

OREGON STATE UNIVERSITY

CS 472 - COMPUTER ARCHITECTURE

SPRING 2014

Lab 5 - The Memory Hierarchy and Endian-Neutral Programming

Author:

Drake Bridgewater
Ryan Phillips

Professor:

Kevin McGRATH

May 31, 2014

Part 1

Code

cache_size.c

```
1      it needs to be tested, and probably
      fixed/modified
2  */
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #define KB 1024
9  #define MB 1024 * 1024
10
11 int main() {
12     unsigned int steps = 256 * 1024 *
13         1024;
14     static int arr[4 * 1024 * 1024];
15     int lengthMod;
16     unsigned int i;
17     double timeTaken;
18     clock_t start;
19     int sizes[] = {
20         1 * KB, 4 * KB, 8 * KB, 16 * KB,
21         32 * KB, 64 * KB, 128 * KB, 256
22         * KB,
23         512 * KB, 1 * MB, 1.5 * MB, 2 * MB
24         , 2.5 * MB, 3 * MB, 3.5 * MB, 4
25         * MB
26     };
27     int results[sizeof(sizes)/sizeof(int)
28         ];
29     int s;
30
31     // for each size to test for ...
32     for (s = 0; s < sizeof(sizes)/sizeof(
33         int); s++) {
34         lengthMod = sizes[s] - 1;
35         start = clock();
36         for (i = 0; i < steps; i++) {
37             arr[(i * 16) & lengthMod] *= 10;
38             arr[(i * 16) & lengthMod] /=
39                 10;
40         }
41
42         timeTaken = (double)(clock() - start
43             )/CLOCKS_PER_SEC;
44         printf("%d, %.8f \n", sizes[s] /
45             1024, timeTaken);
46     }
47     return 0;
48 }
```

Output

cache_size.txt

```
1 1, 16.52000000
2 4, 16.24000000
3 8, 16.31000000
4 16, 17.85000000
5 32, 18.74000000
6 64, 36.05000000
7 128, 78.88000000
8 256, 91.57000000
9 512, 92.83000000
10 1024, 92.88000000
11 1536, 92.84000000
12 2048, 92.94000000
13 2560, 92.83000000
14 3072, 92.86000000
15 3584, 92.83000000
16 4096, 92.86000000
```

From the output our program produces while running on a Beaglebone Black we can see that the size of the **L1 cache is about 32K** as the jump in time happened immediately after that. As for the **L2 cache it is likely to be around 128K** as there is no jump in timing after that.

Part 2

endian_neutral.c

```
1 #include <stdio.h>
   #include <stdint.h>
3
   #define IS_BIG_ENDIAN (*(uint16_t *)"\0\xff" < 0x100)
5
   int main(int argc, char **argv)
7 {
       printf("PART2:\n\n%d\n", IS_BIG_ENDIAN); //0 if false
9
       short val;
       char *p_val;
       p_val = (char *) &val;
13
       if (IS_BIG_ENDIAN){
15 printf("\t\tBIG ENDIAN\n\n");
       p_val[0] = 0x12;
17 p_val[1] = 0x34;
       } else {
19 printf("\t\tlittle ENDIAN\n\n");
       p_val[0] = 0x34;
21 p_val[1] = 0x12;
       }
23 printf("%x\n", val);

25 return 0;
}
```

Part 3

endian_neutral.c

```
#include <stdio.h>
2 #include <stdint.h>

4 #define IS_BIG_ENDIAN (*(uint16_t *)"\0\xff" < 0x100)

6 int main(int argc, char **argv)
{
8     printf("PART2:\n\n%d\n", IS_BIG_ENDIAN); //0 if false

10     short val;
    char *p_val;
12     p_val = (char *) &val;

14     if (IS_BIG_ENDIAN){
        printf("\t\tBIG ENDIAN\n\n");
16         p_val[0] = 0x12;
        p_val[1] = 0x34;
18     } else {
        printf("\t\tlittle ENDIAN\n\n");
20         p_val[0] = 0x34;
        p_val[1] = 0x12;
22     }
        printf("%x\n", val);
24
        return 0;
26 }
```
