ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

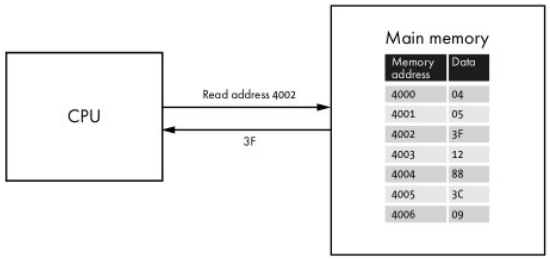
- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Storage Technologies

Introduction

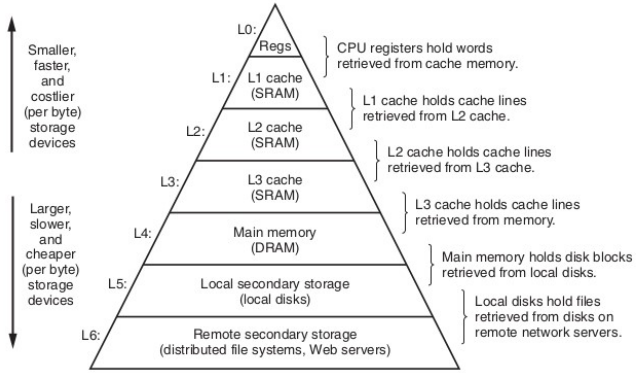
- Till now, we have relied on a simple model of a computer system as a **CPU that executes instructions** and **a memory system that holds instructions and data** for the CPU.
- In our simple model, the **memory system is a linear array of bytes**, and the **CPU can access each memory location in a constant amount of time.**
  - While this is an effective model as far as it goes, it does not reflect the way that modern systems really work.



- The gap between CPU and main memory speed is huge

Memory System

- The solution consists on a **memory system composed by a hierarchy of storage devices** with different capacities, costs, and access times.



- The central idea of a memory hierarchy is that **for each** k, the faster and smaller storage device at level k serves **as a cache for the larger and slower storage device at level** $k + 1$.

Volatile and Nonvolatile Memory

- Volatile memories **lose their information if the supply voltage is turned off**.
- Nonvolatile memories **retain their information even when they are powered off**.

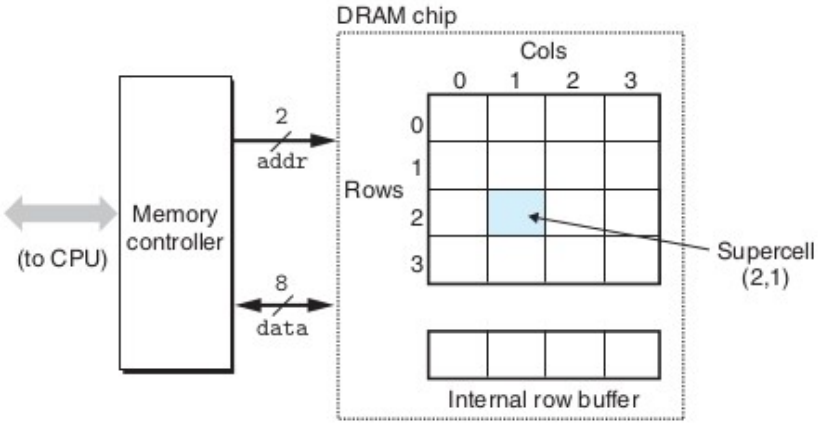
Random-Access Memory (RAM)

- Basic storage unit is usually **a cell** (one bit per cell)
- RAM is traditionally packaged as a chip
  - Multiple chips form memory
- Static RAM (SRAM)
  - Each cell implemented with a six-transistor circuit
  - Holds value as long as power is maintained: volatile
  - Insensitive to disturbances such as electrical noise, radiation, etc.
  - Faster and more expensive than DRAM
- Dynamic RAM (DRAM)
  - Each bit stored as charge on a capacitor
  - Value must be refreshed every 10–100 msec: volatile
  - Sensitive to disturbances
  - Slower and cheaper than SRAM

Conventional DRAM Organization (I)

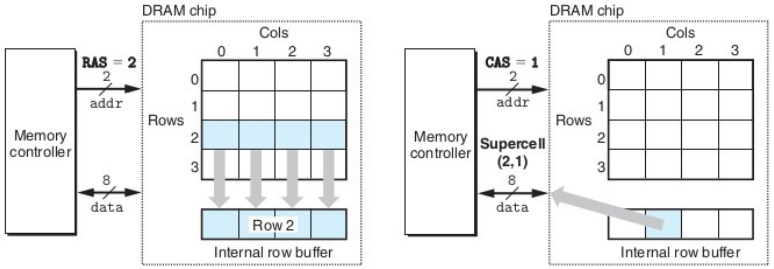
- The cells (bits) in a DRAM chip are partitioned into d **supercells**, each consisting of w DRAM cells, storing $d * w$ **bits** of information.
- The supercells **are organized as a rectangular array with** r **rows and** c **columns**, where $r * c = d$.
- Each supercell has an address of the form (i, j), where i denotes the row, and j denotes the column.
- Information flows in and out of the chip via external connectors called **pins**.
  - Each pin carries a 1-bit signal.
    - There are eight data pins that can transfer 1 byte in or out of the chip, and two addr pins that carry two-bit row and column supercell addresses.
- Each DRAM chip is connected to the **memory controller**, that can transfer w bits at a time to and from each DRAM chip.

