More Synchronization
Sequential Consistency, and the Flush Directive
Memory Consistency Issues and OpenMP

• Let’s start with an old “schematic” of a computer
• Instruction memory, CPU, and data memory
• Each arithmetic instruction is like: “add m1, m2, m3”
  • m1,m2,m3 are memory locations
  • Bring contents of m1 and m2 to ALU, add them up, and store the result in m3
• In addition, there are branch instructions
If Processors Remained That Simple ...

• If parallel computers with shared memory were built from such processors
  • Parallelism would be simpler
  • But they’d be slow

• Computer architects added caches, as we know
  • Some variables may exist in caches as well as memory
  • Cache coherence protocols, such as MESI for snoopy caches, handle data in caches

• But also, they added registers
Complexity of Real Computers: Registers

• Real computers became more complex before they became parallel
  • Remember: mostly to deal with slow memory
• Around 1985, the RISC revolution led to load-store architectures
  • Instructions either:
    • Do arithmetic/logical operations on registers, storing the result in registers,
    • Do a load from memory into register, OR
    • Do a store from register into memory
Complexity of Real Computers: Registers

• Real computers became more complex before they became parallel
  • Remember: mostly to deal with slow memory

• Around 1985, the RISC revolution led to load-store architectures
  • Instructions either:
    • Do arithmetic/logical operations on registers, storing the result in registers,
    • Do a load from memory into register, OR
    • Do a store from register into memory

• Variables can be stored in registers
  • But the association between variables and registers is loose
    • Not visible to the hardware
    • May be “spilled” lazily to memory via “write buffers”
Write Buffers

• For sequential processors, how long should a “Store” instruction take?

• You can speed it up by using a write buffer between the CPU and cache

• But this becomes problematic in parallel
Complexity: Compilers Can Reorder Statements

• Remember, compilers were written for sequential processors
• For example it may transform the first block of code below to the second block, by reordering statements
  • Why? Many reasons.. E.g. reducing the number of registers used, or eliminating a pipeline bubble

```
x = e1; // e1 doesn’t contain y
y = e2; // e2 doesn’t contain x
```
```
y = e2; // e2 doesn’t contain x
x = e1; // e1 doesn’t contain y
```
Complications with Parallel Processors

• Caches were handled using the extra hardware (snoopy cache controllers)
• But:
  • Data in registers
  • Data in “write buffers” (on its way to memory)
  • Instructions reordered by compiler
• When used with parallel processors, these tend to destroy our intuitive understanding of how parallel processors should behave, especially wrt memory
  • Notions of causality, program order, happens-before relation
• Our intuition is captured by the formalized notion of “sequential consistency”
  • Corresponds to the the simple picture of processor and memory we sketched earlier
Sequential Consistency

• This is a “desired property” of parallel programming systems

• The effect of executing a program consisting of k threads should be the same as some arbitrary interleaving of statements executed by each thread, executed sequentially

Modern processors do not satisfy sequential consistency!
Initially: $x$, Flag, are both 0

Thread 0:

$x = 25$;

Flag = 1;

Thread 1:

while (Flag == 0) ;

Print $x$;

What should get printed?
How to deal with lack of sequential consistency?

• Solution? Give up on registers and write buffers? No way!
• Various complicated processors specific synchronization primitives
• OpenMP provides simple machine-independent flush primitives
#pragma omp parallel {
    if (omp_get_thread_num() == 0) {
        // Thread 0: Producer
        Data = computeData();
        flag = 1;
    }
    if (omp_get_thread_num() == 1) {
        // Thread 1: Consumer
        while (flag == 0) {
        };
        print Data;
    }
}
#pragma omp parallel {
    if (omp_get_thread_num() == 0) {
        // Thread 0: Producer
        Data = computeData();
        #pragma omp flush (Data)
        flag = 1;
        #pragma omp flush (flag)
    }
    if (omp_get_thread_num() == 1) {
        // Thread 1: Consumer
        while (flag == 0) {
            #pragma omp flush (flag)
        }
        print Data;
    }
}
Naming Variables in the Flush Directive

• If no variables are named, all are flushed
  • i.e., the processor waits until all writes from it in the past are visible and all registers are stored out to memory/cache (spilled, flushed)

• **flush** is especially useful for point-to-point synchronization
  • i.e. for one thread to signal to another (that some event has happened or some data is ready)