CSCE 513: Computer Architecture, Fall 2018
Assignment #5, Due 12/07/2018, Friday 11:55PM

Covered topics: Thread Level Parallelism and Domain-Specific Architectures
Total Points: 100 points for both undergraduate and graduate students

Submission:

1. **Only electronic submissions on dropbox are accepted.**
2. **You should submit a SINGLE PDF file that includes all your solutions.**
3. **Number your solutions in the same way and in the same order as the questions are numbered in this document and do NOT include the questions as part of your submission.**
4. **Include your full name in the PDF file.**
5. **Scanned copy of handwritten answers will NOT be graded.**

Thread Level Parallelism (80 points)
5.1 (35 points)
5.28 (30 points)
5.32 (15 points)

Domain-specific Architectures
7.1, b, c and d (20 points)

A multicore SMT multiprocessor is illustrated in Figure 5.37. Only the cache contents are shown. Each core has a single, private cache with coherence maintained using the snooping coherence protocol of Figure 5.7. Each cache is direct-mapped, with four lines, each holdin[illegible] 8 bytes (to simplify diagram). For further simplification, the whole line addresses in memory are shown in the address fields in the caches, where the tag would normally exist. The coherence states are denoted M, S, and I for Modified, Shared, and Invalid.

5.1. [10/10/10/10/10/10/10] <5.2> For each part of this exercise, the initial cache and memory state are assumed to initially have the contents shown in Figure 5.37. Each part of this exercise specifies a sequence of one or more CPU operations of the form

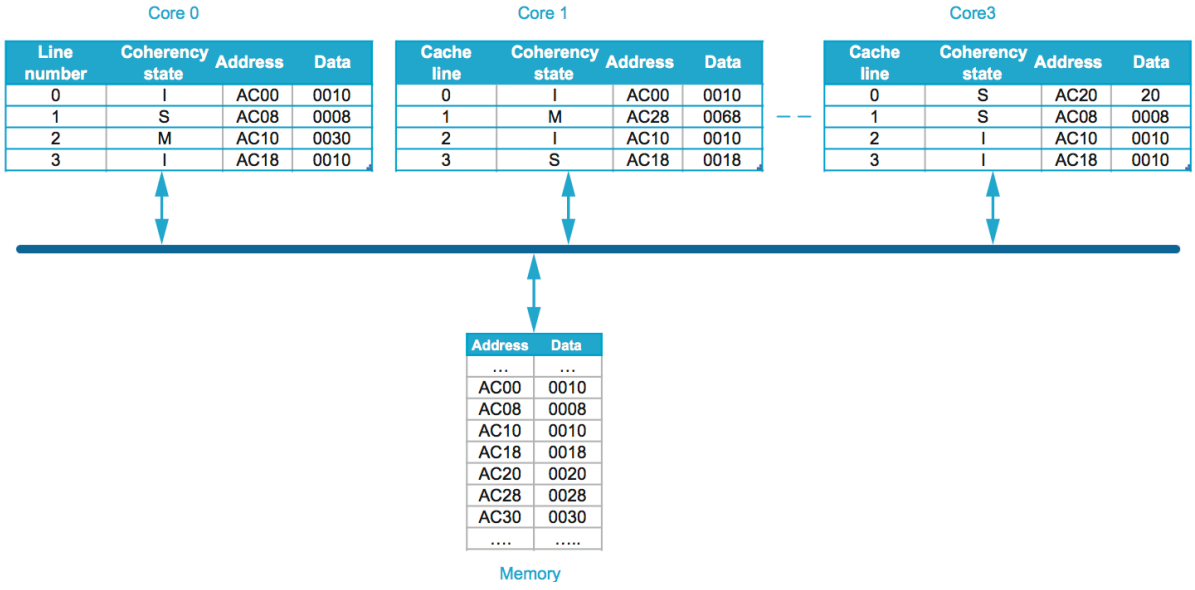

Figure 5.37 Multicore (point-to-point) multiprocessor.

