Name:

CSc 656 Final
(5/16/2011)

Problem 1 (25 points):
Consider the following C/C++ loop, executing on a system with a cache.

```
int i,x[50], y[50];

for (i=0;i<12;i++)
{
  sum = sum + x[i];
  sum = sum + y[i];
}

for (i=0;i<12;i++)
{
  prod = prod*x[i];
  prod = prod*y[i];
}
```

&x[0] is 0x5e4, &y[0] is 0x9c0. The cache is direct-mapped with cache size 512 bytes and block size of 16 bytes. i, prod and sum are allocated to registers. Trace through the loop and count the number of hits and misses. Show work clearly for each iteration; no points will be awarded for lucky guesses.

ANS:

#blocks = 512/16 = 32
index is 5 bits, offset is 4 bits

tag	index	offset		
010	1 1110	0100	x[0] 0x5e4	m
100	1 1100	0000	y[0] 0x9c0	m
010	1 1110	1000	x[1] 0x5e8	h
100	1 1100	0100	y[1] 0x9c4	h
010	1 1110	1100	x[2] 0x5ec	h
100	1 1100	1000	y[2] 0x9c8	h
010	1 1111	0000	x[3] 0x5f0	m
100	1 1100	1100	y[3] 0x9cc	h
010	1 1111	0100	x[4] 0x5f4	h
100	1 1101	0000	y[4] 0x9d0	m
010	1 1111	1000	x[5] 0x5f8	h
100	1 1101	0100	y[5] 0x9d4	h
010	1 1111	1100	x[6] 0x5fc	h
100	1 1101	1000	y[6] 0x9d8	h
011	0 0000	0000	x[7] 0x600	m
100	1 1101	1100	y[7] 0x9dc	h
011	0 0000	0100	x[8] 0x604	h
100	1 1110	0000	y[8] 0x9e0	m
011	0 0000	1000	x[9] 0x608	h
100	1 1110	0100	y[9] 0x9e4	h
011	0 0000	1100	x[10] 0x60c	h
100	1 1110	1000	y[10] 0x9e8	h
011	0 0001	0000	x[11] 0x610	m

```
100    1 1110        1100          y[11] 0x9ec h

7 misses, 17 hits

****

010    1 1110        0100          x[0] 0x5e4   m
100    1 1100        0000          y[0] 0x9c0   h

010    1 1110        1000          x[1] 0x5e8   h
100    1 1100        0100          y[1] 0x9c4   h

010    1 1110        1100          x[2] 0x5ec   h
100    1 1100        1000          y[2] 0x9c8   h

010    1 1111        0000          x[3] 0x5f0   h
100    1 1100        1100          y[3] 0x9cc   h

010    1 1111        0100          x[4] 0x5f4   h
100    1 1101        0000          y[4] 0x9d0   h

010    1 1111        1000          x[5] 0x5f8   h
100    1 1101        0100          y[5] 0x9d4   h

010    1 1111        1100          x[6] 0x5fc   h
100    1 1101        1000          y[6] 0x9d8   h

011    0 0000        0000          x[7] 0x600   h
100    1 1101        1100          y[7] 0x9dc   h

011    0 0000        0100          x[8] 0x604   h
100    1 1110        0000          y[8] 0x9e0   m

011    0 0000        1000          x[9] 0x608   h
100    1 1110        0100          y[9] 0x9e4   h

011    0 0000        1100          x[10] 0x60c h
100    1 1110        1000          y[10] 0x9e8 h

011    0 0001        0000          x[11] 0x610 h
```

100 1 1110 1100 y[11] 0x9ec h

2 misses, 22 hits

Total: 9 misses, 39 hits

Problem 2 (20 points):
Suppose we have a direct-mapped data cache with a 4-bit offset, 8-bit index, and 20-bit tag, operating in combination with a 4-entry fully-associative victim cache, similar to your Project 2. Some selected slots from the direct-mapped cache, and the entire victim cache, are shown in the figure below.

