.
* It is then stored in the cache together with the new tag replacing the previous one.

Diagram, schematic

Description automatically generated

**UNC miniMIPS Assembly Language:**

A typical line of assembly code specifies a single primitive operation, called an instruction, and its operands. Instructions are specified via short mneumonics of the operation specified. A list of comma-separated operands follows each instruction's mneumonic. The assembler uses this mneumonic and operand list to generate a binary encoding of the instruction, which is stored as a word in memory. Thus, an assembly language program is merely a way of generating a sequence of binary words, that the computer interprets as either (or both) program or data.

Example line of assembly code

add $3, $4, $4 .… ("add" is the instruction mneumonic and "$3,$4,$4" is a list of 3 operands)

**CODE:**

**Trace file used:**

main: addu $t0,$0,$0

addiu $t1,$0,80

addu $t2,$0,$0

loop: lw $t3,array($t0)

addu $t2,$t2,$t3

addiu $t0,$t0,4

bne $t0,$t1,loop

\*done: beq $0,$0,done

array: .word 1,2,3,4,5,6,7,8,9,10

.word 11,12,13,14,15,16,17,18,19,20

**trace.txt:**

0x80000000

0x80000004

0x80000008

0x8000000C

0x00000020

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000024

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000028

0x80000010

0x80000014

0x80000018

0x8000000C

0x0000002C

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000030

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000034

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000038

0x80000010

0x80000014

0x80000018

0x8000000C

0x0000003C

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000040

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000044

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000048

0x80000010

0x80000014

0x80000018

0x8000000C

0x0000004C

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000050

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000054

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000058

0x80000010

0x80000014

0x80000018

0x8000000C

0x0000005C

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000060

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000064

0x80000010

0x80000014

0x80000018

0x8000000C

0x00000068

0x80000010

0x80000014

0x80000018

0x8000000C

0x0000006C

0x80000010

0x80000014

0x80000018

**A) Direct Mapping(tag size=8 bits):**

#include <stdio.h>

int tag[8];

int main( )

{

int addr;

int i, j, t;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

printf("Direct Mapping Program (tag size = 8 bits):\n\n");

while (fscanf(fp, "%x", &addr) > 0) {

/\* simulate a direct-mapped cache with 8 words \*/

accesses += 1;

printf("%3d:0x%08x ", accesses, addr);

printf("\n")

i = (addr >> 2) & 7;

printf(" Cache Location:%d ",i);

t = addr | 0x1f;

printf(" Instruction Address:0x%08x ",t);

printf("tag: 0x%08x ",tag[i]);

if (tag[i] == t) {

hits += 1;

printf(" 'Hit' at %d ", i);

} else {

/\* allocate entry \*/

printf(" 'Miss' ");

tag[i] = t;

}

printf("\n Cache Status:");

for (i = 0; i < 8; i++)

printf("0x%08x ",tag[i]);

printf("\n\n");

}

printf("\n\n");

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

return 0;

}

**B) Direct Mapping (tag size=16 bits):**

#include <stdio.h>

int tag[16];

int main( )

{

int addr;

int i, j, t;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

printf("Direct Mapping Program:\n\n");

while (fscanf(fp, "%x", &addr) > 0) {

/\* simulate a direct-mapped cache with 8 words \*/

accesses += 1;

printf("%4d:0x%08x ", accesses, addr);

printf("\n");

i = (addr >> 2) & 15;

printf(" Cache Location:%d ",i);

t = addr | 0x1f;

printf("\t Instruction Address:0x%08x ",t);

printf("\t tag: 0x%08x ",tag[i]);

if (tag[i] == t) {

hits += 1;

printf("\t 'Hit' at %d ", i);

} else {

/\* allocate entry \*/

printf("\t 'Miss' ");

tag[i] = t;

}

printf("\n Cache Status:");

for (i = 0; i < 16; i++)

printf("0x%08x ",tag[i]);

printf("\n\n");

}

printf("\n\n");

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

return 0;

