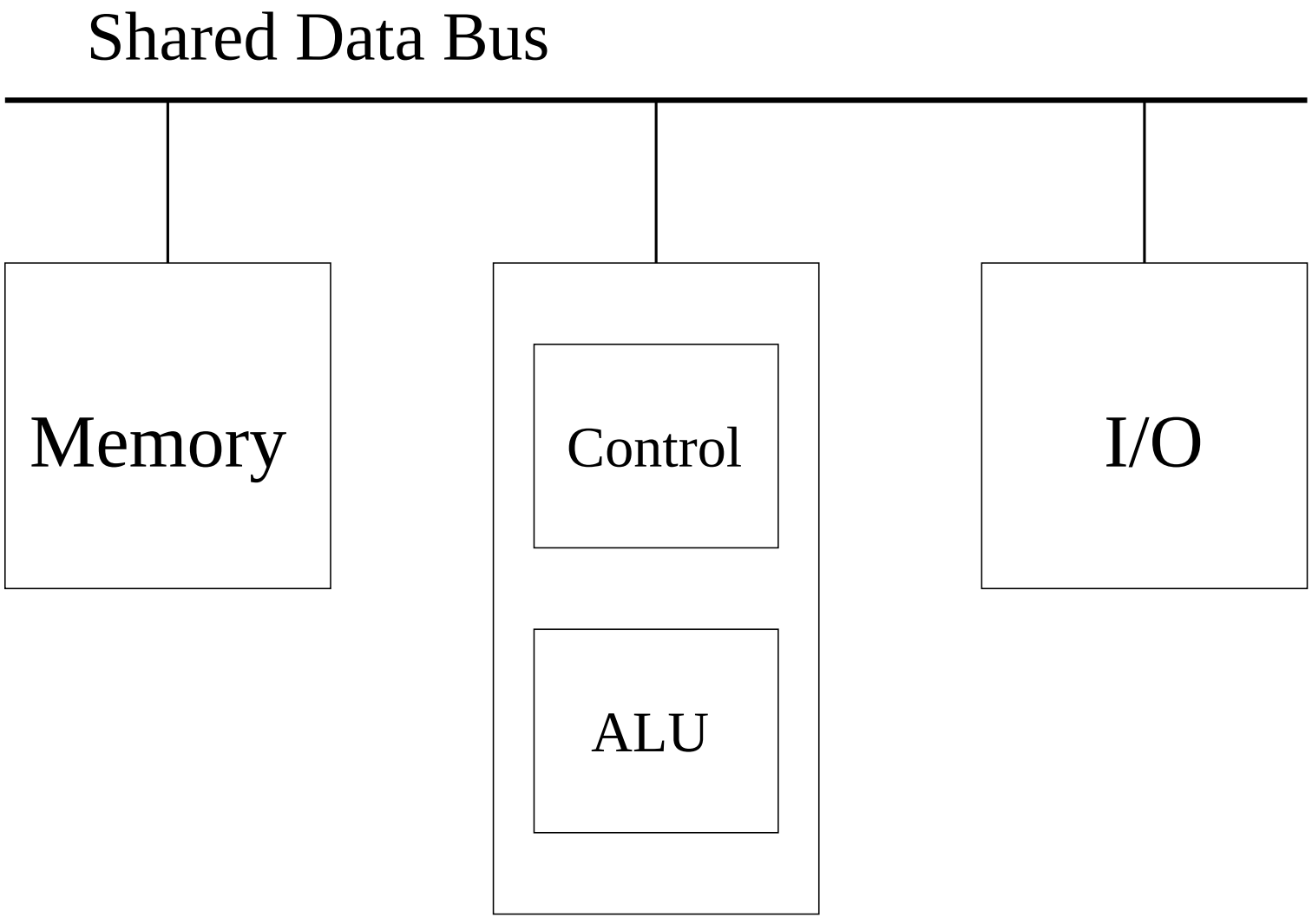
Von Neumann Architecture



Memory Hierarchy

Type	Size	Access Time
Registers	1Kb	2ns
L1 Cache	16Kb	2ns
L2 Cache	2Mb	4ns
RAM	4Gb	50ns
Disk	1000G	5ms

- Data is copied up and down the hierarchy as required

Registers

- At the top of the memory tree
- Located on chip — very fast access
- Usually 1Kb or so with access in 1 clock cycle
- Used as immediate data storage for processor
- Initially processors had a small number of special use registers
  - Accumulator
  - Index register
  - Program counter
  - Stack pointer
- Modern processors have many general purpose registers

