In the system below, the two threads run in parallel on an x86 multiprocessor and share the memory locations x and y, which initially hold 0:

```
Thread 0
MOVL x→EAX
```

```
Thread 1
MOVL y→EAX
```

Can you guess all the possible final states? How many final states are there?

---

### Quiz 2.

In the system below, the two threads run in parallel on an x86 multiprocessor and share the memory locations x and y, which initially hold 0:

```
Thread 0
MOVL x→EAX
```

```
Thread 1
MOVL y→EAX
```

Can the system reach a final state where register 0xEAX holds 1, register 0xEBX holds 0, and the memory location x holds 1?

---

### Quiz 3.

In the system below, the two threads run in parallel on a Power multiprocessor and share the memory locations x and y, which initially hold 0:

```
Thread 0
MOVL x→EAX
```

```
Thread 1
MOVL y→EAX
```

Can the system reach a final state where both registers 0x1:r1 and 1:r2 hold 1? Is this behaviour possible on an x86 multiprocessor?

---

### Quiz 4.

In the system below, the four threads run in parallel on a Power multiprocessor and share the memory locations x and y, which initially hold 0:

```
Thread 0
MOVL x→EAX
```

```
Thread 1
MOVL y→EAX
```

```
Thread 2
```

```
Thread 3
```

Can the system reach a final state where register 2;r1 holds 1 and 2;r2 holds 0 (that is, thread 2 sees the write to y but not the write to x) while register 3;r3 holds 1 and 3;r4 holds 0 (that is, thread 3 sees the write to y but not the write to x)? Is this behaviour possible on an x86 multiprocessor?

---

### Quiz 5.

The code below attempts to implement a common message passing strategy: Thread 0 updates a data structure (at memory location x, initially 0) and then sets a flag (at memory location y, initially 0), while Thread 1 checks the flag and then accesses the data structure:

```
Thread 1
```

```
```

Suppose that at the end of the execution Thread 1 has seen the update to the flag, that is, 1:r0 = 1. Are we guaranteed that 1:r1 holds 1?

If not, can you propose a way to insert memory barriers in the code so that the message passing idiom works implemented correctly? The available barrier instructions are stdsync, stdsync, and stdsync.

---

### Quiz 6.

Consider the following Java program where x and y are global variables initially holding 0:

```
Java
```

```
Consider the following Java program where x and y are global variables initially holding 0:

```
Java
```

```
Suppose that at the end of the execution Thread 1 has seen the update to the flag, that is, 1:r0 = 1. Are we guaranteed that 1:r1 holds 1?

If not, can you propose a way to insert memory barriers in the code so that the message passing idiom works implemented correctly? The available barrier instructions are stdsync, stdsync, and stdsync.

---

### Answers

1. There are 4 possible final states. In the unexpected final state, the memory locations hold 1 while the registers all hold 0 (that is, 0xEAX = 0 and 0xEBX = 0). This test highlights that x86 multiprocessors implement store buffering. See [3,4].

2. The answer is yes. However, an accurate reading of the Intel 64 Architecture Memory Ordering White Paper suggests that such final state is forbidden. This example shows that the informal prose documentation should be trusted only to a limited extent, and rigorous models of weak memory models must rely on formal mathematics and actual testing of the processors. See [3,7].

3. Yes. Power multiprocessors exhibit a memory model much more relaxed than x86 multiprocessors: unless precise conditions are met, loads and stores can be rearranged arbitrarily by each thread. See [4,6].

4. Yes. In a Power multiprocessor stores are not propagated atomically to other processors (while x86 guarantees store-atomicity). Most accounts of weak memory models do not take into account this crucial aspect. See [4,6].

5. The behaviours illustrated by Quiz 3 and 4 imply that the given code does not implement correct message passing. It is possible to recover a correct behavior by inserting sync or lexsync barrier instructions between the two stores and the two loads (but lexsync barriers will not suffice). More efficient implementations can rely on an address dependency between the two reads and an lexsync between the two write. See [4,6]. A strategy to insert barriers to recover sequential consistency is given in [4].

6. The program should not print 1 (even according to the JSR-133 memory model), but it does when compiled with HotSpot or GcJ. We investigate concurrent high-level programming languages and compilation in [8,9].

---

### Summary of scientific results

- A formal model of the x86 relaxed memory model. Joint publications [5] and [8].
- A formal model of the Power relaxed memory model. Joint publications [3, 6] and [8].
- Tools to explore the memory model of modern processors. Joint publications [4] and [6].
- The diy tool suite is available as free software from http://diy.inria.fr.
- CompCertTSO, a verified compiler from Chapel50 to x86. Joint publication [2].
- Compiler available from http://www.cl.cam.ac.uk/~pes20/CompCertTSO.

---

### Students jointly supervised


### Joint Publications


### People

ARIA: Jade Alglave (now at Oxford U.), Thomas Braibant (now at IRINA Grenoble), Luc Maranget, Kayvan Memarian (now at U. of Cambridge), Pankuj Pawan, Francesco Zappa Nardelli (University of Cambridge). Anthony Fox, Mark Baty, Peter Boosan, Magnus Myreen, Suresh Jagannathan (visiting from Purdue U.) Scott Owens, Tom Ridge (now at U. of Leicester) Susmit Sarkar, Peter Sewell, Jaroslav Sewick (now at Microsoft), Viktor Vafeiadis (now at MPS-SWS).