}

**Output:**

**A) Direct Mapping (tag size = 8 bits):**

Text

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

Text

Description automatically generated

**B) Direct Mapping (tag size = 16 bits):**

Table

Description automatically generated

Table

Description automatically generated

Table

Description automatically generated

**ANALYSIS:**

In Direct Mapping, basic requirement is that it Needs only one comparison because of using direct formula to get the effective cache address. Main Memory Address is has 3 fields : TAG, BLOCK & WORD. The BLOCK & WORD together make an index. The least significant TAG bits is used to identify a unique word within a BLOCK of Main Memory. In the cache organization there is aa unique address for each block in the main memory which can be interpreted from the 3 fields. If the processor needs to access same memory location from 2 different main memory pages frequently, cache hit ratio decreases. Search time is less here because there is one possible location in the cache organization for each block from main memory.Thus, Direct mapping is simple and easy to implement, fast performance, and takes less time to detect a cache hit and get data from the cache.

**CONCLUSION:**

In this Experiment I implemented Direct Mapping in Cache memory in C programming language using UNC miniMIPS Simulator. First we generated a trace.txt file and observe the number of hits and misses occurred if the tag matches or not with the instruction address. Finally, Hit Ratio is computed, which is greater than 50% for both cases Direct mapping with tag size =8 bits and 16 bits, thus, the number of hits occurred were greater than the number of miss. For 16 bits tag has a larger hit ratio for direct mapping than the 8 bits tag size for the same number of accesses because of the larger cache size.

**EXPERIMENT 6**

**DATE OF PERFORMANCE:** 28/10/2021 **DATE OF SUBMISSSION:** 11/11/2021

**AIM:** To implement Fully Associative Mapping and Set Associative Mapping for 8bits and 16bits tag using UPC miniMIPS Simulator and C program. Compare the performance of 3 different organizations with the modification in the tag size.

**THEORY:**

**Fully Associative Mapping:**

Fully associative cache contains a single set with B ways, where B is the number of blocks. A memory address can map to a block in any of these ways. A fully associative cache also known as B-way set associative cache with one set. Here the associative memory is used to store the content and the addresses of the memory word. Any block can go into any line of the cache. This means that the word-id bits are used to identify which word in the block is needed, but the tag becomes all of the remaining bits. This enables the placement of any word at any place in the cache memory. It is the fastest and the most flexible mapping form.

Diagram

Description automatically generated

The program maintains an array with 8/16 tags and the most-recently-used ordering of the lines in the array mru[]. When each address is read from the trace file, it is compared to all of the tags in the cache in the first for loop. If the tag is found, a hit is recorded, and the mru[] array is updated using the mruUpdate() function, and the loop is exited via the break statement. A miss is detected when no matches are found after searching all 8/16 tags. In this case the loop index, i, will be set to 8/16. On a miss the least recently-used tag, which should be the last element in mru[], is chosen for replacement, the tag is updated, and the mru[] array is updated.

**Set Associative Mapping:**

In k-way set associative mapping, cache lines are grouped into sets where each set contains k number of lines. A particular block of main memory can map to only one particular set of the cache however, within that set, the memory block can map to any freely available cache line. Set-associative mapping allows that each word that is present in the cache can have two or more words in the main memory for the same index address. Set associative cache mapping combines the best of direct and associative cache mapping techniques.

Diagram, engineering drawing

Description automatically generated

The set of the cache to which a particular block of the main memory can map is given by:

Cache et number = ( Main Memory Block Address ) Modulo (Number of sets in Cache)

Diagram

Description automatically generated

In the program, the 8 tags are stored as a 2 by 4 entry array, where the first array index selects between the lines in the 2-way set and the second index selects one of 4 lines. The second array, mru[], tracks the most recently used of the two lines in each set. When each address is read from the trace file, it is compared to all of the tags in the cache in the first for loop. If the tag is found, a hit is recorded, and the mru[] array is updated using the mruUpdate() function, and the loop is exited via the break statement else it’s a miss and it is loaded into the memory.