Register Usage

- Assembly language subroutines are like functions in higher level languages
- They use registers to store the data they are manipulating
- Therefore when a new subroutine is called the old values in the registers need to be saved
- When the new subroutine finished the old values are restored and execution continues as before
- Two methods for doing this
  - Push register values to stack in main memory (x86)
  - Use a register file with a sliding window (RISC)

Register Usage

- Backing up and restoring register values on the stack is relatively slow
- A register file is a collection of many registers of which only some are visible to the code at any one time
- When a subroutine is called the window changes to a different set of registers — often only 1 clock cycle required to switch
- Use overlap of windows to pass parameters between subroutines

Caches

- A cache is a collection of data that duplicates sections of main memory closer to the CPU than main memory
- A temporary storage area where frequently accessed data can be stored for rapid access
- Caches divide information into lines which are loaded from memory
- Each line contains actual data as well as meta-data associated with the data
  - Address tag of data being cached
  - Validity
  - Shared/Exclusive
  - Clean/Dirty

Cache Operation

- When the CPU wants to read/write a location in memory it checks if that data is in the cache
- Compares the address required with all lines in the cache where the data might be stored
- If the data is found a *cache hit* occurs, otherwise we have a *cache miss*
- With a hit the data is immediately read from the cache by the CPU
- Otherwise the cache fetches the relevant data from main memory and then passes the data to the CPU
- At no point does the CPU directly talk to main memory

Cache Organisation

- There are several ways to organise memory in a cache
  - Fully associative — any line can cache data from anywhere in memory (costly and complicated)
  - Direct mapped — any memory location can only end up in one cache line (simple but prone to thrashing)
  - Set associative — lines grouped together into a set, each of which can store data from the same collection of locations in memory. If a set contains n lines it is called n-way associative
- Associativity is a trade off
  - The more lines in a set the more likely it is for a piece of data to be in the cache
  - However it means more complex checking hardware which is costly in power, space and time

Cache Organisation

- In a set associative cache how do you determine which set you belong to?
- Use lesser significant bits to find out
- If a cache line is 16 bytes (2^4) we ignore the last four bits of the address
- Then we figure out how many sets there are — say 8 sets (2^3)
- Use the next 3 least significant bits as the set id
- We request address: 0010-1010-1101-1010
- Ignore the last 4 bits: 0010-1010-1101-1010
- So this data is placed in set 5 — 101_2

Cache Organisation

- Caches need to copy data back to memory after changes
- This is called the write policy
  - Write through: data copied back immediately after each write
  - Write back: data copied back whenever appropriate
- Caches need to have old data removed when they fill
- Policies include selection of random lines or the least recently used line (LRU)
- These policies can interact badly with code depending on the memory access pattern and the organisation of the code

Cache efficiency

- The effectiveness of a cache is measured by the hit rate
- This is the ratio of the number of times a requested piece of data is found in the cache to the total number of requests
- L1 caches try to have a hit rate of over 90%
- L2 caches with a hit rate of 50% are doing well
- Separate caches for instructions and data

Split Caches

- Often the L1 cache is split into two sections
  - Instruction cache
  - Data cache
- Avoid collisions between data streams and instruction streams
- Often possible to get entire functions into the instruction cache
- Sometimes referred to as Harvard caches