`Ccore#: R, <address>` for reads

and

`Ccore#: W, <address> <-- <value written>` for writes.

For example,

`C3: R, AC10 & C0: W, AC18 <-- 0018`

Read and write operations are for 1 byte at a time. Show the resulting state (i.e., coherence state, tags, and data) of the caches and memory after the actions given below. Show only the cache lines that experience some state change; for example:

`C0.L0: (I, AC20, 0001)` indicates that line 0 in core 0 assumes an "invalid" coherence state (`I`), stores `AC20` from the memory, and has data contents `0001`. Furthermore, represent any changes to the memory state as `M: <address> <- value`.

Different parts (a) through (g) do not depend on one another: assume the actions in all parts are applied to the initial cache and memory states.

a. [10] <5.2> `C0: R, AC20`

b. [10] <5.2> `C0: W, AC20 <-- 80`

c. [10] <5.2> `C3: W, AC20 <-- 80`

d. [10] <5.2> `C1: R, AC10`

e. [10] <5.2> `C0: W, AC08 <-- 48`

f. [10] <5.2> `C0: W, AC30 <-- 78`

g. [10] <5.2> `C3: W, AC30 <-- 78`

5.28. [15] <5.3> An application is calculating the number of occurrences of a certain word in a very large number of documents. A very large number of processors divided the work, searching the different documents. They created a huge array—word_count—of 32-bit integers, every element of which is the number of times the word occurred in some document. In a second phase, the computation is moved to a small SMP server with four processors. Each processor sums up approximately ¼ of the array elements. Later, one processor calculates the total sum.

Five *Thread-Level Parallelism*

```
for (int p=0; p<=3; p++) // Each iteration of is executed on a
                            separate processor.
{
     sum [p] = 0;
     for (int i=0; i<n/4; i++) // n is size of word_count and
                                  is divisible by 4
          sum[p] = sum[p] + word_count[p+4*i];
}
total_sum = sum[0] +sum[1]+sum[2]+sum[3] //executed only
                                           on processor.
```

a. Assuming each processor has a 32-byte L1 data cache. Identify the cache line sharing (true or false) that the code exhibits.

b. Rewrite the code to reduce the number of misses to elements of the array word_count.

c. Identify a manual fix you can make to the code to rid it of any false sharing.

5.32. [10] <5.5> Implement the classical compare-and-swap instruction using the *load linked/store conditional* instruction pair.

7.1 [10/20/10/25/25] <7.3,7.4> Matrix multiplication is a key operation supported in hardware by the TPU. Before going into details of the TPU hardware, it's worth analyzing the matrix multiplication calculation itself. One common way to depict matrix multiplication is with the following triply nested loop:

```
float a[M][K], b[K][N], c[M][N];
// M, N, and K are constants.
for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
                for (int k = 0; k < K; ++k)
                        c[i][j] += a[i][k] * b[k][j];
```

b. [20] Suppose that M=3, N=4, and K=5, so that each of the dimensions are relatively prime. Write out the order of accesses to memory locations in each of the three matrices A, B, and C (you might start with two-dimensional indices, then translate those to memory addresses or offsets from the start of each matrix). For which matrices are the elements accessed sequentially? Which are not? Assume row-major (C-language) memory ordering.

c. [10] Suppose that you transpose matrix B, swapping its indices so that they are B[N][K] instead. So, now the innermost statement of the loop looks like:

```
c[i][j] += a[i][k] * b[j][k];
```

Now, for which matrices are the elements accessed sequentially?

d. [25] The innermost (k-indexed) loop of our original routine performs a dot-product operation. Suppose that you are a given a hardware unit that can perform an 8-element dot-product more efficiently than the raw C code, behaving effectively like this C function:

```
void hardware_dot(float *accumulator,
   const float *a_slice, const float *b_slice) {
        float total = 0.;
        for (int k = 0; k < 8; ++k) {
                total += a_slice[k] * b_slice[k];
        }
        *accumulator += total;
}
```

How would you rewrite the routine with the transposed B matrix from part (c) to use this function?