**CODE:**

**1) Fully-Associative Cache (Tag 8bit):**

include <stdio.h>

int tag[8];

int mru[8] = {7,6,5,4,3,2,1,0};

void mruUpdate(int index)

{

int i;

// find index in mru

for (i = 0; i < 8; i++)

if (mru[i] == index)

break;

// move earlier refs one later

while (i > 0) {

mru[i] = mru[i-1];

i--;

}

mru[0] = index;

}

int main( )

{

int addr;

int i, j, t;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

while (fscanf(fp, "%x", &addr) > 0) {

accesses += 1;

printf("%3d: 0x%08x ", accesses, addr);

for (i = 0; i < 8; i++) {

if (tag[i] == addr) {

hits += 1;

printf("Hit%d ", i);

mruUpdate(i);

break;

}

}

if (i == 8) {

/\* allocate entry \*/

printf("Miss ");

i = mru[7];

tag[i] = addr;

mruUpdate(i);

}

for (i = 0; i < 8; i++)

printf("0x%08x ", tag[i]);

for (i = 0; i < 8; i++)

printf("%d ", mru[i]);

printf("\n");

}

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

}

**2) Fully-Associative Cache (Tag 16bit):**

#include <stdio.h>

int tag[16];

int mru[16] = {15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0};

void mruUpdate(int index){

int i;

for (i = 0; i < 16; i++)

if (mru[i] == index)

break;

while (i > 0) {

mru[i] = mru[i-1];

i--;

}

mru[0] = index;

}

int main()

{

int addr;

int i, j, t;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

while (fscanf(fp, "%x", &addr) > 0) {

/\* simulate fully associative cache with 8 words \*/

accesses += 1;

printf("%4d: 0x%08x ", accesses, addr);

for (i = 0; i < 16; i++) {

if (tag[i] == addr) {

hits += 1;

printf("Hit%d ", i);

mruUpdate(i);

break;

}

}

if (i == 16) {

/\* allocate entry \*/

printf("Miss ");

i = mru[15];

tag[i] = addr;

mruUpdate(i);

}

for (i = 0; i < 16; i++)

printf("0x%08x ", tag[i]);

for (i = 0; i < 16; i++)

printf("%d ", mru[i]);

printf("\n\n");

}

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

}

**3) 2-way set-associative cache (Tag 8bit)**

#include <stdio.h>

int tag[2][4];

int mru[4] = {1,1,1,1};

void mruUpdate(int index){

int i;

for (i = 0; i < 4; i++)

if (mru[i] == index)

break;

while (i > 0) {

mru[i] = mru[i-1];

i--;

}

mru[0] = index;

}

int main(){

int addr;

int i, j;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

while (fscanf(fp, "%x", &addr) > 0) {

printf("%3d: 0x%08x ", accesses, addr);

for( i=0; i<2; i++){

accesses += 1;

for(j=0; j<4; j++){

if (tag[i][j] ==addr){

hits += 1;

printf("Hit%d ", i);

mruUpdate(i);

break;

}

} if(tag[i][j]==addr){

printf("Hit%d ", i);

}else {

printf("Miss");

tag[i][j] = addr;

//hits +=1;

}

} for(i = 0; i < 2; i++){

for(j=0; j<4; j++){

printf("0x%08x ", tag[i][j]);

}

}

for (i = 0; i < 2; i++)

printf("%d ", mru[i]);

printf("\n");

}

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

}

**4) 2-way set-associative cache (Tag 16bit)**

#include <stdio.h>

int tag[2][8]; // ERROR DECIDE SIZE

int mru[8] = {1,1,1,1,1,1,1,1};

void mruUpdate(int index){

int i;

for (i = 0; i < 8; i++)

if (mru[i] == index)

break;

while (i > 0) {

mru[i] = mru[i-1];

i--;

}

mru[0] = index;

}

int main(){

int addr;

int i, j;

int hits, accesses;

FILE \*fp;

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

hits = 0;

accesses = 0;

while (fscanf(fp, "%x", &addr) > 0) {

printf("%4d: 0x%08x ", accesses, addr);

for( i=0; i<2; i++){

accesses += 1;

for(j=0; j<8; j++){

if (tag[i][j] ==addr){

hits += 1;

printf("Hit%d ", i);

mruUpdate(i);

break;

}

} if(tag[i][j]==addr){

printf("Hit%d ", i);

}else {

printf("Miss");

tag[i][j] = addr;

//hits +=1;

