Coherence and Consistency
The Meaning of Programs

- An ISA is a programming language
- To be useful, programs written in it must have meaning or “semantics”
  - *Any* sequence of instructions must have a meaning.
  - The semantics of arithmetic operations are pretty simple:
- What about memory?
What is Memory?

• It is an array of bytes
  • Each byte is at a location identified by a number (i.e., it’s address)
  • Bytes with consecutive addresses are next to each other
  • The difference between two addresses is the number of bytes between the two addresses
Memory in Programming Languages

• C and C++
  • Pointers are addresses
  • Arrays are just pointers
  • You can take the address of (almost) any variable
  • You can do math on pointers

• Java
  • No pointers! References instead.
  • Math on references is meaningless
  • They “name” objects.
  • They do not “address” bytes.
  • Arrays are separate construct.

• Python?
• Perl?
ISA Semantics and Order

- The semantics of RISC ISAs demand the sequential, one-at-a-time execution of instructions.
- The execution of a program is a totally ordered sequence of “dynamic” instructions.
- “Next,” “Previous,” “before,” “after,” etc. all have precise meanings.
- This is called “Program order.”
- It must appear that the instructions executed in that order.

```
or  $s0, $0, 0
addi $s0, $s0, 1
bge $s0, $a0, done
lw  $t1, 0($s3)
addi $s3, $s3, 4
add  $s1, $s1, $t1
j  check
addi $s0, $s0, 1
bge $s0, $a0, done
lw  $t1, 0($s3)
addi $s3, $s3, 4
add  $s1, $s1, $t1
j  check
addi $s0, $s0, 1
bge $s0, $a0, done
```

```
or  $s0, $0, 0
check:
addi $s0, $s0, 1
bge $s0, $a0, done
lw  $t1, 0($s3)
addi $s3, $s3, 4
add  $s1, $s1, $t1
j  check
done:
```
Vocabulary: Ordering

• An ordering is a set of ordered pairs over some set of symbols (with no cycles)
  • Ex.: a -> b, c -> d, d -> f is an ordering or some English letters.
• An ordering is “total” if there is only one linear arrangement of the symbols that is consistent with the ordered pairs
  • Ex.: a -> b, b -> c, c -> d, is a total ordering over a through e.
• A partial ordering is an ordering that is not total.
  • Ex: a -> b, a -> c, c -> d, b -> d is a partial ordering
  • c and b are unordered.
• Two orderings are ‘consistent’ if they don’t disagree
  • Ex: a -> b, b -> c, c -> d is consistent with b -> c, c -> d. But inconsistent with c -> b
ISA Semantics and Memory (for 1 CPU)

• Formal definition of a load:
  • A load from address $A$ returns the value stored to $A$ by the previous store to address to $A$
• This is the only definition in common use.
• But others are possible
  • Lazy memory: The load will return the value stored by some previous store
  • Monotonic memory: The load, $L_1$, will return the value stored by some previous store, $S_1$. If another load $L_2$ comes after $L_1$, the value it returns will be the valued stored by a Store, and $S_2$, will either be $S_1$ or come after $S_1$.
  • There’s a surprising number of potentially usable options.
Appearance is Everything (1 CPU)

- In a uniprocessor, the processor is free to execute the stores in any order
- They are all to different addresses
- The effect is indistinguishable from sequential execution.
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?
Memory for Multiple Processors

Thread 1

ori $s0, $0, 0
ori $s3, $0, 0
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[0] = 1

addi $s3, $s3, 4
add $s1, $s1, $t1
j check
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[4] = 2

addi $s3, $s3, 4
add $s1, $s1, $t1
j check
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[8] = 3

Thread 2

ori $s0, $0, 1000
ori $s3, $0, 0
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[0] = 1001

addi $s3, $s3, 4
add $s1, $s1, $t1
j check
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[4] = 1002

addi $s3, $s3, 4
add $s1, $s1, $t1
j check
addi $s0, $s0, 1
bge $s0, $a0, done
sw $s0, 0($s3) ; Mem[8] = 1003

• Now what?
Memory for Multiple Processors

• Multiple, independent sequences of instructions
  • “Next,” “Previous,” “before,” “after,” etc. no longer have obvious means for instructions on different CPUs
  • They still work fine for individual CPUs
• There are many different, possible “interleavings” of instructions across CPUs
  • Different processors may see different orders
• Non-determinism is rampant
  • “Heisenbugs”
Memory for Multiple Processors

Thread 1

\[
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[0] = 1 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[4] = 2 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[8] = 3 \\
\]

Thread 2

\[
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[0] = 1001 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[4] = 1002 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[8] = 1003 \\
\]

OR

\[
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[0] = 1 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[4] = 2 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[8] = 3 \\
\]

OR

\[
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[0] = 1001 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[4] = 1002 \\
\text{sw } s0, 0(s3) \; ; \; \text{Mem}[8] = 1003 \\
\]
ISA Semantics and Memory (for N CPUs)

• Our old definition:
  • A load from address $A$ returns the value stored to $A$ by the previous store to address to $A$
  • If there is no previous store to $A$, the value is undefined.

• A multi-processor alternative
  • For a particular execution, there is a total ordering on all memory accesses to an address $A$.
  • The same total ordering is seen by all processors.
  • The total ordering on $A$ is consistent with the program orders for all the processors.
  • A load from address $A$ returns the value stored to $A$ by the previous (in that total order) store to address to $A$

• This is “Memory coherence”
Memory Coherence

• Coherence only defines the behavior of accesses to *the same* address

• What does it tell us about this program? The final value of M[8] is either 3 or 1003, and all processors will agree on it.

