Full Name:_____

15-740/18-740, Fall 2012

Exam 1

October 9, 2012, 9:00am-10:20am

Instructions:

- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- The exam has a maximum score of 100 points.
- The problems are of varying difficulty. The point value of each problem is indicated. Pile up the easy points quickly and then come back to the harder problems.
- This exam is CLOSED BOOK, CLOSED NOTES. You may use a calculator, but no networked devices (e.g., PDAs, laptops, etc.).

Do not write below this line

Problem	Your Score	Possible Points
1		9
2		10
3		9
4		12
5		24
6		12
7		24
Total		100

Understanding Performance

Problem 1. (9 points):

A. Show an equation (commonly referred to as "Amdahl's Law") that computes *overall speedup* as a function of two parameters: f = the fraction of the original execution time that has been accelerated, and s = the speedup achieved by this accelerated portion of the code.

Overall Speedup =

B. What is the significance of Amdahl's Law in computer architecture? In other words, what lessons should architects learn from this equation?

C. You have developed a new graphics pipeline for a video game. This pipeline consists of three stages, where the individual stages have the following properties:

Stage	Latency	Throughput
1	4 seconds	3 images/second
2	5 seconds	1 image/second
3	2 seconds	8 images/second

How long will it take to send a sequence of 20 images through your pipeline?

Problem 2. (10 points):

This problem tests your understanding of x86-64 assembly language. Consider the following C code:

```
long int silly_sum (long int threshold, long int* array, long int length) {
  long int i;
  long int sum = 0;
  long int not_summed = 0;
  for (i = 0; i < length; i++) {
    if (array[i] > threshold)
      sum += (array[i]-threshold);
    else not_summed++;
  }
  return (sum - not_summed);
}
```

The corresponding machine code (output from `objdump -d`) is as follows:

```
 0x0:  41 b9 00 00 00 00     mov    $0x0,%r9d
 0x6:  41 b8 00 00 00 00     mov    $0x0,%r8d
 0xc:  48 85 d2              test   %rdx,%rdx
 0xf:  7e 2f                 jle    0x40
0x11:  b9 00 00 00 00        mov    $0x0,%ecx
0x16:  41 b9 00 00 00 00     mov    $0x0,%r9d
0x1c:  41 b8 00 00 00 00     mov    $0x0,%r8d
0x22:  48 8b 04 ce           mov    (%rsi,%rcx,8),%rax
0x26:  48 39 f8              cmp    %rdi,%rax
0x29:  7e 08                 jle    0x33
0x2b:  48 29 f8              sub    %rdi,%rax
0x2e:  49 01 c1              add    %rax,%r9
0x31:  eb 04                 jmp    0x37
0x33:  49 83 c0 01           add    $0x1,%r8
0x37:  48 83 c1 01           add    $0x1,%rcx
0x3b:  48 39 ca              cmp    %rcx,%rdx
0x3e:  7f e2                 jg     0x22
0x40:  4c 89 c8              mov    %r9,%rax
0x43:  4c 29 c0              sub    %r8,%rax
0x46:  c3                    retq
```

Recall that under the x86-64 register convention, the first three arguments to a procedure are passed in registers `%rdi`, `%rsi`, and `%rdx`, respectively, and the return value is passed in register `%rax`.

A. Identify the cycle: what are the addresses of the *first* and *last* instructions in the machine code that perform the actual iterating in the loop? (Ignore any test instructions outside the cycle.)

B. Give the valid C language expression that best describes the contents of the following registers. A valid C language expression is one that can be parsed by a C compiler (e.g., `i+1`, `v[0]`, etc.).

   (a) `%r8` (same as `%r8d`):

   (b) `%r9` (same as `%r9d`):

   (c) `%rax` for the instruction at address `0x26`:

   (d) `%rax` for the instruction at address `0x2e`:

Caches

Problem 3. (9 points):

A. **On-Chip Cache Hierarchies**

Why does a modern microprocessor typically have *two or three* levels of on-chip cache? (i.e. why not just one? Why not five?)

B. **Blocking/Tiling vs. Prefetching**

During our lecture on the memory hierarchy, we described two software techniques for improving cache performance: (i) *blocking* (also known as *tiling*, and not to be confused with either "cache blocks" or a synonym for stalling), as illustrated by the blocked matrix multiplication algorithm, and (ii) *prefetching*. Both of these techniques have advantages and limitations.

(a) Describe an advantage of blocking/tiling relative to prefetching.

(b) Describe an advantage of prefetching relative to blocking/tiling.

Comparing Parallel Programming Models

Problem 4. (12 points):

Recall that in class we used a simplifed Grid Solver kernel to highlight some of the differences between the (i) *Data Parallel*, (ii) *Shared Address Space*, and (iii) *Message Passing* parallel programming models.

A. **Shared Address Space vs. Message Passing:** With regards to the *Shared Address Space* and the *Message Passing* programming models, *please contrast these two programming models* for each of the following aspects of orchestrating the parallel computation:

(a) How *communication* takes place (e.g., is it implicit or explicit, etc.).

(b) How *synchronization* takes place.

(c) How *replication* is managed.

Cache Coherence

Problem 5. (24 points):

The following questions are related to cache coherence.

A. Consider a machine with 32 byte cache lines that uses a flat, memory-based directory cache coherence protocol. For each of the two machine sizes below, compute the storage overhead of the "presence" state only (i.e. you can ignore the dirty and overflow bits) as a *fraction of main memory size* (or percentage, if you prefer) for the following two directory schemes: (i) full bit vector, and (ii) the limited pointer Dir_iB scheme where $i = 3$.

(a) 16 processors

(b) 1024 processors

B. Recall that when we discussed snoopy invalidation-based coherence protocols in class for bus-based multiprocessors, one potential optimization is to allow for *cache-to-cache sharing*: i.e. allowing another cache to supply the data on a read miss to clean data. Would this form of cache-to-cache sharing make any sense in a flat, memory-based *directory* coherence protocol? Explain your answer in terms of how this idea would be incorporated into a directory-based coherence scheme, and the expected impact on performance.

C. The following questions are related to object sharing patterns and how they affect invalidation traffic for a directory-based implementation of cache coherence.

(a) Why is it that objects that are frequently read and written (e.g., a shared task queue) are not expected to cause large numbers of invalidations upon writes?

(b) Why are high-contention spin locks the worst possible scenario in terms of invalidation traffic for directory-based cache coherence? Be specific.

(c) Given that they are the same data structure, why don't *low-contention* spin locks cause a problem with respect to coherence traffic?