Method for Verifying Consistent Access Rules in a Multi-Agent, Multi-Resource System: A General Solution

(delivered to RISC-V tech-virt-mem group on 2021/04/28)

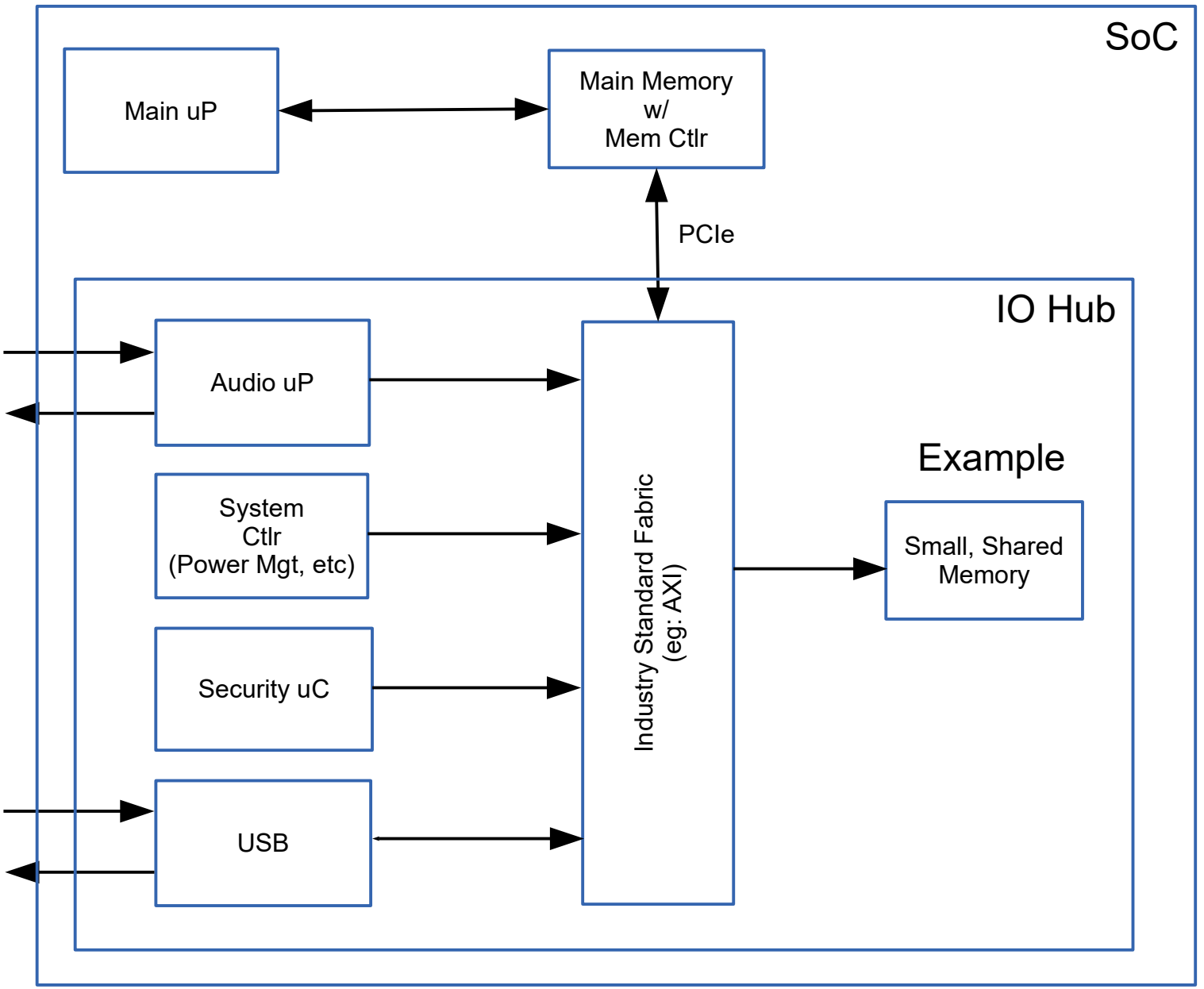
William C. McSpadden

-
- Architectural bugs are the hardest to find. And they are often the most expensive.