}

} for(i = 0; i < 2; i++){

for(j=0; j<8; j++){

printf("0x%08x ", tag[i][j]);

}

}

for (i = 0; i < 4; i++)

printf("%d ", mru[i]);

printf("\n");

}

printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits, accesses, ((float)hits)/accesses);

close(fp);

}

**OUTPUT:**

**1) Fully-Associative Cache (Tag 8bit):**

Graphical user interface, text, application

Description automatically generated

**2) Fully-Associative Cache (Tag 16bit):**

Graphical user interface, text, application

Description automatically generated

**3) 2-way set-associative cache (Tag 8bit)**

Graphical user interface, application

Description automatically generated

**4) 2-way set-associative cache (Tag 16bit)**

Graphical user interface, text

Description automatically generated

**CONCLUSION:**

In this experiment, I implemented Fully Associative Mapping and Set Associative Mapping for 8bits and 16bits tag using UPC miniMIPS Simulator and C program. First I generated a trace.txt file using UNC miniMIPS Simulator. Then coded Associative Mapping and Set Associative Mapping for 8bits and 16bits tag in C language. For Fully Associative Mapping with tag size 8bits, the Hit Ratio was 0.737 and with tag size 16bits, the hit ratio was 0.74. Similarly, for 2 way Set Associative, the hit ratio was 0.5. Finally, associative mapping is fast and easy to implement, however it is expensive to implement as it requires storing of addresses along with the data. The placement policy is a trade-off between direct-mapped and fully associative cache however Set Associative offers the flexibility of using replacement algorithms if a cache miss occurs but the placement policy will not effectively use all the available cache lines in the cache and suffers from conflict miss.

**EXPERIMENT 7**

**DATE OF PERFORMANCE:** 11/11/2021 **DATE OF SUBMISSSION:** 18/11/2021

**AIM:** Study MIPS Assembly Language Programming using MIPS simulator and implement the following:

1. To add 10 numbers
2. To print message "Hello MIPS".
3. To Reverse the input string ("ABC" - "CBA").

**THEORY:**

**MIPS Assembly Language Programming:**

MIPS is an acronym for Microprocessor without Interlocked Pipeline Stages. It is a reduced instruction set architecture developed by an organization called MIPS Technologies. The MIPS assembly language is a very useful language to learn because many embedded systems run on the MIPS processor. Knowing how to code in this language brings a deeper understanding of how these systems operate on a lower level.

A MAL program is divided into two types of sections:

**Data sections** specify actions to be taken during assembly. Usually declare memory variables used by the program.

**Text sections** define sequences of instructions executed by the program at run time.

Syntax: *label operation operand\_list # comment*

**Looping in MIPS:**

**for**

li $t0, 10 # t0 is a constant 10

li $t1, 0 # t1 is our counter (i)

loop:

beq $t1, $t0, end # if t1 == 10 we are done

loop body

addi $t1, $t1, 1 # add 1 to t1

j loop # jump back to the top

end:

**while:**

top\_while:

t0 = evaluate Cond

beqz $t0,end\_while

execute Statements

**j** top\_while

end\_while:

**CODE:**

**1)Program to add 10 nos.**

.data

newline: .asciiz "\n"

.text

main:

li $t0,0

li $t2,5

loop:

bgt $t0, 45, exit

addi $t0, $t0, 5

li $v0, 1

move $a0, $t0

syscall

li $v0, 4

la $a0, newline

syscall

j loop

exit:

li $v0,10

syscall

Graphical user interface, application

Description automatically generated**OUTPUT**

**2) To print hello message**

.data

myMessage \_: .asciiz "Hello MIPS"

