Parallel Memory Systems (cont.)

March 6th, 2023

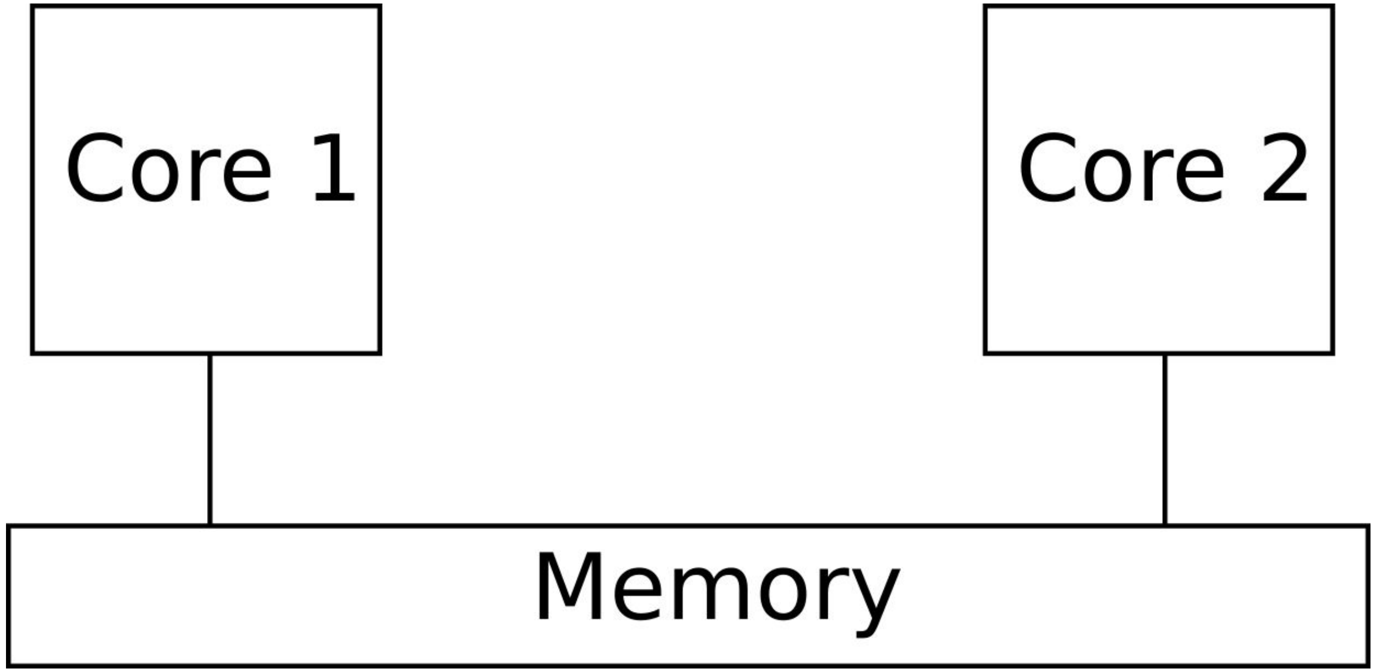
Outline

- Synchronization problem in shared memory.

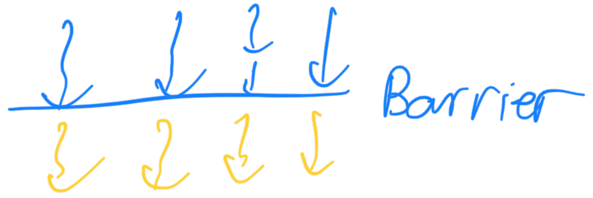
Communication in shared memory

- Software synchronization primitives
  - Barrier: a point where **all** threads must arrive to before proceeding.
  - Semaphore:
    - Waiting room for sharing multiple of the same resource.
    - Allowing the next thread to be notified when one of the threads using the resource released the resource.
    - Mechanism: the thread wanting to access the resource send the signal saying that it wants to use the resource, and then waiting for the resource via a waiting mechanism. The thread will be notified when the semaphore allows it to access the resource.
  - Mutex:
    - Holding a binary status of a lock: acquired or not acquired.
    - Used to guarantee that, at most one thread can a lock at a time.
- Note: waiting mechanisms: spin-waiting, sleeping, etc.

