Announcements

- Reading for this class finish Chapter 10
- Project returned today
  - generally good, but some didn't run on my data
    - please see me if this was the case for your program
  - many people forgot to check the return value of pvm funcs

Dataflow Hardware (Manchester)

- token (96 bits)
  - 37 bits for data (5 for type)
  - 36 bits for tag (activation name, iteration, array index)
  - 22 bits for destination (graph node # and input port)
  - 1 bit marker (how many nodes required for enable)
- memory
  - hold program - as execution graph
  - temporary storage of waiting tokens
    - 20 parallel 64K-tokens
- matching unit
  - match tokens with same tag destined for same insn node
  - simulates associative memory with a hash table
    - if match not found, must store token in memory

Manchester (cont.)

- Performance
  - Matching: 1.29M matches/second
  - 0.23MIPS per Function unit
  - 20 units keep the matching unit busy
- Pipeline information
  - 27-31 cycles latency in the pipe
  - memory (array access) can add an extra 15-20 cycles
- Required Parallelism
  - 30-50 per PE
  - need about 20 PE's to be worth hardware costs
  - thus require parallelism in the thousands for "small" machines

Static vs. Dynamic Dataflow

- Static
  - only one token instance per arc
  - finite (compile time) storage requirements
  - only permits pipeline concurrency
  - use acknowledgments to permit reentrancy
- Dynamic
  - multiple instances of tokens per arc
  - in general could required unbounded storage
  - use tags to identify instances (for reentrancy)