.text

main:

li $v0, 4

myMessage syscall

li $v0, 10

syscall

Graphical user interface, application

Description automatically generated**OUTPUT**

**3) To reverse the input string**

.data

input: .space 256

output:.space 256

.text

.globl main

main:

li $v0, 8

la $a0, input

li $a1, 256

syscall

li $v0, 4

la $a0, input

syscall

jal strlen

add $t1, $zero, $v0

add $t2, $zero, $a0

add $a0, $zero, $v0

li $v0, 1

syscall

reverse:

li $t0, 0

li $t3, 0

reverse\_loop:

add $t3, $t2, $t0

lb $t4, 0($t3)

beqz $t4, exit

sb $t4, output($t1)

subi $t1, $t1, 1

addi $t0, $t0, 1

j reverse\_loop

exit:

li $v0, 4

la $a0, output

syscall

li $v0, 10

syscall

strlen:

li $t0, 0

li $t2, 0

strlen\_loop:

add $t2, $a0, $t0

lb $t1, 0($t2)

beqz $t1, strlen\_exit

addiu $t0, $t0, 1

j strlen\_loop

strlen\_exit:

subi $t0, $t0, 1

add $v0, $zero, $t0

add $t0, $zero, $zero

jr $ra

**OUTPUT:**

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

**CONCLUSION:**

In this experiment, I implemented 3 programs in MIPS Assembly Language (MAL) viz, addition of 10 numbers, print “Hello MIPS” and reverse the input string (‘MEITH’ -> ‘HTIEM’) using MIPS stimulator. The MAL is divided into 2 sections: Data and Text section. The Data section usually have declaration of memory variables used by the program and the Text sections define sequences of instructions executed by the program at run time. In this experiment I learnt how to implement loops in the MAL, how to take inputs and print the outputs and convert a complex expression into an assembly language.

**EXPERIMENT 8**

**DATE OF PERFORMANCE:** 18/11/2021 **DATE OF SUBMISSSION:** 02/12/2021

**AIM:** Implement 8086 based Assembly programs.

**THEORY:**

Intel 8086 is built on a single semiconductor chip and packaged in a 40-pin IC package. The type of package is DIP (Dual Inline Package). Intel 8086 uses 20 address lines and 16 data- lines. It can directly address up to 220 = 1 Mbyte of memory. 8086 is designed to operate in two modes, i.e., Minimum and Maximum mode.

Table

Description automatically generated

It consists of a powerful instruction set, which provides operation like division and multiplication very quickly.

8086 microprocessor supports 8 types of instructions:

* Data Transfer Instructions
* Arithmetic Instructions
* Bit Manipulation Instructions
* String Instructions
* Program Execution Transfer Instructions (Branch & Loop Instructions)
* Processor Control Instructions
* Iteration Control Instructions
* Interrupt Instructions

**CODE:**

**1) Program to add two word length numbers**

OPR1: DW 0x6969 ; declare first number

OPR2: DW 0x0420 ; declare second number

RESULT: DW 0 ; declare place to store result

; actual entry point of the program

start:

MOV AX, word OPR1 ; move first number to AX

MOV BX, word OPR2 ; move second number to BX

CLC ; clear the carry flag

ADD AX, BX ; add BX to AX

MOV DI, OFFSET RESULT ; move offset of result to DI

MOV word [DI], AX ; store result

print reg ; print result

**OUTPUT:**

A picture containing text, indoor, screenshot, several

Description automatically generated

**2) A Program to move data from one segment to another**

SET 0 ; set address for segment 1

src:DB 0x3 ; store data

DB 0x5

DB 0x7

SET 0x1 ; set addresss for segment 2

dest:DB [0,3] ; store data

; actual entry point of the program

start:

print mem 0:8 ; print initial state of segment 1

print mem 0x10:8 ; print initial state of segment 2

MOV AX, 0 ; move address of seg1

MOV DS,AX ; to ds

MOV AX , 0x1 ; move address of seg2

MOV ES,AX ; to es

MOV SI, OFFSET src ; move offset of source data

MOV SI, OFFSET dest ; move offset of destination data

MOV CX, 0x3 ; move number of data items

print reg ; print state of registers

\_loop:

mov AH, byte DS[SI] ; move one byte from source to ah

mov byte ES[DI],AH ; move ah to destination

inc SI

inc DI

dec CX ; decrement count

jnz \_loop ; if count is not zero jump back

print mem 0:8 ; print final state of segment 1

print mem 0x10:8 ; print final state of segment 2

**OUTPUT:**

Table

Description automatically generated

**3) Program to calculate factorial using looping**

NUM: DW 0x6 ; calculate factorial of 6

RESULT: DW 0 ; place to store the reult

; actual entry point of the program

start:

MOV CX,word NUM ; move number into cx

MOV AX, 0x1 ; initialize accumulator with 1

NOTZEROLOOP: ; label to jump back to

MUL CX ; multiple by the number

DEC CX ; decrement the number

JNZ NOTZEROLOOP ; if not zero jump back

MOV word RESULT,AX ; store the result in memory

print reg ; print registers

**OUTPUT:**

Table

Description automatically generated

**4) Program to show use of interrupts**

hello: DB "Hello World" ; store string

; actual entry point of the program, must be present

start:

MOV AH, 0x13 ; move BIOS interrupt number in AH

MOV CX, 11 ; move length of string in cx

MOV BX, 0 ; mov 0 to bx, so we can move it to es

MOV ES, BX ; move segment start of string to es, 0

MOV BP, OFFSET hello ; move start offset of string in bp

MOV DL, 0 ; start writing from col 0

int 0x10 ; BIOS interrupt

**OUTPUT:**

Shape, rectangle

Description automatically generated with medium confidence

Table

Description automatically generated

**5) Program to show use of interrupts**

hello: DB "Hello World" ; store string

; actual entry point of the program

start:

MOV AH, 0x13 ; move BIOS interrupt number in AH

MOV CX, 12 ; move length of string in cx

MOV BX, 0 ; mov 0 to bx, so we can move it to es

MOV ES, BX ; move segment start of string to es, 0

MOV BP, OFFSET hello ; move start offset of string in bp

MOV DL, 0 ; start writing from col 0

int 0x10 ; BIOS interrupt

**OUTPUT:**

Graphical user interface, table

Description automatically generated

**6) Program to calculate LCM and GCD of two numbers**

no1: dw 0x6 ; number 1

no2: dw 0x5 ; number 2

gcd: dw 0 ; place to store gcd

lcm: dw 0 ; place to store lcm

; actual entry point of the program

start:

mov ax, word no1 ; move number 1 in accumulatore

mov bx, word no2 ; move number 2 in register BX

loop0: mov dx, 0x0 ; place to loop back

; cannot use 'loop' as label, as loop is an opcode which will give error when used with jumps

div bx ; divide accumulator by bx

mov ax, bx

mov bx, dx

cmp bx, 0x0 ; check if bx is 0

jnz loop0 ; if not loop back

mov word gcd, ax ; store gcd

mov cx, ax ; move ax in cx

mov ax, word no1 ; move number 1 in accumulatore

mov bx, word no2 ; move number 2 in register BX

mul bx ; multiply accumulator by BX

div cx ; divide accumulator by CX

mov word lcm, ax ; store lcm

print mem :16 ; print memory

**OUTPUT:**

Graphical user interface, table

Description automatically generated

**CONCLUSION:**

In this experiment, I implemented 8086 microprocessor’s assembly language based programs. The codes were run on an online 8086 emulator.The programs were to add two word length numbers, to calculate LCM and GCD of two numbers, to transfer the data, to calculate the factorial using loop in 8086 assembly instruction set and programs to implement interrupts.

**EXPERIMENT 9**

**DATE OF PERFORMANCE:** 02/12/2021 **DATE OF SUBMISSSION:** 09/12/2021

**AIM:** To study and implement Macros and Dos Interrupt in Assembly Language Programming.

1) To implement Macros for calculating Factorial of a number

2) To calculate and display sum of 2 user entered inputs using DOS interrupts

**THEORY:**

**Macros:**

A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program. Macros are useful for the following purposes:

* To simplify and reduce the amount of repetitive coding
* To reduce errors caused by repetitive coding
* To make an assembly program more readable.

A macro consists of name, set of formal parameters and body of code. The use of macro name with set of actual parameters is replaced by some code generated by its body. This is called macro expansion.

Macros allow a programmer to define pseudo operations, typically operations that are generally desirable, are not implemented as part of the processor instruction, and can be implemented as a sequence of instructions. Each use of a macro generates new program instructions, the macro has the effect of automating writing of the program.