The Problem

- Unit level testbenches do a good job of checking out a building block. We know how to build Unit Level Testbenchs. However...
- System level features are different, and can be much more difficult.
  - Ordering rules
  - Performance
  - Access types
  - Arbitration

Example SoC



Problem: Memory Ordering

- Different architectures have different ordering rules. Main camps:
  - Strong: Reads and writes are not re-ordered or coalesced.
  - Weak: Aggressive re-ordering and coalescing.
- This can also be an issue for the memory controller in the chipset. Different attributes for different memory areas.
- Interrupt delivery is also a memory transaction (PCIe MSI/X, IOAPIC, etc.)

Problem: Access types

- Different agents can have different access types:
  - Endianness
  - Size (8/16/32/64 bit accesses)
  - Bursts (cache line, other)

Problem: Performance

- When all agents are engaged, are we meeting the performance goals (bandwidth, latency) of our system? How do we know, in a pre-Si environment?
- Are Quality of Service (QoS) requirements being met? Does the stimulus adequately stress the system? Are there checkers in place to flag QoS issues?

Problem: Arbitration

- I included this particular part of the problem, in order to emphasize that a **resource** is not just a memory. In a fabric, it includes utilization of busses, which is controlled with an arbiter. The arbiter should be considered as a resource.
  - Note: Most issues with the arbiter are found when doing performance measurements.

The Solution

- Set up a set of bi-directional buffers where 2 agents pass data. Access of the buffers are controlled with a semaphore.
- Want to make the solution as general as possible, something that can be readily ported from project to project, from testbench to testbench.

Peterson's Algorithm

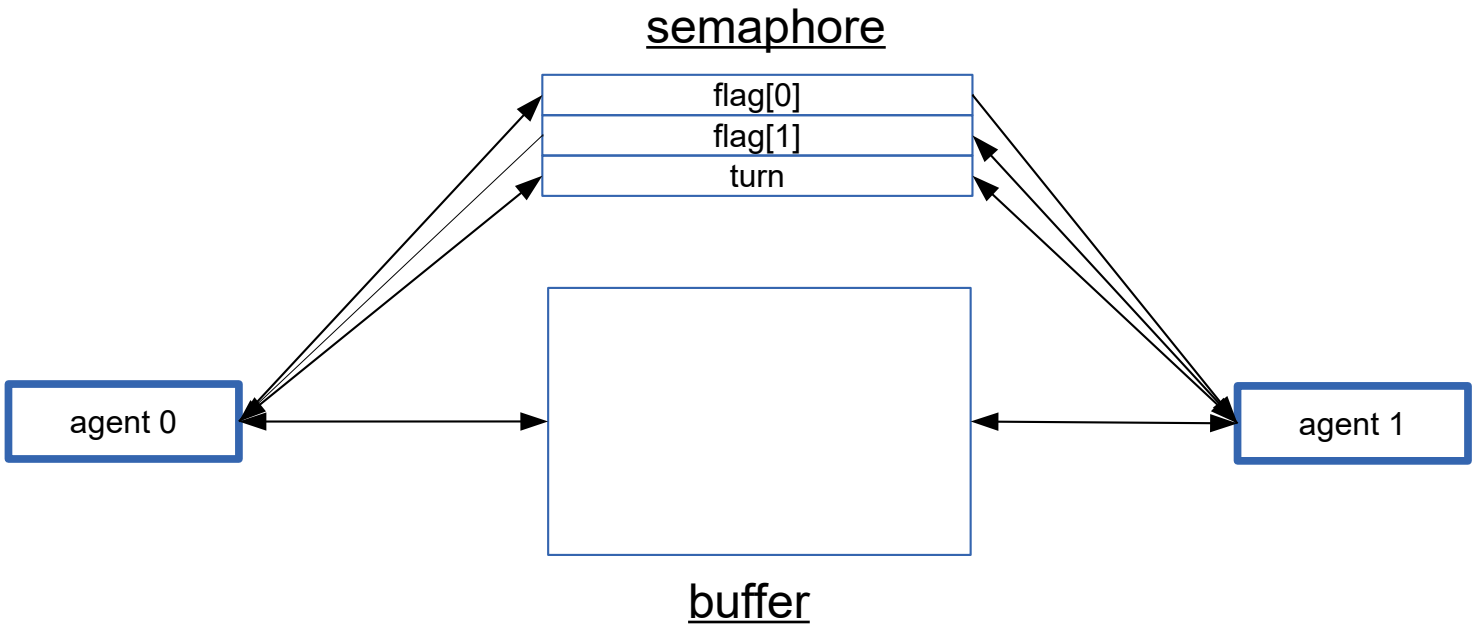
(taken from Wikipedia article)

```
bool flag[2]   = {false, false};
int turn;
```

```
P0:      flag[0] = true;
P0_gate: turn = 1;
         while (flag[1] && turn == 1)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[0] = false;
```

```
P1:      flag[1] = true;
P1_gate: turn = 0;
         while (flag[0] && turn == 0)
         {
             // busy wait
         }
         // critical section
         ...
         // end of critical section
         flag[1] = false;
```

