COMP 35112

Two hours

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Chip Multiprocessors

Date: Friday 18th May 2018

Time: 14:00 - 16:00

Please answer all Questions.

© The University of Manchester, 2018

This is a CLOSED book examination

The use of electronic calculators is permitted provided they
are not programmable and do not store text

[PTO]

Note: Where a question asks for instruction-level code, a format similar to ARM assembler is expected. However, marks will not be lost if the format is incorrect, as long as the meaning of each instruction is clear (from an accompanying explanation and/or comments).

Question 1.

a) Explain the key distinctions between parallel programming models that are based on **data-sharing** and those based on **message-passing**.

(2 marks)

b) Hardware multicore memory architectures fall into two main classes, namely **shared memory** (in which load and store instructions access a single address space that is shared by every core) or **distributed memory** (in which each core has its own private address space which no other core can access). Both classes are Turing complete (i.e. capable of being programmed to compute any computable function) so either kind of parallel programming model from part a) can be implemented using either class of memory architecture. Explain how you would approach the implementation of each kind of parallel programming model (**data-sharing** and **message-passing**) on each class of architecture (**shared memory** and **distributed memory**), highlighting anything you expect to be difficult, and comment on the performance you would expect to achieve in each case.

(5 marks)

c) In the context of thread-based data-sharing parallel programming in Java, explain why synchronisation constructs, such as barriers, locks and semaphores, are needed by a programmer. Give examples to illustrate your answer.

(3 marks)

d) Using standard load-store instructions for code implementing a binary semaphore as an example, explain why it is necessary to provide hardware-level support for synchronisation operations in a multicore processor.

(3 marks)

e) Explain how the pair of instructions ‘load linked’ and ‘store conditional’ can be used to implement a binary semaphore). Your answer should include an indication of the code required at instruction level.

(3 marks)

f) The following instruction level code is to be executed by N threads after the memory location pointed to by the value in r2 has been initialised to N. Explain what high-level function the code is implementing and, therefore, describe the behaviour of the threads up to the point when all threads reach the line labelled done.

```
loop:  ldl r1, r2       // load linked into r1 – memory address is in r2
       sub r1, r1, #1   // decrement r1
       stc r1, r2       // store conditional r1 – memory address is in r2
       cmp #1, r1       // compare r1 with 1
       bne loop         // branch if not equal to loop
spin:  ldr r1, r2       // load r1 – memory address is in r2
       cmp r1, #0       // compare r1 with 0
       bne spin         // branch if not equal to spin
done:  …
```

(4 marks)

[PTO]

Question 2.

a) Explain the role of Transactional Memory in a chip multiprocessor and describe the key steps that are required when executing a transaction.

(5 marks)

b) On a chip multiprocessor with Transactional Memory, each thread contains the transaction below which computes the histogram (a one dimensional array) of the values contained in a two dimensional array. You may assume the iteration space has been partitioned amongst the threads in a sensible way.

```
atomic {
   hist[array[i][j]]++;
}
```

Write two equivalent (pseudo)code solutions, one using coarse-grained locking and one which uses fine-grained locking, to achieve the same effect. Explain any differences in behaviour you would expect between each of the locking codes and the code using a transaction.

(5 marks)

c) Explain what the **readset** and **writeset** are used for in an implementation of Transactional Memory.

(2 marks)

d) Explain the difference between **eager versioning** and **lazy versioning** in respect of writes to shared data during execution of a transaction. Under what circumstances would either scheme be preferred?

(3 marks)

e) Explain the difference between **eager validation** and **lazy validation** in respect of detecting conflicts when trying to commit or abort a transaction. Under what circumstances would either scheme be preferred?

(3 marks)

f) Briefly explain why the use of load-linked and store conditional instructions can be considered to implement a transaction.

(2 marks)

Question 3.

a) Explain why cache coherence is a key feature in chip multiprocessor design.

(4 marks)

b) Assume a 'snooping bus' MESI cache coherence protocol in which a cache line can be in one of the four states: M (modified), E (Exclusive), S (shared), I (invalid). Explain briefly what each of these states means in terms of the collective state of the multiple caches.

(2 marks)

c) In the context of your answer to part b), explain briefly why it is additionally helpful to differentiate the following state: a modified cache line that has been copied by another core (which now holds this cache line in state S) without having been written back to memory.

(1 marks)

d) What are the main impediments to the scalability of a "snoopy bus"-based coherence protocol and how would using a directory-based protocol affect both the performance and scalability of cache coherence?

(3 marks)

e) For an MSI-like protocol, compare the details of the bus-based and directory-based protocols for a state change required by (i) a read-miss to a line, and (ii) a write-miss in the cache of a core for a cache line currently in the shared state.

(4 marks)

f) Briefly explain what is meant by *sequential consistency* and give a simple example of how locks or synchronized methods in Java can be used to ensure sequential consistency at the level of Java methods.

(3 marks)

g) Write buffers can lead to a core allowing stores to be performed out-of-order and thus break sequential consistency. Briefly explain how *release consistency* allows some out-of-order execution while maintaining sequential consistency from the perspective of programmers.

(3 marks)

END OF EXAMINATION