Setup



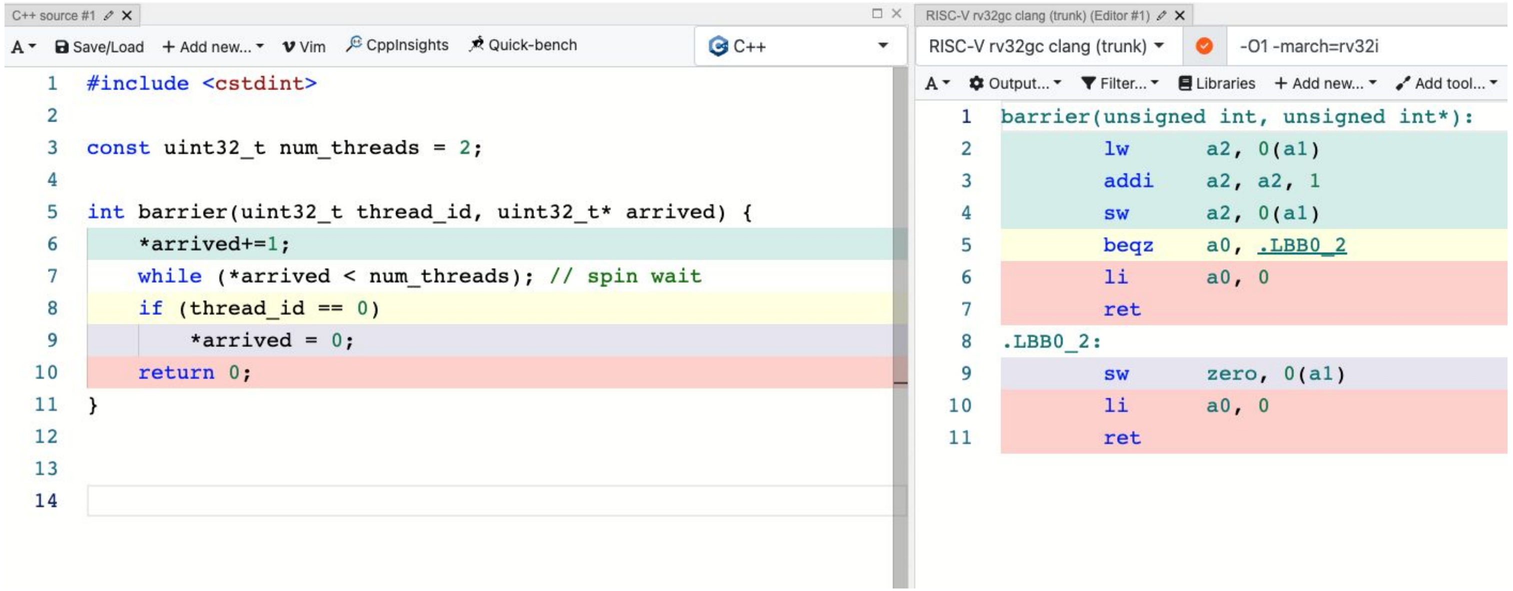
Example: Barrier



- Main idea: all threads must reach the same execution point before proceeding.
- Software multithreading Reminders
    - Software threads spawned from one thread sharing many things, including instructions, heap, address space.
    - Sharing instructions: All threads execute the same code! Thus, they can reach the same execution point.
    - Sharing heap: All threads can access any thing in the heap!
    - Software threads do not share architectural states (PC, registers, etc.) This means different threads can be at different PCs in the same cycle.

Barrier: Implementation 1

- Compilers are smart: they just optimize our intention to spin-wait away!
  - Dead-code elimination: the spin wait does not change the state of the program (in this context)



```cpp
#include <cstdint>

const uint32_t num_threads = 2;

int barrier(uint32_t thread_id, uint32_t* arrived) {
    *arrived+=1;
    while (*arrived < num_threads); // spin wait
    if (thread_id == 0)
        *arrived = 0;
    return 0;
}
```

RISC-V rv32gc clang (trunk) -O1 -march=rv32i

```asm
barrier(unsigned int, unsigned int*):
        lw      a2, 0(a1)
        addi    a2, a2, 1
        sw      a2, 0(a1)
        beqz    a0, .LBB0_2
        li      a0, 0
        ret
.LBB0_2:
        sw      zero, 0(a1)
        li      a0, 0
        ret
```

https://godbolt.org/z/re8a7YGYx

Barrier: Implementation 1 Problems

→ compiler doesn't know that "arrived" can be updated not by this thread.