Peterson's Algorithm



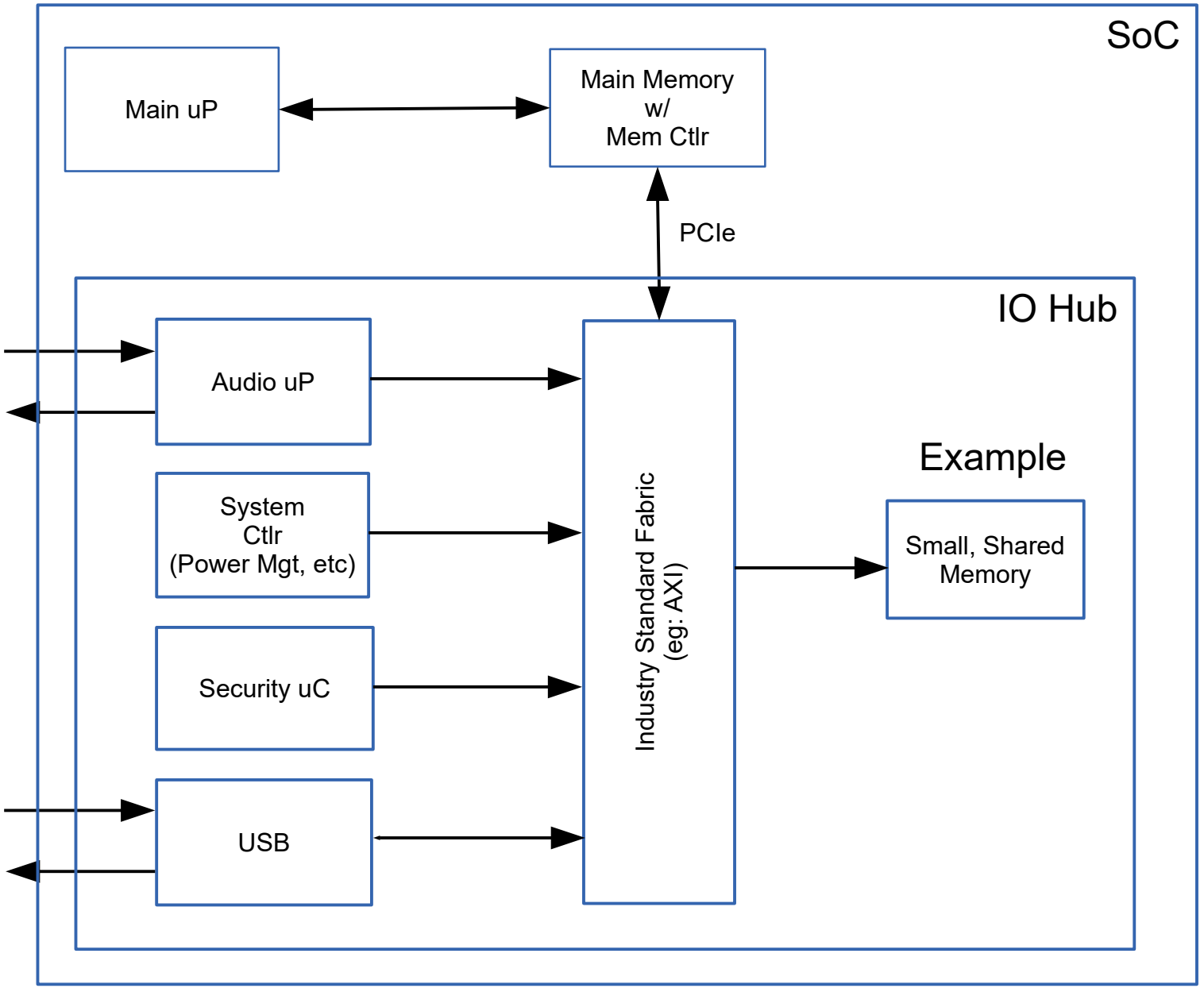
Memory Access Characteristics for the Semaphore

