MultiProcessors
Key Points

• What is a CMP?
• Why have we started building them?
• Why are they hard to use?
• What is deadlock?
• What is cache coherence?
• What is cache consistency?
Last Time

- Wide-issue
  - Some ILP, and some complexity
- VLIW
  - Some ILP, but less complexity
- OOO Superscalars
  - More ILP, lots and lots of complexity
  - Instruction windows
  - Register renaming
What’s good about OOO

• It’s responsible for a large fraction of CPU performance -- i.e., it works
• It delivers single-thread performance
  • No changes are required from the program
  • Buy a new, faster, bigger OOO machine, and your program will run faster.
• This means that OOO machines are easy to use and easy to program.
The Problems with OOO

- Limited per-thread ILP
  - Bigger windows don’t buy you that much
- Complexity
  - Building and verifying large OOO machines is hard (but doable)
- Area efficiency (per-xtr efficiency)
  - Doubling the area devoted to OOO mechanisms doesn’t come close to doubling performance
- Power efficiency
  - Large OOO don’t provide good power efficiency returns either.
- For all these reasons, OOO growth has almost stopped.
Frequency and Power

- $P = CfV^2$
- $f =$ processor frequency
- $V =$ supply voltage
- $C =$ circuit capacitance (basically xtr count)
- To increase $f$ you need to increase $V$ as well
  - Approximately: $P = Cf^3$

- This means that even for in-order processors, frequency scaling is not power efficient
  - doubling the frequency doubles performance
  - increased power by 8x
- It is, however, very area-efficient/xtr-efficient
Multi-processors

- An alternative approach to increased performance: Build more processors
- N processors will do N times as much work per time
- Area efficiency:
  - Pretty good -- twice the area -> twice the performance (Maybe. Sometimes. More on this in moment)
- Power efficiency:
  - $P = Cf^3$
  - Two processors means doubling C, so 2x the power.
What should we build?

• Building bigger OOO processors doesn’t pay
• Power budgets are fixed.
• Moore’s law keeps delivering more xtrs
• Consequences
  • Power efficiency is more important than area efficiency
  • Multi-processors are now more attractive.
Multiprocessors

- Specifically, shared-memory multiprocessors have been around for a long time.
- Originally, put several processors in a box and provide them access to a single, shared memory.
- Expensive and mildly exotic.
  - Big servers
  - Sophisticated users/data-center applications
Chip Multiprocessors (CMPS)

- Multiple processors on one die
- An easy way to spend xtrs
- Now common place
  - Laptops/desktops/game consoles/etc.
  - Less sophisticated users, all kinds of applications.
Why didn’t we get here sooner

• Doubling performance with frequency increases power by 8x
• Doubling performance with multiple cores increases power by 2x
• No brainer?!? -- Only a good deal if
  • Power matters -- for a long time it didn’t
  • and you actually get twice the performance
The Trouble With CMPs

- Amdahl’s law
  - $Stot = \frac{1}{(x/S + (1-x))}$
- In order to double performance with a 2-way CMP
  - $S = 2$
  - $x = 1$
  - Usually, neither is achievable
Threads are Hard to Find

- To exploit CMP parallelism you need multiple processes or multiple “threads”
- Processes
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Common in servers
  - Much less common in desktop/laptops
- Threads
  - Independent portions of your program that can run in parallel
  - Most programs are not multi-threaded.
- We will refer to these collectively as “threads”
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)
Parallel Programming is Hard

• Difficulties
  • Correctly identifying independent portions of complex programs
  • Sharing data between threads safely.
  • Using locks correctly
  • Avoiding deadlock

• There do not appear to be good solutions
  • We have been working on this for 30 years (remember, multi-processors have been around for a long time.)
  • It remains stubbornly hard.
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1;    // Line 1
    shared_value = t;            // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the `shared_value` to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.

Instructions in the two threads can be interleaved in any way.
Critical Sections and Locks

- By adding a lock, we can ensure that only one thread executes the critical section at a time.

```c
int shared_value = 0;
lock shared_value_lock;
void IncrementSharedVariable()
{
    acquire(shared_value_lock);
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
    release(shared_value_lock);
}
```

- In this case we say shared_value_lock “protects” shared_value.
Locks are Hard

- The relationship between locks and the data they protect is not explicit in the source code and not enforced by the compiler
- In large systems, the programmers typically cannot tell you what the mapping is
- As a result, there are many bugs.
Locking Bug Example

void Swap(int * a, lock * a_lock,
          int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}

Thread 1

Swap(foo, foo_lock,
     bar, bar_lock);

...  