Barrier: Implementation 2 - adding the “volatile” keyword

- Adding the “volatile” keyword to a variable forces compilers to generate code such that,
    - Per operation, the variable will be loaded from memory before the operation.
    - Without the keyword, if the value of variable was loaded to a register before, the value in that register will be reused.

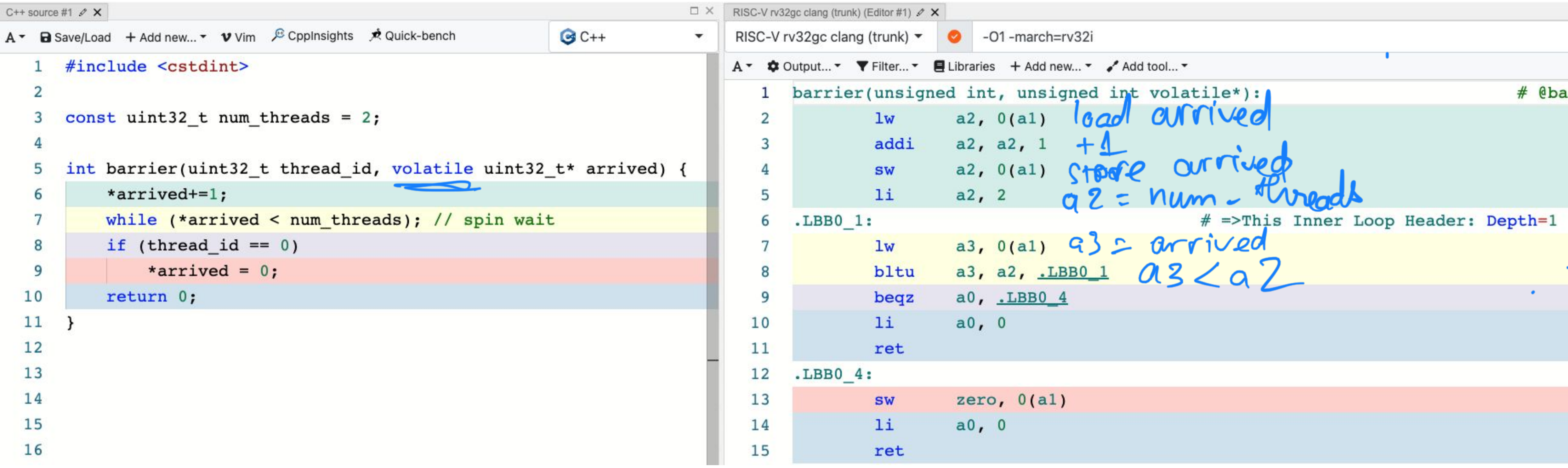
volatile

load then operation

non-volatile

can reuse val in register

Barrier: Implementation 2 - adding the “volatile” keyword



C++ source #1
A Save/Load + Add new... Vim CppInsights Quick-bench
C++
RISC-V rv32gc clang (trunk) (Editor #1)
RISC-V rv32gc clang (trunk)
A Output... Filter... Libraries + Add new... Add tool...

