David Greenberg

6.111 Final Project Proposal

Theory

Idea

Deterministic multiprocessing systems can provide important benefits in many applications. A deterministic multiprocessing system has the same output given the same input. For our purposes, we will fix the inputs to the system to be the code that's running and the data supplied to the program when it starts running or read by the program at well-defined times. This precludes nondeterminism introduced by button presses by forcing it to be only read in a deterministic way.

Concept

This implementation of deterministic multiprocessing will essentially be a time-slicing multitasking system in which the inherent parallelism of the problem is recovered across the multiple CPU cores. In order to recover determinism, we must look at how a time-slicing multitasking system orders reads and writes to memory. Essentially, reads and writes only must ordered on the inter-time slice level, since within a single time slice the ordering of the reads and writes taking effect is given by the order of instructions executed. We will snoop inter-thread communications and enforce a deterministic ordering on them. By enforcing determinism at this point, we will produce a deterministic system.

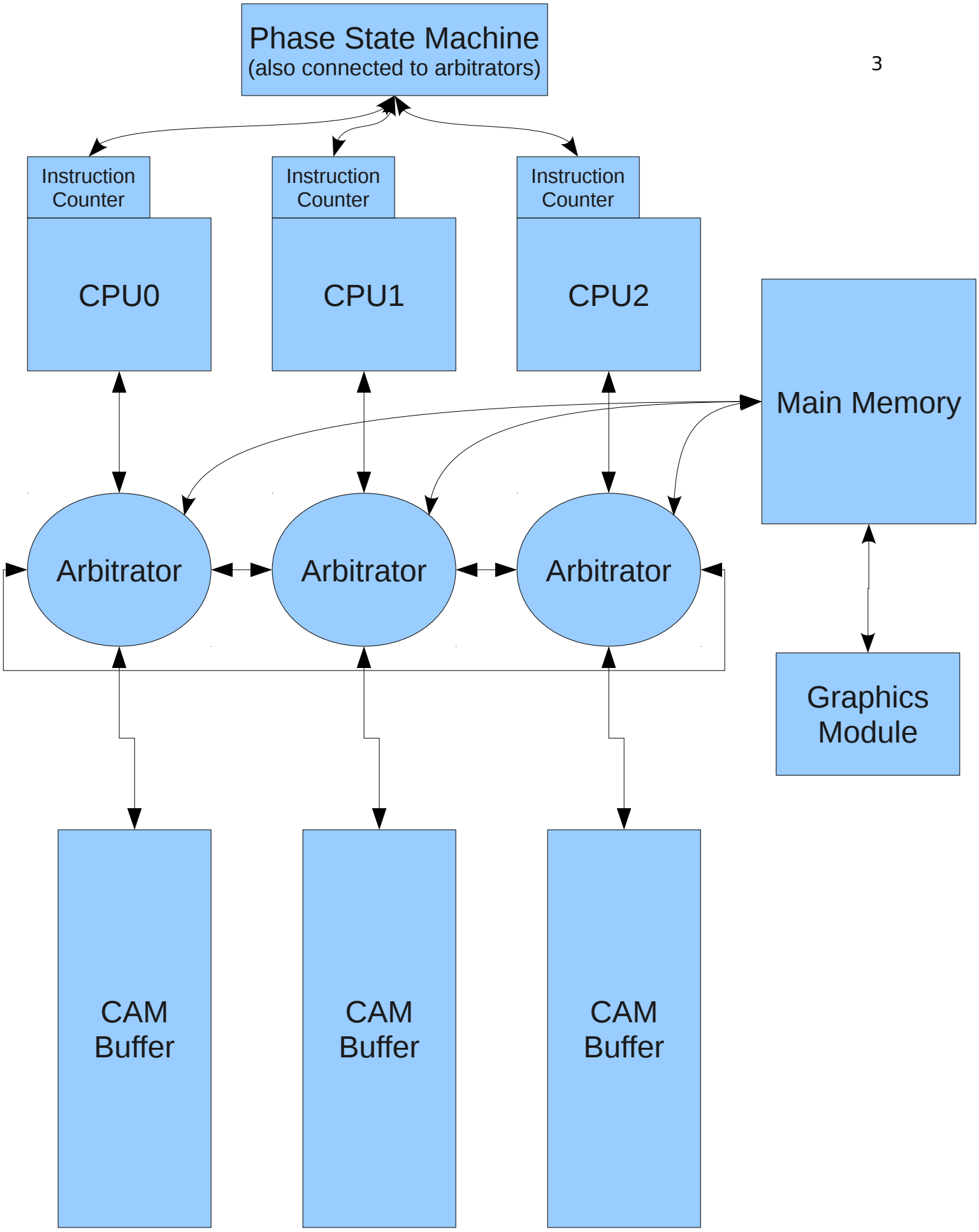
High Level Design

In order to enforce determinism between time slices, we will implement this system as a cache that sits between the processors and the main memory. To account for the time slices, each processor will incorporate an instruction counter, as this gives us fine-grained deterministic times-slicing (a timer interrupt would not work because it would be nondeterministic if the interrupt didn't have the highest preemption priority). We will use a scheme similar to CoreDet in which we buffer all loads and stores in per-processor caches. Since we store all this information, we can construct a snooping protocol that will detect communication between threads. When threads communicate, their communication must be handled deterministically. Otherwise, we can rely on each processor to remain self-consistent, and allow them to run without synchronization until communication is detected (or the end of a timeslice is reached). Synchronization is handled by a protocol that causes all pending writes to be flushed quickly and deterministically.

Design Details