Conventional DRAM Organization (II)



Reading the contents of a DRAM supercell (I)

- To read the contents of supercell (i, j), **the memory controller sends the row address i to the DRAM, followed by the column address j.**
  - The row address i is called a RAS (Row Access Strobe) request.
  - The column address j is called a CAS (Column Access Strobe) request.
  - Notice that the RAS and CAS requests share the same DRAM address pins.



- The DRAM responds **by sending the contents of supercell (i, j) back to the controller.**

Reading the contents of a DRAM supercell (II)

- One reason circuit designers organize DRAMs as two-dimensional arrays instead of linear arrays **is to reduce the number of address pins on the chip**.
- For example, if our example 128-bit DRAM were organized as a linear array of 16 supercells with addresses 0 to 15, then the **chip would need four address pins** instead of two.
- The disadvantage of the two-dimensional array organization is **that addresses must be sent in two distinct steps**, which increases the access time.

Nonvolatile Memory (I)

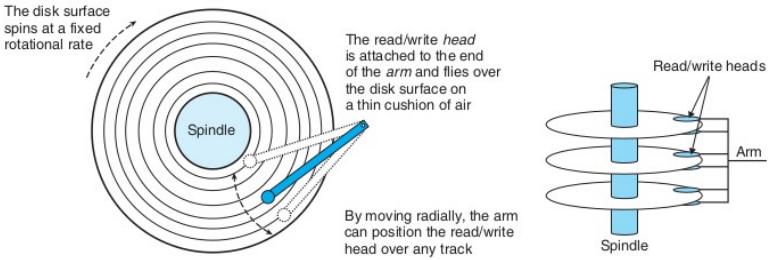
- **Information retained if supply voltage is turned off.**
- Collectively referred to as **read-only memories (ROM)** although some **may be written to as well as read**.
- Distinguishable by the number of times they can be reprogrammed (written to) and by the mechanism for reprogramming them
- Used for firmware programs (BIOS, controllers for disks, network cards, graphics accelerators, security subsystems. . .), solid state disks, disk caches

Nonvolatile Memory (II)

- Read-only memory (ROM)
  - Programmed during production
- Programmable ROM (PROM)
  - Can be programmed once
- Eraseable PROM (EPROM)
  - Can be erased and reprogrammed about 1000 times
- Electrically eraseable PROM (EEPROM)
  - Can be reprogrammed about 100,000 times
- Flash Memory
  - Based on EEPROM technology
  - Wears out after about 100,000 repeated writes

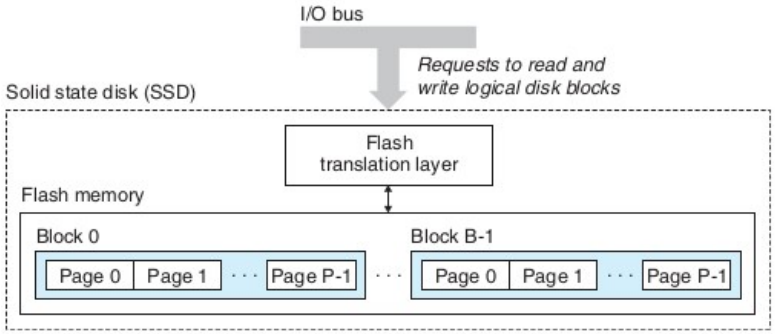
Hard Disk Drive (HDD)

- Disks hold enormous amount of data
  - On the order of **hundreds to thousands of gigabytes** compared to hundreds to thousands of megabytes in memory.
- Disks **are slower than RAM-based memory**
  - On the order of milliseconds to read information on a disk, a hundred thousand times longer than from DRAM and a million times longer than SRAM.