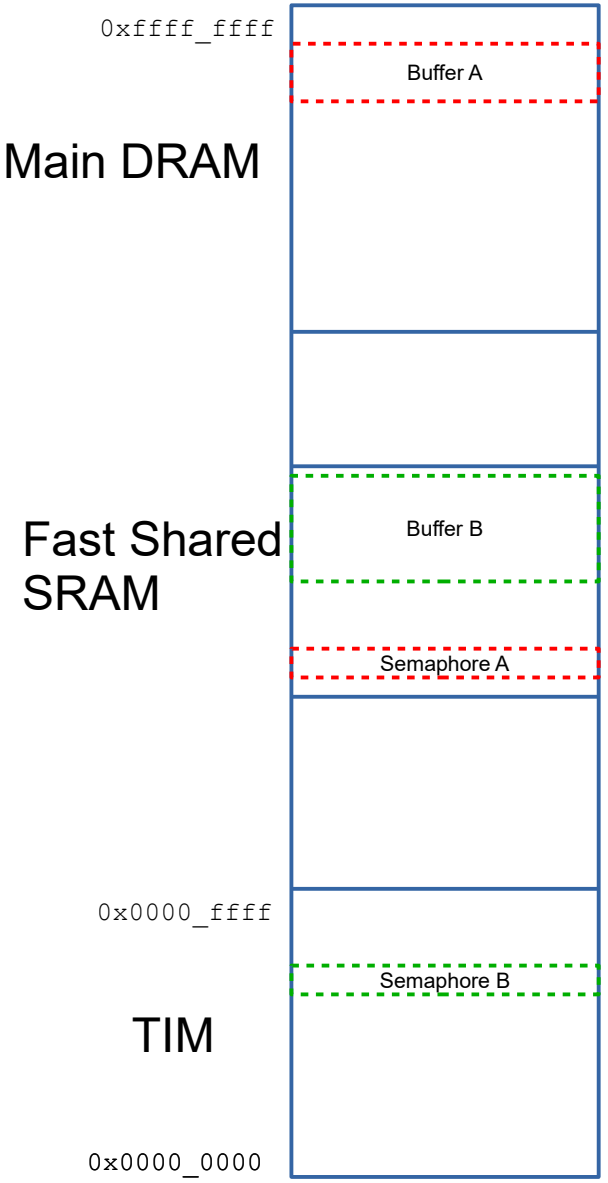
- Basic access types: ORD8, ORD16, ORD32, ORD64... The common access types supported by both agents.
- If single bit, must move the bit around.
- Atomic
- Access speeds: for finding bugs, fast is not always best.

Memory Access Characteristics for the Buffer

- All access types for each agent
  - Burst accesses!!
- bcopy()/memcopy()
  - Some uPs have specialized/optimized versions for performance. Use them!
- Top-to-bottom, bottom-to-top
- Data integrity check: checksum, CRC
  - Checksum may not catch address ordering issues; CRC would
- Length needs to be in bytes.

Example SoC





Randomize!

- Why randomize?
  - I have **never** worked on a project where a new random stimulus generator did not find a bug in the first 6 hours of use.
  - It's an industry-proven method. All modern methodologies use randomization.

Randomize!

- How to randomize this environment?
  - Randomize the number of buffer/semaphore pairs.
  - Put Semaphores and Buffers in random areas in memory.
  - Randomize the agents who share the pair.
  - Randomize the buffer data.
  - Randomize the access timing. That is, allow delays between semaphore and buffer acceses.

Weaknesses in the Method

- Uni-directional agents (can write or read, but not both). Think USB. Need to build a contrived solution.
- For more than 1 pair, a deadlock condition exists (“Starving Diner” problem). Need a mechanism for breaking deadlock.
- Not always reflexive: Upstream interrupts but downstream semaphores.

Colophon

- Presentation prepared using LibreOffice Impress and Draw, 4.1.2.3, on a MacBook.
- Exported to PDF using LibreOffice “export” function.