The system operates by moving through three phases: the parallel phase, the commit phase, and the serial phase. The parallel phase is when the bulk of the work is done by all processors in parallel. The processors halt when they begin to communicate or reach the end of their time slice. When all processors are halted, they move to the commit phase, where pending writes are flushed to main memory in a deterministic order. Finally, the serial phase happens, when each processor is given a second, short time slice to run as the only processor in the system. This allows short extents of operations on mutexes to occur without causing most of a time slice to be wasted due to the processor halting due to communication.

The system is broadly divided into 3 components: the Processors (CPUs), the Arbitrators, and the CAM (content addressable memory) buffers. The CPUs execute and issue reads and writes to the arbitrators. The arbitrators maintain coherency in the CAM Buffers, detect communication between threads, and execute the algorithms. The CAM buffers provide a fast way to check what memory locations have been modified to maintain the protocol. In addition, there is a state machine that maintains which phase is currently being executed, as well as several support modules like the instruction counters, main memory controller, and graphics output. On the following page is a block diagram.


Phase State Machine
(also connected to arbitrators)
Instruction Counter
Instruction Counter
Instruction Counter
CPU0
CPU1
CPU2
Main Memory
Arbitrator
Arbitrator
Arbitrator
Graphics Module
CAM Buffer
CAM Buffer
CAM Buffer

Parallel Phase

The parallel phase takes place until the instruction counter has expired or an arbitrator has detected communication. Thus, the majority of the complexity in the components that run this phase is in detecting communication, since instruction counting and processor halting are trivial operations. Each CAM buffer slot must store the address of memory that was accessed, the data that is in that location after the most recent access, and three additional bits of information (the tag): whether the address has been accessed Exclusively by the local processor or whether it is Shared, whether the address has been Read or not (0), and whether the address has been Written or not (0). Example states of a memory location could thus be E/R/W (exclusively held, read and written) or S/R/0 (shared, read-only). The following table describes how the state transitions must take place. Any transition not listed in the table represents a communication and thus results in a halt until after the deterministic commit phase. From the perspective of each arbitrator, a memory address can either be found in the local, directly connected CAM buffer or found in one of the other arbitrator's CAM buffers (a friend buffer).