Thread 2

Swap(bar, bar_lock,
     foo, foo_lock);

...
Locking Bug Example

```c
void Swap(int * a, lock * a_lock,
           int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}
```

Thread 1

```c
Swap(foo, foo_lock,
     bar, bar_lock);
```

Thread 2

```c
Swap(bar, bar_lock,
     foo, foo_lock);
```

Thread 1 locks foo_lock, thread 2 locks bar_lock, both wait indefinitely for the other lock.
Finding, preventing, and fixing this kind of bug are all hard
The Future of Threads

• Optimists believe that we will solve the parallel program problem this time!
  • New languages
  • New libraries
  • New paradigms
  • Revamped undergraduate programming courses

• Pessimists believe that we won’t
  • There is probably not a good, general solution
  • We will make piecemeal progress
  • Most programs will stop getting faster
  • CMPs just make your spyware run faster.

• Intel and Microsoft believe typical users can utilize up to about 8 cores effectively.
  • Your laptop will be there in 2-3 years.
Architectural Support for Multiprocessors

- Allowing multiple processors in the same system has a large impact on the memory system.
- How should processors see changes to memory that other processors make?
- How do we implement locks?
Shared Memory

- Multiple processors connected to a single, shared pool of DRAM
- If you don’t care about performance, this is relatively easy... but what about caches?
Uni-processor Caches

• Caches mean multiple copies of the same value
• In uniprocessors this is not a big problem
  • From the (single) processor’s perspective, the “freshest” version is always visible.
  • There is no way for the processor to circumvent the cache to see DRAM’s copy.
Caches, Caches, Everywhere

- With multiple caches, there can be many copies.
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?

Diagram:
- Store 0x1000
- Bus/arbiter
- Local caches
- Main Memory 0x1000: B
Caches, Caches, Everywhere

- With multiple caches, there can be many copies.
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies.
- No one processor can see them all.
- Which one has the “right” value?
Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?
Keeping Caches Synchronized

- We must make sure that all copies of a value in the system are up to date
  - We can update them
  - Or we can “invalidate” (i.e., destroy) them
- There should always be exactly one current value for an address
  - All processors should agree on what it is.
- We will enforce this by enforcing a total order on all load and store operations to an address and making sure that all processors observe the same ordering.
- This is called “Cache Coherence”
The Basics of Cache Coherence

• Every cache line (in each cache) is in one of 3 states
  • Shared -- There are multiple copies but they are all the same. Only reading is allowed
  • Owned -- This is the only cached copy of this data. Reading and write are allowed
  • Invalid -- This cache line does not contain valid data.
• There can be multiple sharers, but only one owner.
• There is one copy of the state machine for each line in each coherent cache.
Caches, Caches, Everywhere

Local caches

Bus/arbiter

Main Memory
Caches, Caches, Everywhere

Local caches

Exclusive

Bus/arbiter

Main Memory

Main Memory

Main Memory

Main Memory
Caches, Caches, Everywhere

Local caches

Exclusive

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive

Main Memory

0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere

Local caches

Exclusive
0x1000: A

Bus/arbiter

Main Memory

0x1000: Z

Read 0x1000
Caches, Caches, Everywhere

Local caches

Exclusive
0x1000: A

Exclusive

Exclusive

Main Memory

0x1000: Z

Bus/arbiter

Read 0x1000
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive 0x1000: A

Exclusive 0x1000: A

Main Memory

0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive
0x1000: A

Exclusive
0x1000: A

Exclusive
0x1000: A

Main Memory

0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere

Local caches

Exclusive 0x1000:A

Exclusive 0x1000:A

Exclusive 0x1000:A

Exclusive

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Read 0x1000

Local caches

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Local caches

Exclusive 0x1000: A

Read 0x1000

Exclusive 0x1000: A

Read 0x1000

Exclusive 0x1000: A

Read 0x1000

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Local caches

Read 0x1000

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Local caches

Bus/arbiter

Main Memory
Caches, Caches, Everywhere

Local caches

Exclusive

Bus/arbiter

Main Memory
Caches, Caches, Everywhere

Local
caches

Exclusive

Bus/arbiter

Main
Memory

0x1000: Z
Caches, Caches, Everywhere

- Store 0x1000
- Local caches
  - Exclusive
- Bus/arbiter
- Main Memory
  - 0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive
0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

0x1000: Z

Main Memory

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive
0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Bus/arboriter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Main Memory

0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive
0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Bus/arbiter

Main Memory

0x1000: Z
Caches, Caches, Everywhere

Local caches

Bus/arbiter

Main Memory

Store 0x1000

Exclusive 0x1000: A

0x1000: Z

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Exclusive 0x1000: A

Main Memory 0x1000: Z

Bus/arbiter
Caches, Caches, Everywhere
Caches, Caches, Everywhere

Local caches

Shared

Bus/arbiter

Main Memory
Caches, Caches, Everywhere
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared
0x1000: A