For each block in the data cache, VALID is the valid bit, TAG is the stored tag, DIRTY is the dirty bit, and USED is a timestamp for when the block was last used.

For each block in the victim cache, VALID, DIRTY and USED are defined as in the data cache. For the victim cache TAG, we store both the TAG and INDEX for that block, as computed for the data cache (since it's fully associative). Hence, the first slot in the victim cache came from index 0x20 of the data cache; its tag in the data cache was 0x10082. The victim cache uses LRU replacement, based on the USED field.

From the cache states shown in the figure, we have three accesses, one after another:

a write to 0x1001c438 (timestamp 291)
a write to 0x7ffff420 (timestamp 292)
a read to 0x40000454 (timestamp 293)

Mark the changes in the cache and victim cache, after the three accesses. Make sure you update all the fields that will change as a result of the two accesses.

Direct-mapped cache (excerpt)

INDEX	VALID	TAG	DIRTY	USED
0x42	1	0x1001c	1	211
0x43	1	0x1001c	0	256
0x44	1	0x7fff8	0	101
0x45	1	0x40000	1	13

Victim cache:

VALID	TAG	INDEX	DIRTY	USED
1	0x10082	0x24	0	47
1	0x10024	0x44	1	100
1	0x7ffff	0x42	0	78
1	0x10010	0x43	1	231

ANS: *[Show work and mark changes below…]*

Direct-mapped cache (excerpt)

INDEX	VALID	TAG	DIRTY	USED
0x42	1	~~0x1001c~~ 0x7ffff	~~1~~ 0	~~211~~ 292
0x43	1	0x1001c	~~0~~ 1	~~256~~ 291
0x44	1	0x7fff8	0	101
0x45	1	0x40000	1	~~13~~ 293

Victim cache:

VALID	TAG	INDEX	DIRTY	USED
1	0x10082	0x24	0	47
1	0x10024	0x44	1	100
1	~~0x7ffff~~ 0x1001c	0x42	~~0~~ 1	~~78~~ 211
1	0x10010	0x43	1	231

a write to 0x1001c438 (timestamp 291)
index = 0x43, tag = 0x1001c
write hit, turn on DIRTY bit, update USED

a write to 0x7ffff420 (timestamp 292)
index = 0x42, tag = 0x7ffff
miss in cache, hit in victim cache
swap victim cache block with cache block, update DIRTY and USED

a read to 0x40000454 (timestamp 293)
index = 0x45, tag = 0x40000
hit in cache, update USED

Problem 3 (10 points):
Suppose you are writing a cache simulator for a direct-mapped cache. These variables are given:

```
unsigned int address;
    // current memory address to be processed
int bytesPerBlock; // number of bytes per block, power of 2
int blocksInCache; // number of blocks in cache, power of 2
int tag, index;
```

Suppose the code to calculate bytesPerBlock and blocksInCache are given but not shown. The code to read an address from a trace file is also given but not shown. You are also given this log2() function to use (if you wish):

int log2(int arg)

log2() returns the log (base 2) of arg; arg is restricted to powers of 2.

Show the code to calculate the tag and index. You may write C++ or Java code, but are not allowed to use Java/C++ String utilities.

ANS: see Project 2 code

Problem 4 (15 points):
An array x[] with N elements has been initialized for you. Complete the program on the next page, using the pthreads library to do the following:

fork 2 threads, 0 and 1
thread 0 computes the sum of x[i] for even i (x[0], x[2], x[4], etc)
thread 1 computes the sum of x[i] for odd i (x[1], x[3], x[5], etc)
thread 1 prints the sum it computed
thread 0 prints the sum it computed

You must make sure thread 1 prints its sum (of x[i] for odd i) first. You don't have to check for errors.