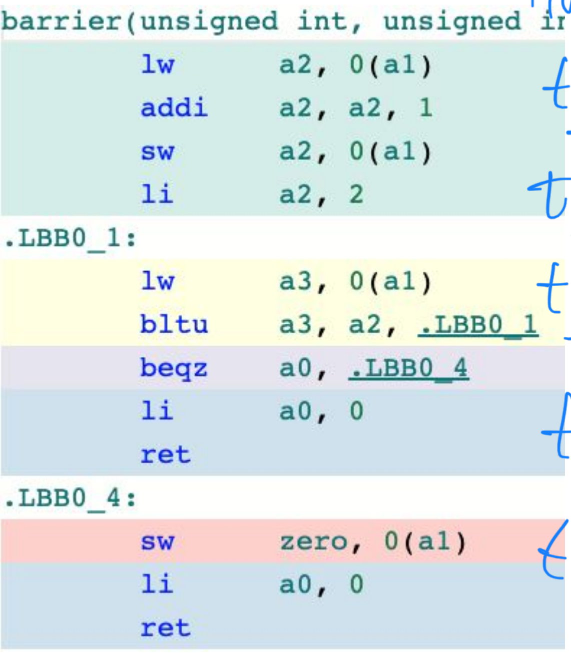
-O1 -march=rv32i

```cpp
#include <cstdint>

const uint32_t num_threads = 2;

int barrier(uint32_t thread_id, volatile uint32_t* arrived) {
    *arrived+=1;
    while (*arrived < num_threads); // spin wait
    if (thread_id == 0)
        *arrived = 0;
    return 0;
}
```

```asm
barrier(unsigned int, unsigned int volatile*):                # @ba
        lw      a2, 0(a1)       load arrived
        addi    a2, a2, 1       +1
        sw      a2, 0(a1)       store arrived
        li      a2, 2           a2 = num_threads
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
        lw      a3, 0(a1)       a3 = arrived
        bltu    a3, a2, .LBB0_1 a3<a2
        beqz    a0, .LBB0_4
        li      a0, 0
        ret
.LBB0_4:
        sw      zero, 0(a1)
        li      a0, 0
        ret
```

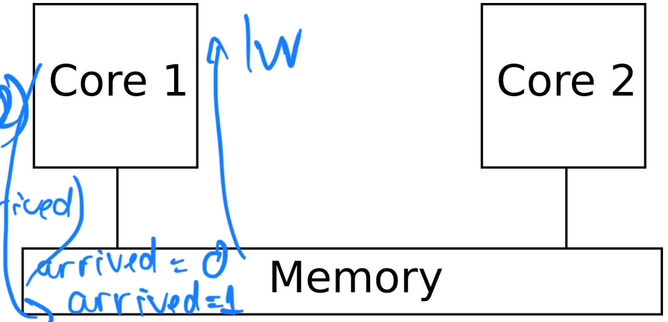
https://godbolt.org/z/MoGdY7Ezo

Barrier: Implementation 2 Problems



```
barrier(unsigned int, unsigned in
        lw      a2, 0(a1)
        addi    a2, a2, 1
        sw      a2, 0(a1)
        li      a2, 2
.LBB0_1:
        lw      a3, 0(a1)
        bltu    a3, a2, .LBB0_1
        beqz    a0, .LBB0_4
        li      a0, 0
        ret
.LBB0_4:
        sw      zero, 0(a1)
        li      a0, 0
        ret
```

Time	Thread 1 (Core1)	Thread2 (Core2)
t_1	lw (arrived =0)	
t_2	addi (arrived =1)	lw (arrived =0)
t_3	sw (arrived =1)	addi (arrived =1)
t_4	spin wait	sw (arrived =1)
t_5	spin wait	spin wait