Macros can be defined used in many programming languages, like C, C++ etc. Example macro in C programming.Macros are commonly used in C to define small snippets of code. If the macro has parameters, they are substituted into the macro body during expansion; thus, a C macro can mimic a C function. The usual reason for doing this is to avoid the overhead of a function call in simple cases, where the code is lightweight enough that function call overhead has a significant impact on performance.

**DOS Interrupts:**

The interrupt types 20h-3Fh are serviced by DOS routines that provide high-level service to hardware as well as system resources such as files and directories. The most useful is INT 21H, which provides many functions for doing keyboard, video, and file operations.

|  |  |
| --- | --- |
| **Function Number** | **Description** |
| AH=01h | READ CHARACTER FROM STANDARD INPUT, WITH ECHO |
| AH=02h | WRITE CHARACTER TO STANDARD OUTPUT |
| AH=05h | WRITE CHARACTER TO PRINTER |
| AH=06h | DIRECT CONSOLE OUTPUT |
| AH=07h | DIRECT CHARACTER INPUT, WITHOUT ECHO |
| AH=08h | CHARACTER INPUT WITHOUT ECHO |
| AH=09h | WRITE STRING TO STANDARD OUTPUT |
| AH=0Ah | BUFFERED INPUT |
| AH=0Bh | GET STDIN STATUS |
| AH=0Ch | FLUSH BUFFER AND READ STANDARD INPUT |
| AH=0Dh | DISK RESET |
| AH=0Eh | SELECT DEFAULT DRIVE |
| AH=19h | GET CURRENT DEFAULT DRIVE |
| AH=25h | SET INTERRUPT VECTOR |
| AH=2Ah | GET SYSTEM DATE |
| AH=2Bh | SET SYSTEM DATE |
| AH=2Ch | GET SYSTEM TIME |
| AH=2Dh | SET SYSTEM TIME |
| AH=2Eh | SET VERIFY FLAG |
| AH=30h | GET DOS VERSION |
| AH=35h | GET INTERRUPT VECTOR |
| AH=36h | GET FREE DISK SPACE |
| AH=39h | "MKDIR" - CREATE SUBDIRECTORY |
| AH=3Ah | "RMDIR" - REMOVE SUBDIRECTORY |
| AH=3Bh | "CHDIR" - SET CURRENT DIRECTORY |
| AH=3Ch | CREATE OR TRUNCATE FILE |
| AH=3Dh | "OPEN" - OPEN EXISTING FILE |
| AH=3Eh | "CLOSE" - CLOSE FILE |
| AH=3Fh | "READ" - READ FROM FILE OR DEVICE |
| AH=40h | "WRITE" - WRITE TO FILE OR DEVICE |
| AH=41h | "UNLINK" - DELETE FILE |
| AH=42h | "LSEEK" - SET CURRENT FILE POSITION |
| AH=43h | GET FILE ATTRIBUTES |
| AH=47h | "CWD" - GET CURRENT DIRECTORY |
| AH=4Ch | "EXIT" - TERMINATE WITH RETURN CODE |
| AH=4Dh | GET RETURN CODE (ERRORLEVEL) |
| AH=54h | GET VERIFY FLAG |
| AH=56h | "RENAME" - RENAME FILE |
| AH=57h | GET/SET FILE'S DATE AND TIME, GET EXTENDED ATTRIBUTES FOR FILE |

**CODE:**

**1) Program to calculate factorial using Macros.**

NUM: DW 0x6

RESULT: DW 0

MACRO fact(no) -> MUL word no <-

start:

MOV AX, 0x0001

NOTZEROLOOP:

fact(NUM)

DEC word NUM

JNZ NOTZEROLOOP

MOV word RESULT,AX

print mem :16

**OUTPUT:**

A screenshot of a computer

Description automatically generated with medium confidence

**2) Program to calculate the sum of 2 number using DOS Interrupts.**

DATA SEGMENT

NUM1 DB ?

NUM2 DB ?

RESULT DB ?