Useful pthreads library info:

```
pthread_t pt;  // pt is a POSIX thread struct

/* create a thread, return 0 on success, non-zero code on failure */
int pthread_create(pthread_t *t, /* pointer to pthread_t struct */
                   NULL, /* attribute, usually NULL */
                   void * (*start_routine)(void *),
                   /* pointer to entry-point function of thread */
                   void arg); /* arg for entry-point function */

/* wait for thread with struct someThread to finish */
/* return 0 on success, non-zero code on failure */
int pthread_join(
     pthread_t someThread, /* thread struct */
  void **thread_return /* NULL for now */
}

/* terminate the calling thread (i.e., self) */
int pthread_exit(
     void *retval   /* return value of thread */
}

/* send termination request to thread with struct someThread */
int pthread_cancel(
     pthread_t someThread /* thread struct */
}
```

```
/* insert your variables and code below… */

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

int x[2000];
pthread_t thread0, thread1;


int main() {
   int i;

   for (i=0; i<2000; i++) {
      x[i] = random();
   }
   pthread_create(&thread0, NULL, pw, (void *) 0);
   pthread_create(&thread1, NULL, pw, (void *) i);
   allDone = 1;

   pthread_join(thread0, NULL);

}

void *pw(void *arg) {
  int id = (int ) arg;
  int i;
  int sum = 0;

  if (id == 0) {
    for (i=0; i<2000; i+=2) {
      sum += x[i];
    }
    while (allDone == 0);
    pthread_join(thread1, NULL);
    printf("sum of evens = %d\n", sum);
  }
  else {
    for (i=1; i<2000; i+=2) {
      sum += x[i];
    }
    printf("sum of odds = %d\n", sum);
  }


}
```

Short question 1 (10 points):
Suppose we are measuring the total access time for data accesses, for the benchmark BOO running on a MIPS CPU. The instruction count is IC, with 45% ALU, 30% loads, 10% stores, and 15% branches. The data miss rate is 4%, hits take 2 cycles, and the miss penalty is 50 cycles. What is the total access time (in cycles) for data accesses only? (You don't have to show a single number as the solution; just show an equation that will evaluate to the answer.)

Total data access = IC * (.3+.1) = .4IC
Each data access takes 2 cycles (at least)
Each data miss takes an additional 50 cycles
There are .4IC * .04 misses.

Total data access time = 2(.4)IC + .04(.4)IC * 50

Short question 2 (5 points):
For a 8000 rpm disk drive, what is the average rotational latency in milliseconds?

8000 rotations in 60 seconds
1 rotation in 60 / 8000 seconds = 60 * 1000 / 8000 ms
average rotational latency = time for .5 rotations = .5 * 60 * 1000/8000 ms

Short Question 3 (5 points):
Most commercial CPUs have split L1 instruction/data caches; however, there is usually a single unified L2 cache. Why? Explain in 40 or fewer words.

L1 caches are split because a pipeline often needs 1 instruction access + 1 data access in the same cycle.
L2 cache is unified because it is more efficient for instructions and data to utilize a single shared storage.

Short question 4 (5 points):
Consider a system with a 40-bit virtual address, with 4KB pages. The TLB is direct-mapped with 64 entries. How many bits are in the TLB index? How many bits are in the tag for each TLB entry? Draw the 40-bit virtual address, and show clearly where the index and tag come from.

Page offset has log2(4K) = 12 bits
index has log2(64) = 6 bits
tag has 40 – 12 – 6 = 22 bits

Address = tag || index || offset

Short Question 5 (5 points):
Given the following sum reduction code for a shared memory system, show a trace of how the partial sums are collected into the final sum. Show each iteration clearly, and what each processor (0-5) is executing in the iteration.

```
half = 6; // no. of processors
do {
     synch();
     // processor 0 handles odd number of partial
     // sums
     if (half % 2!= 0 && Pn == 0)
          sum[0] = sum[0] + sum[half-1];
     half = half / 2;
     if (Pn < half)
          sum[Pn] = sum[Pn] + sum[Pn+half];
} while (half > 1);
```

ANS:

Half = 6
Sync()
If (half % 2…) false
Half = 3
P0: Sum[0] += sum[3] P1: sum[1]+=sum[4] P2: sum[2]+=sum[5]
Sync()
If (half %2…) true
P0: sum[0]+=sum[2]
P0: sum[0]+=sum[1]