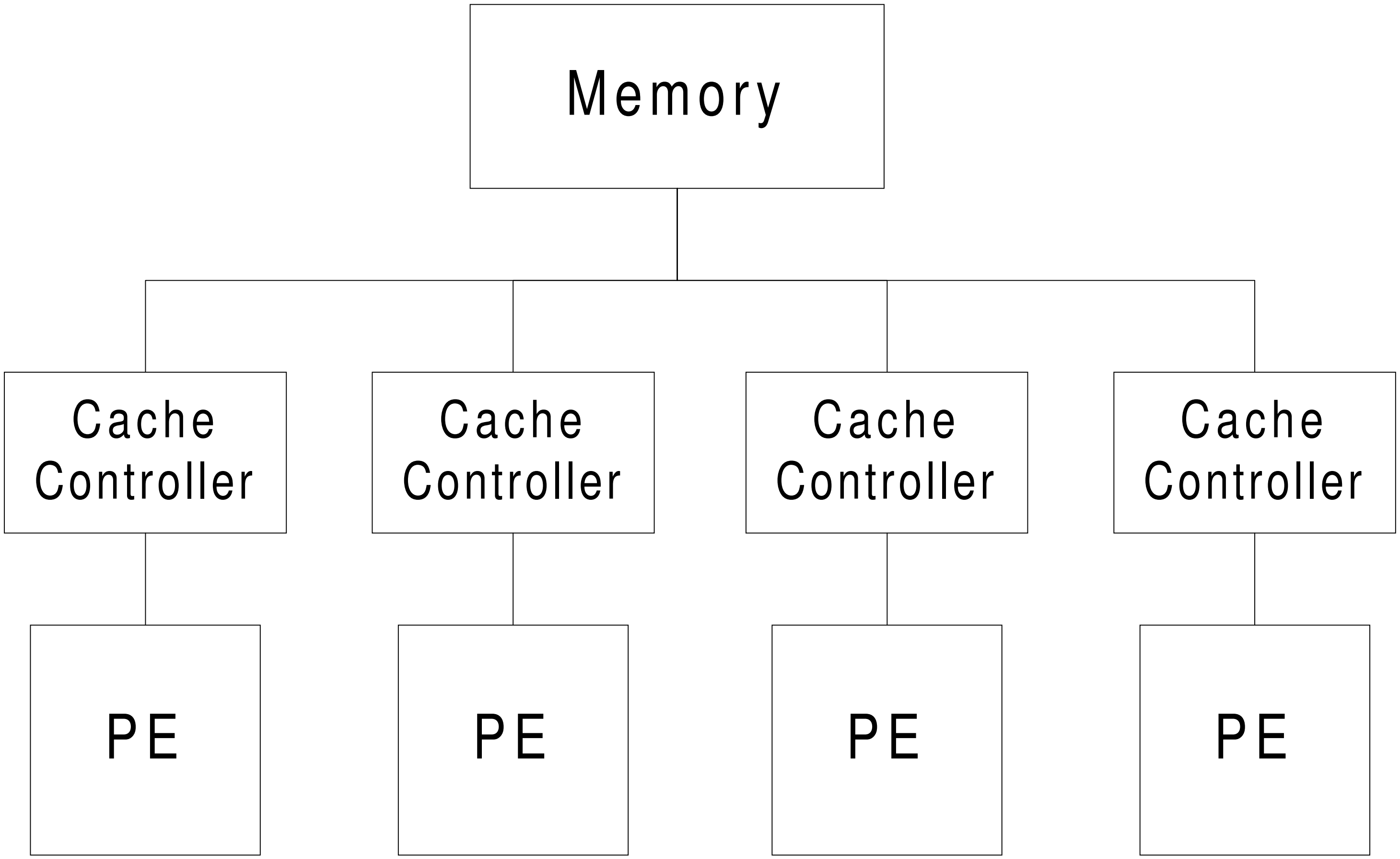
Data Flow Issues

- How to represent structures?
  - one giant token
  - a whole bunch of small tokens
  - special case "structure store"
    - token becomes an index into the structure store
    - this changes the basic dataflow model
- Programming
  - requires different language to extract parallelism
- Hardware
  - requires fully custom design
  - will never be able to keep pace as long as micro-processors remain on a super-linear speed vs. time curve

Bus based Multiprocessors

- biggest commercial success of parallel computing
- limited number of processors can share a single bus
- use caches to keep the bus traffic lower
  - the cache is useful even if it is not that much faster than main memory
  - caches act as private memory to reduce bus requests
- Cache coherency
  - need to ensure that each processor get latest version of data
  - how soon does a processors sees changes by other processors is a design parameter
    - by the next instruction “sequential consistency”
    - by the next synchronization operation “released consistency”

Basic Structure of Bus-based SMP



Cache Coherency (simple)

- Read only values are cached
- Writeable values
  - not cached, all reads and writes go to main memory
  - good performance for frequently updated values
    - if many processors update the same location
  - poor performance for
    - many updates by the same processor
    - infrequent updates and frequent writes
- Who marks regions for caching?
  - static: compiler marks shared writeable areas
  - dynamic: runtime support to change cachability of lines
    - compiler emits code to change status
    - user makes explicit calls

Cache Coherency (write through)

- Read only data cached
- Writeable values can be cached by one processor
  - a processor needs to gain write access
    - must force invalidation of other cached copies
  - all writes go back to main memory
  - reads can be served from cache for processor with write access
- Performance
  - good for
    - updates and reads by same processor
  - bad for
    - multiple updates by the same processor (many bus writes)

Cache Coherency (write back)

- Read only lines cached by multiple processors
- Writeable lines cached by one processors
  - only write value back to memory when needed
    - due to cache line eviction of updated value
    - due to need for other processors to see the value
- Performance
  - good for
    - many reads/writes by the same processor
  - cost is increased complexity for
    - write backs
    - more complex to know where the “good” value is located

How to Manage Caches

- Snooping
  - each cache controller watches bus for "interesting" info
  - may result in cache lines being invalidated if write seen
    - i.e. a write through cache
  - limited by speed of cache controllers to watch the bus
    - must see everything to maintain correctness
- Directories
  - memory stores information about cached copies
  - does not require each cache controller to snoop
  - permits more scaleable interconnect networks

Directory Based Cache Controllers

- Requires additional circuits to maintain directories
- directories must be updated when a processors
  - starts caching a value
  - stops caching a value
  - changes from read to write caching (or back)
- each cache line has a directory entry
  - can use sparse schemes that only have entries for actively cached items
- can have several memory controllers in a machine
  - each manages a region of physical memory
  - bit vectors (one bit per processor)
  - addresses (several $\log_2 n$ entries)

Representing Directories

- bit vectors
  - one bit per processor
  - uses lots of space for a large machine
  - permits each processor to cache a value
- addresses
  - several entries for PE id (each entry is $\log_2 n$ bits)
  - what happens if a processor wishes to cache, and all entries are full?
    - use a linked list of directories (SCI uses this approach)
    - use a “wildcard” and force a broadcast to invalidate