Main Memory

Bus/arbiter

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000: A

Shared

Shared

Shared

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000: A

Shared 0x1000: A

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Main Memory

0x1000: A

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000:A

Main Memory

0x1000: A

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Local caches

Shared 0x1000:A

Shared 0x1000:A

Main Memory

0x1000: A

Bus/arbiter
Caches, Caches, Everywhere
Caches, Caches, Everywhere
Caches, Caches, Everywhere

Local caches

invalid

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid
0x1000: A

Main Memory

0x1000: A

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid
0x1000: A

invalid

invalid

invalid

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid 0x1000:A

invalid 0x1000:A

Main Memory

0x1000: A

Bus/arbiter
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid 0x1000:A
invalid 0x1000:A
Owned

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid 0x1000:A
invalid 0x1000:A
Owned 0x1000:C

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid 0x1000:A
invalid 0x1000:A
Owned 0x1000:C

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Local caches

invalid 0x1000: A
invalid 0x1000: A
Owned 0x1000: C

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

Store 0x1000
Read 0x1000

Local caches
invalid
0x1000:A
invalid
0x1000:A
Owned
0x1000:C

Bus/arbiter

Main Memory
0x1000: A
Caches, Caches, Everywhere

Store 0x1000

Read 0x1000

Store 0x1000

Local caches

invalid 0x1000: A
invalid 0x1000: A
Owned 0x1000: C

Bus/arbiter

Main Memory

0x1000: A
Caches, Caches, Everywhere

- Store 0x1000
- Read 0x1000
- Store 0x1000

Local caches:
- invalid 0x1000:A
- invalid 0x1000:A
- Owned 0x1000:C
- 

Bus/arbiter

Main Memory:
- 0x1000: A
- 

Coherence in Action

```
a = 0

Thread 1
while(1) {
    a++;
}

Thread 2
while(1) {
    print(a);
}
```

Sample outputs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

possible?
Coherence in Action

```c
while(1) {
a++;
}
```

Thread 1

```
a = 0
while(1) {
    a++;
    printf(a);
}
```

Thread 2

Sample outputs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

possible? yes
Coherence in Action

Sample outputs

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

possible? yes yes
Coherence in Action

```
a = 0

Thread 1
while(1) {
    a++;
}

Thread 2
while(1) {
    print(a);
    }
```

Sample outputs

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>4</td>
</tr>
</tbody>
</table>

possible?   yes   yes   no
Live demo.
Coherence In The Real World

- Real coherence have more states
  - e.g. “Exclusive” -- I have the only copy, but it’s not modified
- Often don’t bother updating DRAM, just forward data from the current owner.
- If you want to learn more, take 240b
Cache Consistency

• If two operations occur in an order in one thread, we would like other threads to see the changes occur in the same order.

• Example:

Thread 0
A = 10;
A_is_valid = true;

while(!A_is_valid);

Thread 1
B = A;

• We want B to end up with the value 10

• Coherence does not give us this assurance, since the state machine only applies to a single cache line

• This is called “cache consistency” or “the consistency model”
Simple Consistency

• The simplest consistency model is called “sequential consistency”
• In which all stores are immediately visible everywhere.

Thread 0
A = 10;
A_is_valid = true;

Thread 1
while(!A_is_valid);
B = A;

• If thread 1 sees the write to A_is_valid, it will also see the write to A.
What about this?

```c
a = b = 0

Thread 1
while(1) {
    a++;
    b++;
}

Thread 2
while(1) {
    print(a, b);
}
```

Sample outputs

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

possible under sequential consistency?
What about this?

\[
a = b = 0
\]

Thread 1
while(1) {
    a++;
    b++;
}

Thread 2
while(1) {
    print(a, b);
}

Possible under sequential consistency? yes

Sample outputs
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 8
1 1
2 2
3 1000
4 1000
5 1000
6 1000
7 1000
8 1000
What about this?

```c
while(1) {
    a++;
    b++;
}
```

```
while(1) {
    print(a, b);
}
```

```
a = b = 0
```

**Thread 1**  
**Thread 2**

<table>
<thead>
<tr>
<th>Sample outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>2 2</td>
</tr>
<tr>
<td>3 3</td>
</tr>
<tr>
<td>4 4</td>
</tr>
<tr>
<td>5 5</td>
</tr>
<tr>
<td>6 6</td>
</tr>
<tr>
<td>7 7</td>
</tr>
<tr>
<td>8 8</td>
</tr>
</tbody>
</table>

Is it possible under sequential consistency?  
Yes  
No
Live demo.
Consistency in the Real World

• Consistency is probably the most subtle aspect of computer architecture
• No one implements sequential consistency because it is too slow
  • Make all accesses visible everywhere, right away takes a long time
• Real machines (like mine) use “relaxed” models.
  • All manner of non-intuitive things can happen
  • Special instructions to enforce sequential consistency when it’s needed
• Threading libraries (like pthreads) provide locking routines that use those special instructions to make locks work properly.
• For more, take 240b