MSG1 DB 10,13,"ENTER FIRST NUMBER TO ADD : $"

MSG2 DB 10,13,"ENTER SECOND NUMBER TO ADD : $"

MSG3 DB 10,13,"RESULT OF ADDITION IS : $"

ENDS

CODE SEGMENT

ASSUME DS:DATA, CS:CODE

START:

MOV AX,DATA

MOV DS,AX

LEA DX,MSG1

MOV AH,9

INT 21H

MOV AH,1

INT 21H

SUB AL,30H

MOV NUM1,AL

LEA DX,MSG2

MOV AH,9

INT 21H

MOV AH,1

INT 21H

SUB AL,30H

MOV NUM2,AL

ADD AL,NUM1

MOV RESULT,AL

MOV AH,0

AAA

ADD AH,30H ADD AL,30H

MOV BX,AX

LEA DX,MSG3

MOV AH,9

INT 21H

MOV AH,2

MOV DL,BH

INT 21H

MOV AH,2

MOV DL,BL

INT 21H

MOV AH,4CH

INT 21H

ENDS

END START

**OUTPUT:**

Text

Description automatically generated

**CONCLUSION:**

In this experiment, I implemented Macros and Dos Interrupts in Assembly language Program. Firstly, I learnt about Macros and it’s implementation in Assembly Level Language. Macros help reduce the code repetition and are typically faster than functions as they don't involve actual function call overhead. Secondly, I learnt about Dos Interrupt and how to implement them in Assembly Language program. Interrupts is a condition that halts the microprocessor temporarily to work on a different task and then return to its previous task. The most useful is INT 21H, which provides many functions for doing keyboard, video, and file operations.

**EXPERIMENT 10**

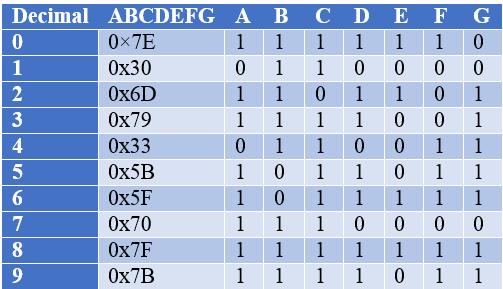
**DATE OF PERFORMANCE:** 09/12/2021 **DATE OF SUBMISSSION:** 16/12/2021

**AIM:** Write a program using ALP to stimulate 8051 Microcontroller interfacing Seven segment display. Display your SAP ID using this tool. (Convert your SAP ID in to Hex and then use 7segment display to Glow your SAP ID)

**THEORY:**

**8085 Microcontroller:**

Digit pattern of a seven segment LED display is simply the different logic combinations of its terminals ‘a’ to ‘h’ to display different digits and characters. For example, if you want to display the digit 3 on seven segment then you need to glow the segments a, b, c, d and g, having a binary pattern: 3 → 1 1 1 1 0 0 1, in hexadecimal converts to 0x79. The table below, demonstrates the Hex decimal values that we need to send to the Display from PORT selected.



**Applications**

* Seven segments are widely used in digital clocks to display the time.
* These are used in electronic meters for displaying the numerical information.
* Used in Instrument panels.
* Used in digital readout displays.

**Limitations**

* The complexity is increased to display large information.
* It is not possible to display the symbols on seven segment.

Diagram, schematic

Description automatically generated

**CODE:** To display the last bits of the SAP ID (4190068 → 3FEF74)

MOV P0,#79h

MOV P0,#47h

MOV P0,#4Fh

MOV P0,#47h

MOV P0,#70h

MOV P0,#33h

**OUTPUT:**

Graphical user interface, chart, application

Description automatically generated

Graphical user interface, application

Description automatically generatedGraphical user interface

Description automatically generatedGraphical user interface, application

Description automatically generated

Graphical user interface

Description automatically generatedGraphical user interface, application

Description automatically generated

**CONCLUSION:**

In this experiment, I learnt interfacing 7 segment display with 8051 microcontroller. The 7 segment display have 7 segments a, b, c, d, e, f, g and the respective segments need to be lit in order to display a digit. Each digit has it’s own pattern for example, 3 → 1 1 1 1 0 0 1, in hexadecimal converts to 0x79. Thus, in the microcontroller we feed 79H to the port and the number is displayed. In this experiment I displayed the hex value of the last bits of my SAP ID (4190068 → 3FEF74)