  "Proof": Either mem[8] = 3 is before mem[8] = 1003 or vice versa. Exactly one of these occurs in the single, global ordering for each execution.

Thread 1

```assembly
sw $s0, 0($s3) ; Mem[0] = 1
sw $s0, 0($s3) ; Mem[4] = 2
sw $s0, 0($s3) ; Mem[8] = 3
```

Thread 2

```assembly
sw $s0, 0($s3) ; Mem[0] = 1001
sw $s0, 0($s3) ; Mem[4] = 1002
sw $s0, 0($s3) ; Mem[8] = 1003
```
A Simple Locking Scheme

• Send a value in A from thread 0 to thread 1
• What to prove:
  • If 5 executes, B will end up equal to 10

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: A = 10;</td>
<td>while(1)</td>
</tr>
<tr>
<td>2: A_is_valid = true;</td>
<td>3: if (A_is_valid)</td>
</tr>
<tr>
<td></td>
<td>4: break;</td>
</tr>
<tr>
<td></td>
<td>5: B = A;</td>
</tr>
</tbody>
</table>

• What we need (-> represents a coherence ordering):
  An ordering such that 1->…->5
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: (A = 10;)</td>
<td>while(1)</td>
</tr>
<tr>
<td>2: (A_is_valid = true;)</td>
<td>3: if ((A_is_valid))</td>
</tr>
<tr>
<td></td>
<td>4: break;</td>
</tr>
<tr>
<td></td>
<td>5: (B = A;)</td>
</tr>
</tbody>
</table>

• Prove: If 4 executes, B will end up equal to 10, so we need 1->…->5

• What **globally visible** orderings do we have available
  • Coherence order on A: 1->5
  • Coherence order on B: empty
  • Coherence order on \(A\_is\_valid\): 2->3 or 3->2
  • “causal order”: 2->4

• Coherence is not enough!
  • Communication requires coordinated updates to multiple addresses.
Memory Consistency

• Consistency provides orderings among accesses to multiple addresses
• There are many consistency models
• We will examine two
  • Sequential Consistency
  • Relaxed consistency
Sequential Consistency

• Sequential consistency is similar to coherence, but applies across all addresses
  • For a particular execution, there is a total ordering on all memory accesses to an address \( A \).
  • The same total ordering is seen by all processors.
  • The total ordering on \( A \) is consistent with the program orders for all the processors.
  • A load from address, \( A \), returns the value stored to \( A \) by the previous (in that total order) store to address to \( A \).

• This amounts to interleaving the program orders for each of the CPUs.
• This is expensive!
• But useful!
Thread 0

1: A = 10;
2: A_is_valid = true;

Thread 1

while(1)
3: if (A_is_valid)
4: break;
5: B = A;

• Prove: If 4 executes, B will end up equal to 10, so we need 1->…->5

• What **globally visible** orderings do we have available
  • Seq. Consistency ordering: 1->2 and 3->4->5
  • “causal order”: 2->4

• Proof is now easy: 1->2, 2->4, 4->5
Sequential Consistency

- **Advantages**
  - Simple
  - Intuitive. SC is what you think should happen.

- **Disadvantages**
  - Expensive to implement, since it requires global coordination to determine the global ordering
  - Prevents reordering within a single CPU.
    - If performs the stores out of order, they will be seen out of order.
  - No one implements sequential consistency.
    - Amdahl’s law says it is a bad idea
    - What fraction of memory operations implement inter-CPU communication?
Relaxed Models

• SC provided too much ordering
• Plain coherence provides not enough.
• Relaxed models
  • Provide basic coherence by default
  • Provide an instruction ("fence") to enforce orderings exactly where they are needed
• There are many relaxed models.
A Simple Relaxed Model

• Coherence
  • For a particular execution, there is a total ordering on all memory accesses to an address A.
  • The same total ordering is seen by all processors.
  • The total ordering on A is consistent with the program orders for all the processors.
  • A load from address A returns the value stored to A by the previous (in that total order) store to address to A.

• In addition, there is global ordering.
  • It is consistent with program order
  • It places a total order on all fence instructions
  • If a load, L, occurs before a fence, F, in the program order of a processor, than L→F in the global order.
Thread 0

1: A = 10;
2: A_is_valid = true;

Fence A

Thread 1

while(1)
3: if (A_is_valid)
4: break;
5: B = A;

Fence B

• Prove: If 4 executes, B will end up equal to 10, so we need 1->...->5
• We need some fence instructions, otherwise, we just have coherence.
• What **globally visible** orderings do we have available
  • Coherence orders on A, B, and A_is_valid
  • Fence order: A->B, 1->A, A->2, B->5
  • “causal order”: 2->4
• Proof:
  1->A, A->2, 2->4, 4->B, B->5
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?
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?

Diagram:
- Store 0x1000
- Read 0x1000
- Store 0x1000
- Local caches:
  - 0x1000: A
  - 0x1000: ??
  - 0x1000: C
- Bus/arbiter
- Main Memory:
  - 0x1000: B
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.
Simple Cache Coherence

- There is one copy of the state machine for each line in each coherent cache.
Caches, Caches, Everywhere

Local caches

Exclusive 0x1000: A

Store 0x1000

Bus/arbiter

0x1000: Z

Main Memory
Caches, Caches, Everywhere

Local caches

Store 0x1000

Read 0x1000

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? yes yes yes no
False Sharing