CS2610 Computer Organization Laboratory

Lab - 10

Objective:

In this lab you will use the C programming language to write simulators for several cache organizations and evaluate their performance. You are provided with a series of memory addresses which are to be accessed in trace.txt file.

Problem:

(1) Direct mapped cache:

The following code fragment simulates a direct-mapped cache with 8 lines of 1 word.

```
#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;
    while (fscanf(fp, "%x", &addr) > 0) {
        /* simulate a direct-mapped cache with 8 words */
        accesses += 1;
        printf("%3d: 0x%08x ", accesses, addr);
        i = (addr >> 2) & 7;
        t = addr | 0x1f;
        if (tag[i] == t) {
            hits += 1;
            printf("Hit%d ", i);
        } else {
            /* allocate entry */
            printf("Miss ");
            tag[i] = t;
        }
        for (i = 0; i < 8; i++)
            printf("0x%08x ", tag[i]);
        printf("\n");
    }
      printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits,
accesses, ((float)hits)/accesses);
    close(fp);
}
```

Compile and execute the direct-mapped cache simulator given above. Put appropriate comments for the code sections and report the final number of hits, accesses and hit rate output by the code.

(2) Fully Associative Cache:

The following code fragment simulates a fully associative cache with 8 lines each of 1 word, with least recently used (LRU) cache replacement algorithm. The 'mru' in the code fragment represents most recently used.

```
include <stdio.h>

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

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, 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) {

               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); }
```

Compile and execute the fully associative cache simulator given above. Put appropriate comments for the code sections and report the final number of hits, accesses and hit rate output by the code.

(3) Two Way Set Associative Cache:

Use the following code fragment as a basis for implementing a 2-way set associative cache with LRU replacement.

```
#include <stdio.h>

  int tag[2][4];
  int mru[4] = {1,1,1,1};

   int main( ){
    int addr;
    int hits, accesses;
    FILE *fp;

    fp = fopen("trace.txt", "r");
    hits = 0;
    accesses = 0;
    while (fscanf(fp, "%x", &addr) > 0) {
     /********* YOUR CODE HERE ************/
    }
   printf("Hits = %d, Accesses = %d, Hit ratio = %f\n", hits,
accesses, ((float)hits)/accesses);
     close(fp);
 }
```

Where 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.

Finish the supplied program for simulating a 2-way set associative cache with LRU replacement using the trace in the "trace.txt" file. Compile and execute it. Put appropriate comments for the code sections and report the final number of hits, accesses and hit rate output by the code.

How will you approach for a 4 way set associative cache?

Post-lab:

Submit a post-lab report with your verified outputs. The report should include your well commented code snippets, snapshots of the output indicating the number of cache hits.
All your submissions should be clear and concise.