Solid State Disks (SSDs)

- A SSD is a storage technology, based on flash memory



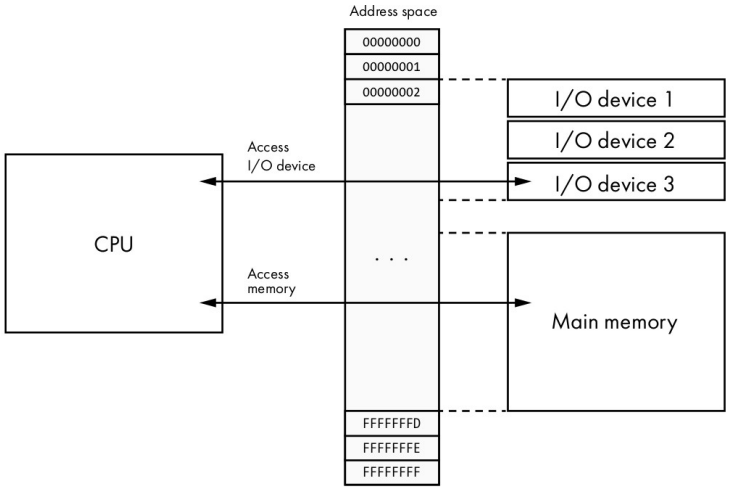
- Data read/written in units of pages.
- Page can be written only after its block has been erased
- Erasing a block is slow (around 1 ms)

I/O devices

Input/Output (I)

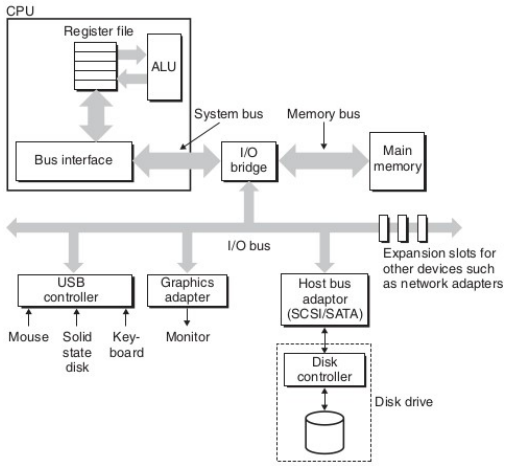
- An input/output (I/O) device is a component that allows a computer to **receive input** from the outside world (keyboard, mouse), **send data** to the outside world (monitor, printer), or both (touch- screen).
- A computer **can have a wide variety of I/O devices attached to it**.
- CPU needs a **standard way to communicate with any such device.**
  - Memory-Mapped I/O (MMIO)
    - **Addresses in Main memory address space do not always refer to bytes of memory**;
    - They can also refer **to an I/O device**.
    - When physical address space is mapped to an I/O device, the **CPU can communicate with that device just by reading or writing to its assigned memory address(es)**.

Input/Output (II)



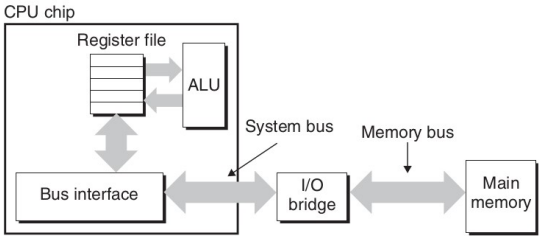
Input/Output (III)

- I/O devices such as graphics cards, monitors, mice, keyboards, and disks **are connected to the CPU and main memory using an I/O bus**.
  - I/O bus is slower than the system and memory buses



Accessing Main Memory & Disks

CPU to/from Memory



- The main components are the **CPU chip**, a chipset that we will call an **I/O bridge** (which includes the memory controller), and the **DRAM memory modules that make up main memory**.
- These components are connected by a pair of buses: a **system bus** that connects the CPU to the I/O bridge, and a **memory bus** that connects the I/O bridge to the main memory.
  - A bus is a collection of parallel wires that carry address, data, and control signals.
  - Buses are typically shared by multiple devices.

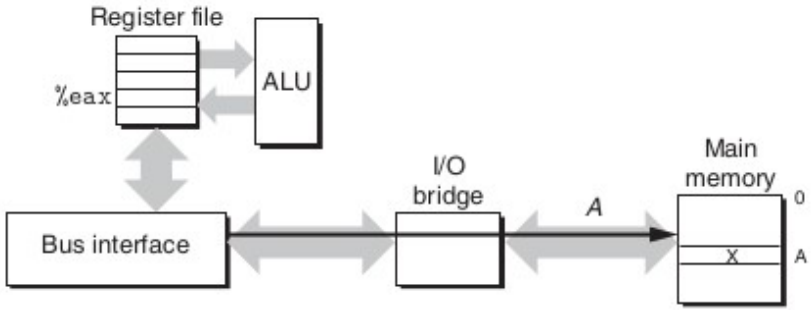
Memory Read Transaction

```
A : .int ...
...
movl A(%rip), %eax
```