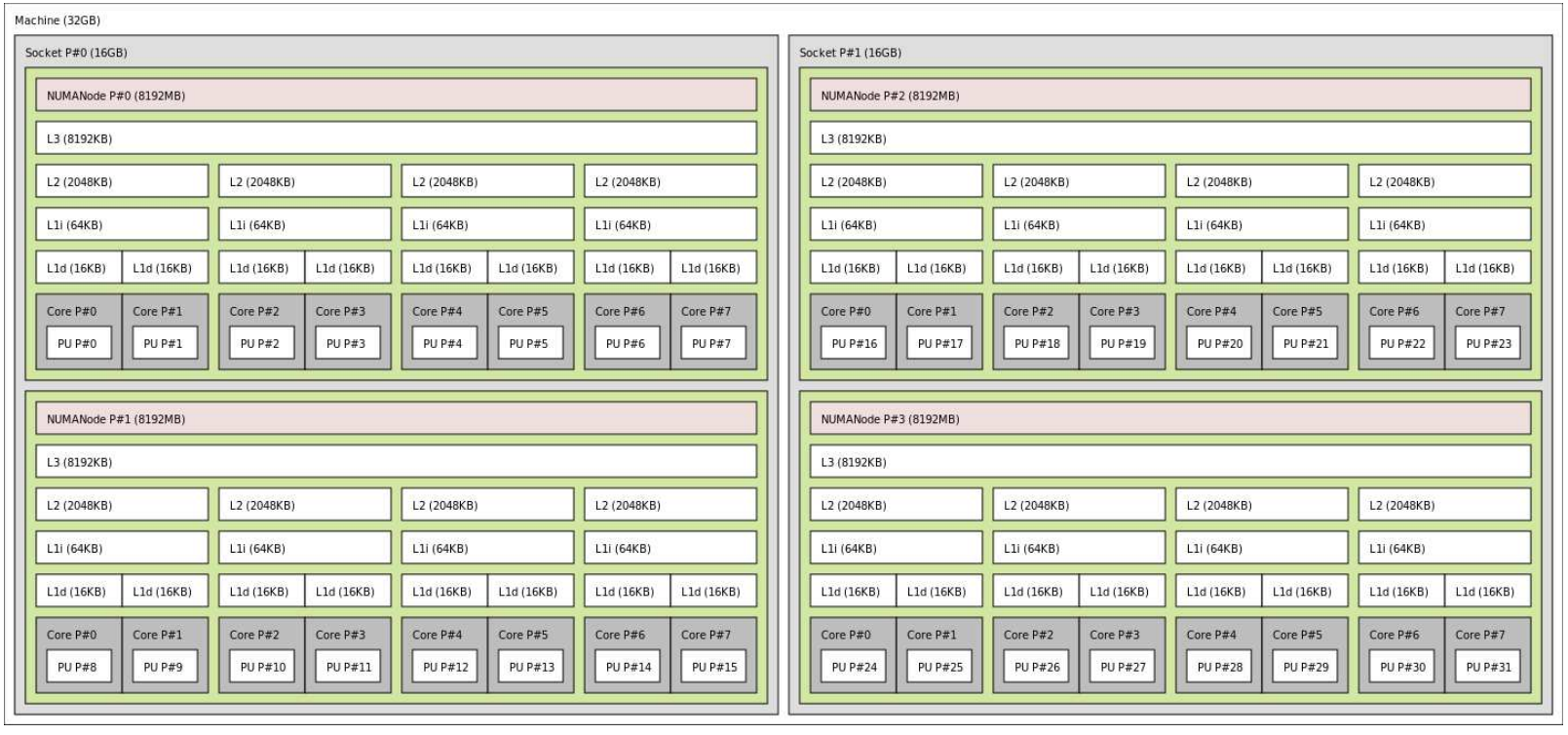
Specialist Caches

- *Trace Cache* — found in P4 processors
- Cache groups the decoded CISC instructions
- These sets are called instruction traces
- Only include the path taken
- *Micro-op cache* — Sandy Bridge and onwards
- Similar to trace cache but less complex
- Single decoded instructions rather than traces
- *Victim Cache* — stores data recently evicted from the main cache
- Usually fully associative
- Helps to avoid trashing problems

Multi-level Caches

- A large cache provides a good hit rate but poor (relative) performance
- Solve this by using a small fast cache backed up by a larger slower cache
- In modern processors up to three levels common
- Multi-level caches can be either inclusive (P4) or exclusive (Athlon)
  - Inclusive caches allow data to be in both L1 and L2
  - Exclusive caches do not
  - Strictly inclusive caches guarantee that data in L1 is also in L2 — useful for multi-processor systems

Multi-level Caches



Memory and cache layout from Bulldozer

Cache Coherency

- In systems with more than one compute core we need to balance the performance of having the same data in multiple caches with the need to have the correct value available
- This is called cache coherency
- Three main methods for addressing this
  - Directory protocols
  - Snoopy protocols
  - Software solutions

Directory Protocols

- A central controller keeps track of cached locations
- When a core requests permission to update a cached location the controller invalidates all other copies of the data
- When another read request is made for the same location the controller tells the locking core to flush the data back to memory
- The memory location can then become read-only in multiple caches again
- Main problem - central bottleneck and comms overhead

Snoopy Protocols

- Distribute responsibility for cache coherency between all the caches
- Caches monitor a shared data bus to know when lines they have are also loaded into other caches
- Caches designed to reduce bus traffic. Often use a dedicated snoopy bus for cache coherency
- Two basic approaches — write invalidate and write update
- Write invalidate is the most common
- Often called MESI protocol — Modified, Exclusive, Shared, Invalid

Software Solutions

- Transfer the onus to the compiler and operating system
- Often require every shared variable to not be cached
- Massively reduced performance
- Reduces complexity and overhead of processor design