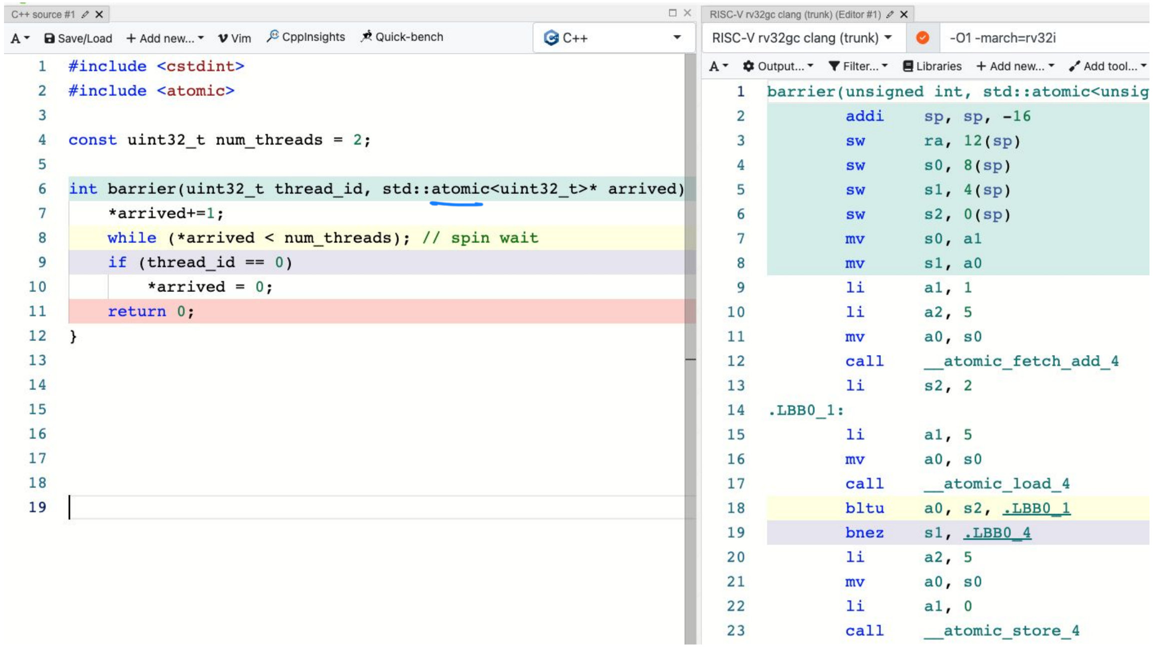


Barrier: Implementation 2 Problems

- There is no guarantee when an instruction of one thread will be executed relative to any other threads!

Barrier: Implementation 3 - Atomic access to a variable

- `*arrived+=1` consists of loading the value, incrementing the value, and storing the value.
- atomic primitives guarantee that only one thread has access to `arrived` when the above three operations being executed.
- Main take away: programmers must explicitly specify which variables are racy.



```cpp
#include <cstdint>
#include <atomic>

const uint32_t num_threads = 2;

int barrier(uint32_t thread_id, std::atomic<uint32_t>* arrived)
    *arrived+=1;
    while (*arrived < num_threads); // spin wait
    if (thread_id == 0)
        *arrived = 0;
    return 0;
}

```

RISC-V rv32gc clang (trunk) -O1 -march=rv32i

```
barrier(unsigned int, std::atomic<unsig
        addi    sp, sp, -16
        sw      ra, 12(sp)
        sw      s0, 8(sp)
        sw      s1, 4(sp)
        sw      s2, 0(sp)
        mv      s0, a1
        mv      s1, a0
        li      a1, 1
        li      a2, 5
        mv      a0, s0
        call    __atomic_fetch_add_4
        li      s2, 2
.LBB0_1:
        li      a1, 5
        mv      a0, s0
        call    __atomic_load_4
        bltu    a0, s2, .LBB0_1
        bnez    s1, .LBB0_4
        li      a2, 5
        mv      a0, s0
        li      a1, 0
        call    __atomic_store_4
```