On Read

Found in	Valid Tags	Change Tag To
Local CAM Buffer	E/?/? or S/R/0	E/R/? or S/R/0
Friend CAM Buffer	E/R/0 or S/R/0	S/R/0
Not found (fetch)	---	E/R/0

On Write

Found in	Valid Tags	Change Tag To
Local CAM Buffer	E/?/? or S/0/W	E/?/W or S/0/W
Friend CAM Buffer	E/0/W or S/0/W	S/0/W
Not found (fetch)	---	E/0/W

Reads and writes will proceed in the above-defined way until every CPU halts. CPUs halt for three reasons: during a read or write operation, an address was found in a local or friend buffer but wasn't in a valid state; the instruction count for that CPU was reached; or the CAM buffer for that CPU overflowed.

The snooping protocol will work as follows. Every arbitrator will have a connection to every other arbitrator. When an arbitrator wants to know whether a friend has a piece of memory in its CAM buffer, it signals all arbitrators that it needs to read or write some address. Then all of the other arbitrators will signal the requesting arbitrator when they've determined that they A) do not have that address, B) have that address and the pending operation can continue, or C) they have that address and the pending operation must be deferred until the commit phase runs. This interconnect protocol allows the address bits to be shared, so that for K processors and an N bit address bus, N+2 bits are required to signal the other arbitrators (address bus, start, read/write), and each arbitrator needs 2*K bits to get responses from the others (A, B, C, or invalid encoded in two bits).

Commit Phase

The commit phase runs after all CPUs have been halted. The commit algorithm works as follows:

1. For each CPU in a deterministic round-robin order:
    1.1. For each written CAM buffer cell in chronological order:
        a) Write out modified bytes in buffer cell
        b) Notify other arbitrators that the given address has been written once and should not be written again
            - This causes the other arbitrators to erase their copies of this address
        c) Mark this cell as empty
    1.2. Mark all CAM buffer cells invalid

This algorithm is easily executed by having each arbitrator flush its pending writes in deterministic order and using the communication channels from the parallel phase to notify the other arbitrators as addresses are committed.

It is important to note here that this system is not exactly a cache, since it periodically flushes all of the cached data regardless of whether or not it has been modified. This is necessary to prevent stale data from being kept in the cache after a commit cycle.

This algorithm is intended to be amenable to deep pipelining, which should mitigate any performance impact caused by this stage.

Serial Phase

There may not be enough time to implement this phase, as it only exists to improve performance by exploiting locality in lock acquisition. It only requires use of the instruction counter and a simple round-robin state machine, and it essentially bypasses the CAM memory, causing all reads and writes to be directly read from or written to main memory.

Other Elements

Memory Controller

The main memory controller will have to multiplex access to the ZBT RAM in the labkit so that all of the arbitrators can access it. It will use a request queue to schedule memory operations in the order they're received. This request queue will be of limited size, and will attempt to be fair in the order it services requests from arbitrators. For the commit phase, there may be an alternative mode that exposes the pipelining available in the ZBT RAM to increase performance.

Graphics Device

A region of main memory will be mapped to the VGA output controller from Lab 5 for use as a frame buffer for demonstration applications.

Processors

The processors will be the PicoBlaze compatible processors we went over in lecture due to their simplicity unless there is enough time to integrate with a more complex architecture. They will need to modified to support reading and writing to another memory controller, arbitrary halting, and instruction counting.

Instruction Counter

Each processor will be retrofitted with a programmable instruction counter that signals when a specifiable number of instructions has been reached. This will be used by the halt detection logic and the serial phase.

Testing

For the three phases of operation, testing will be relatively straightforward, as each phase is essentially an algorithm that maintains a set of invariants. The testing code will apply a variety of inputs to the state machines and ensure that the invariants are maintained. Nondeterminism will be introduced by varying the clock frequencies of the processors to simulate cache misses, pipeline bubbles, and bus protocol overheads. Every module will be heavily tested prior to synthesis in order to ensure that the entire system behaves exactly as expected.

Hardware Usage

Each PicoBlaze processor takes very few slices of logic (on the order of 100). The arbitrators shouldn't take more than the processors, and the CAM buffers should take very few slices and no more than 3 BRAMs. If there are enough BRAMs available, I will not implement a ZBT memory controller and just use the other BRAMs to simulate main memory. No other hardware will be needed.