atomic access

load

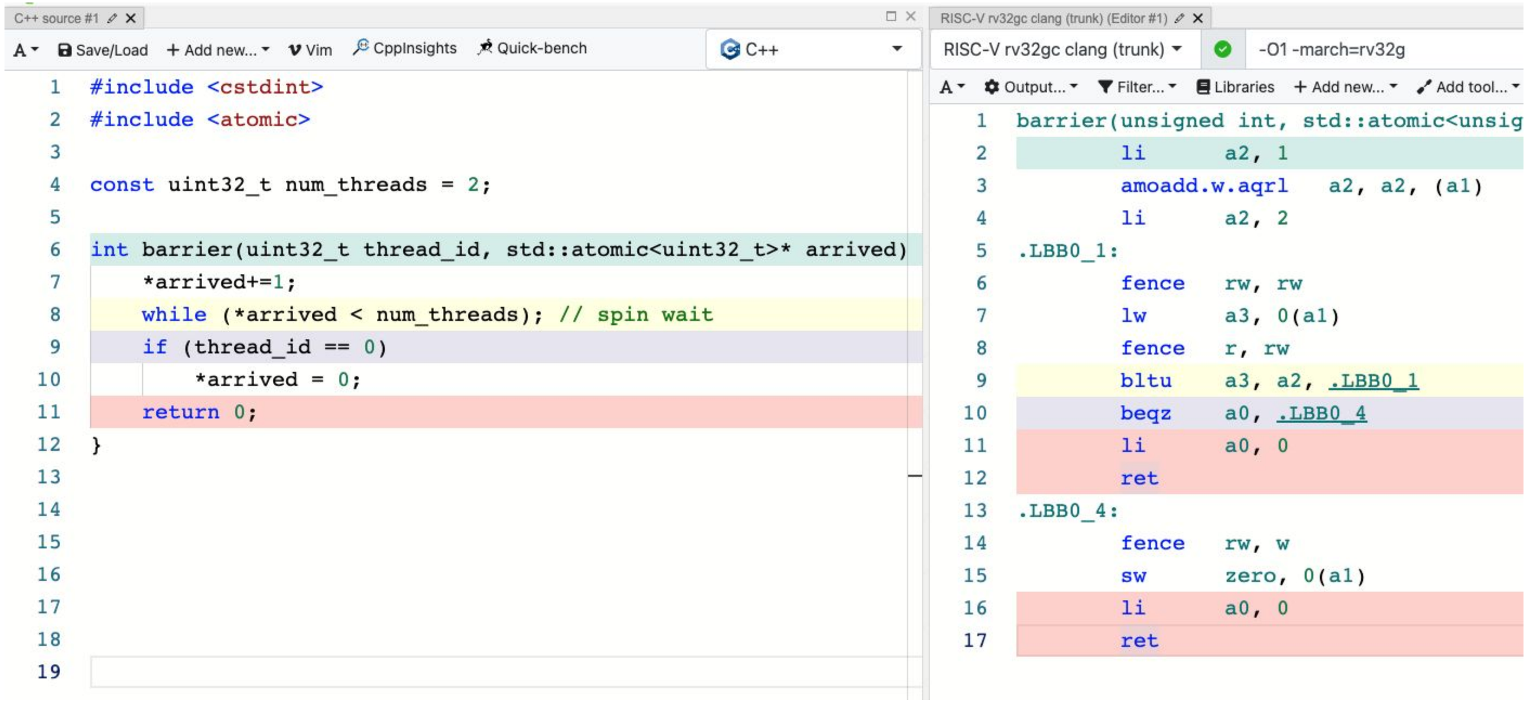
then do operation

then store

https://godbolt.org/z/b7xhszfo7

Barrier: Implementation 3 - Atomic access to a variable

- Side note: RISC-V supports such an operation in a more elegant way. Next lectures will covers the use of memory fences.



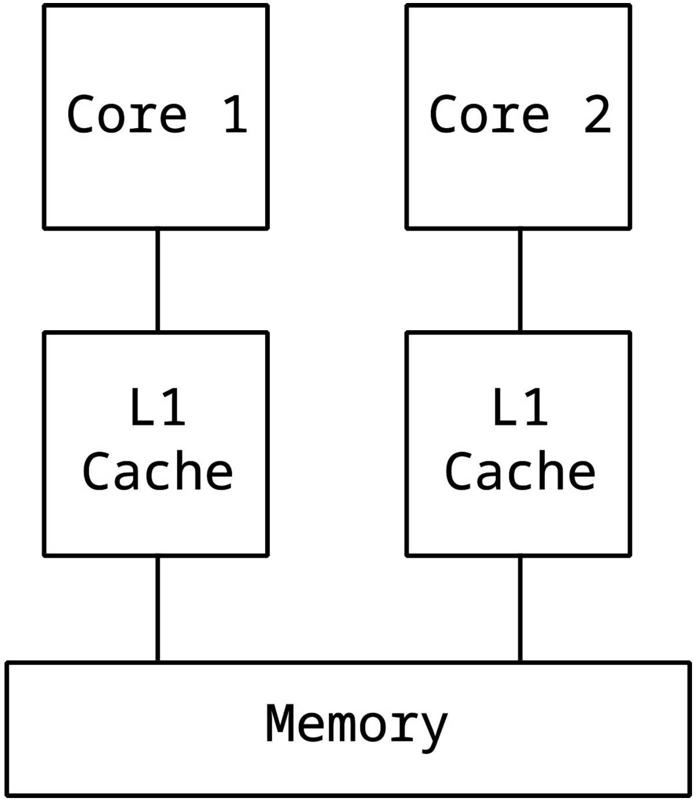
-O1 -march=rv32g

```cpp
#include <cstdint>
#include <atomic>

const uint32_t num_threads = 2;

int barrier(uint32_t thread_id, std::atomic<uint32_t>* arrived){
    *arrived+=1;
    while (*arrived < num_threads); // spin wait
    if (thread_id == 0)
        *arrived = 0;
    return 0;
}
```

```asm
barrier(unsigned int, std::atomic<unsig
        li      a2, 1
        amoadd.w.aqrl   a2, a2, (a1)
        li      a2, 2
.LBB0_1:
        fence   rw, rw
        lw      a3, 0(a1)
        fence   r, rw
        bltu    a3, a2, .LBB0_1
        beqz    a0, .LBB0_4
        li      a0, 0
        ret
.LBB0_4:
        fence   rw, w
        sw      zero, 0(a1)
        li      a0, 0
        ret
```

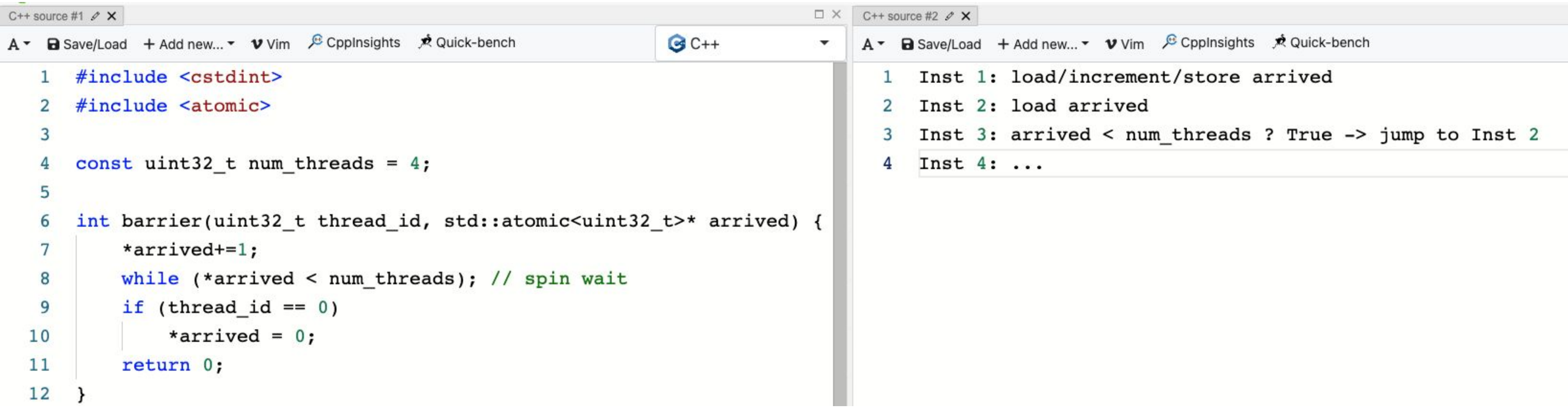
https://godbolt.org/z/a83sec7ss

Setup - Adding private caches



Implementation 3

- Simplified version of implementation 3



```cpp
#include <cstdint>
#include <atomic>

const uint32_t num_threads = 4;

int barrier(uint32_t thread_id, std::atomic<uint32_t>* arrived) {
    *arrived+=1;
    while (*arrived < num_threads); // spin wait
    if (thread_id == 0)
        *arrived = 0;
    return 0;
}
```

```
Inst 1: load/increment/store arrived
Inst 2: load arrived
Inst 3: arrived < num_threads ? True -> jump to Inst 2
Inst 4: ...
```

Problem with multiple caches

Thread 1:

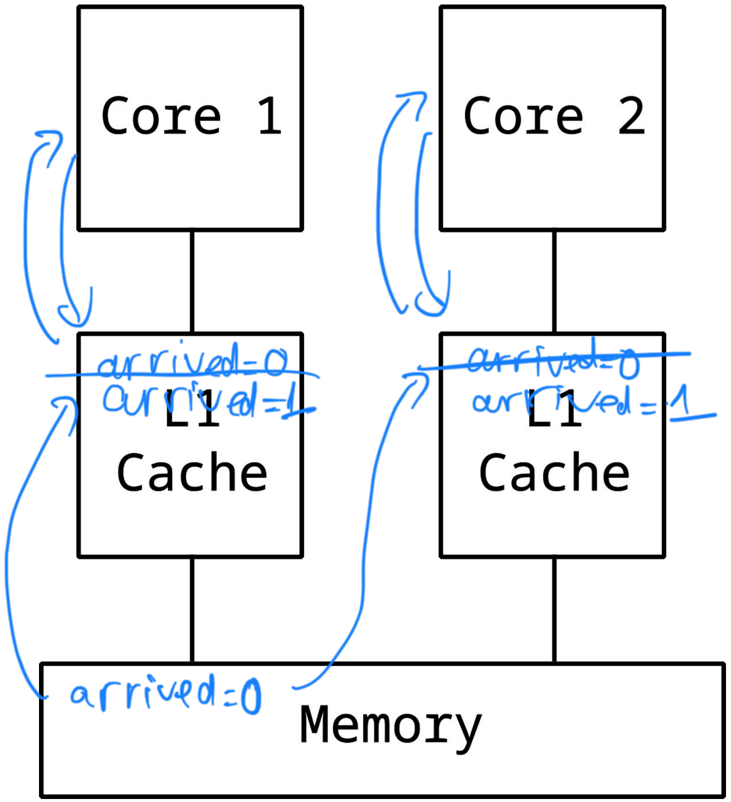
t_1 1. load-add-store arrived

t_2

2. load arrived
3. spin-wait

Thread 2:

1. load-add-store arrived
2. load arrived
3. spin-wait

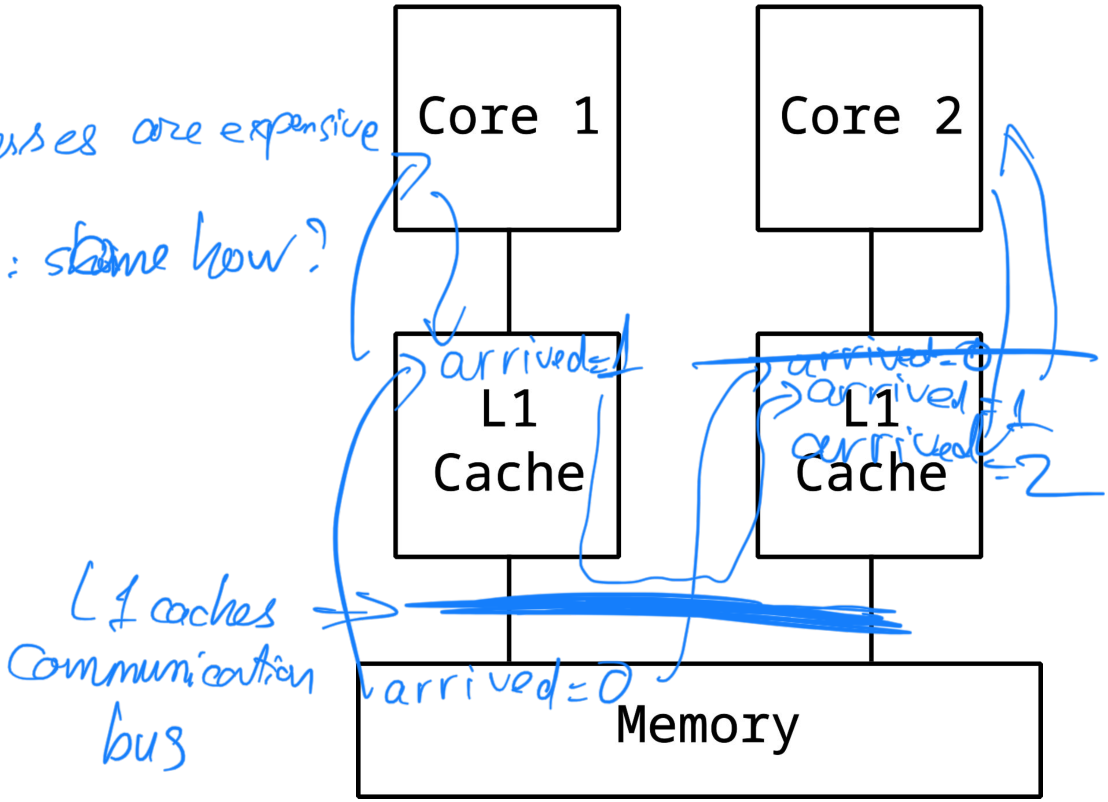


Problem with multiple caches

- Solution?

→ write - through ; $\overset{\text{store}}{\vee}$ accesses are expensive

→ cache communication: some how?


Core 1
Core 2
L1 Cache
L1 Cache
Memory
arrived=1
~~arrived=0~~
arrived=1
arrived=2
arrived=0
L1 caches communication bus