- Circuitry on the CPU chip called the **Bus interface** initiates a read transaction on the bus.
- The read transaction consists of **three steps.**
  1. The **CPU places the address A on the system bus**.
     - The I/O bridge passes the signal along to the memory bus.
  2. The **main memory senses the address signal on the memory bus, reads the address from the memory bus, fetches the data word** from the DRAM, and **writes the data to the memory bus**.
     - The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus
  3. The **CPU senses the data on the system bus, reads it from the bus, and copies it to register `%eax`**

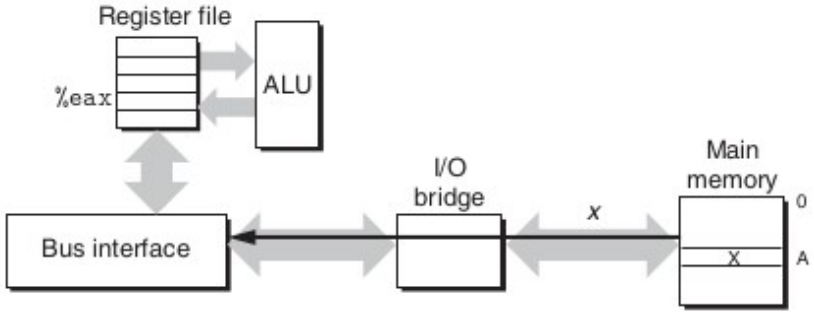
Memory Read Transaction: Step I

- CPU places address A on the memory bus.



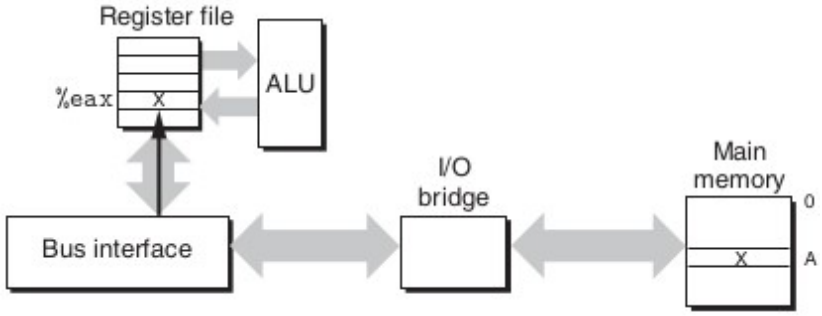
Memory Read Transaction: Step II

- Main memory reads A from the bus, retrieves word x, and places it on the bus.



Memory Read Transaction: Step III

- CPU reads word x from the bus, and copies it into register `%eax`



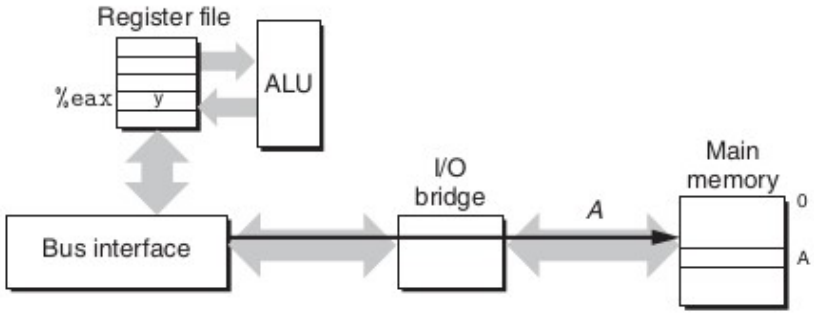
Memory Write Transaction

```
A : .int ...
...
movl %eax, A(%rip)
```

- Circuitry on the CPU chip called the **Bus interface** initiates a write transaction on the bus.
- The write transaction consists of **three steps**.
  1. The **CPU places the address A on the system bus**
     - The **memory reads the address from the memory bus and waits for the data to arrive.**
  2. The **CPU copies the data word in `%eax` to the system bus**.
  3. The **main memory reads the data word from the memory bus and stores the bits in the DRAM**.

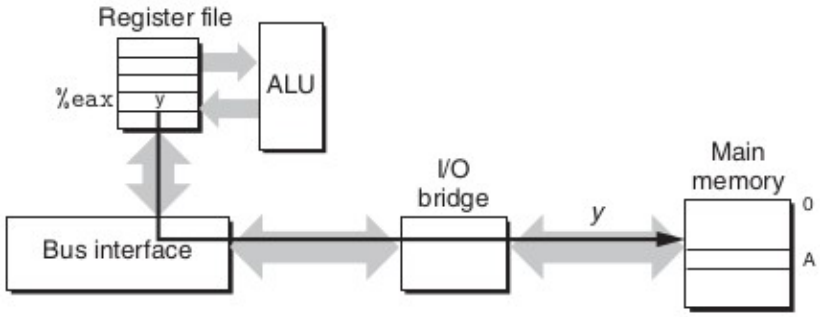
Memory Write Transaction: Step I

- CPU places address A on the memory bus. Main memory reads it and waits for the data word.



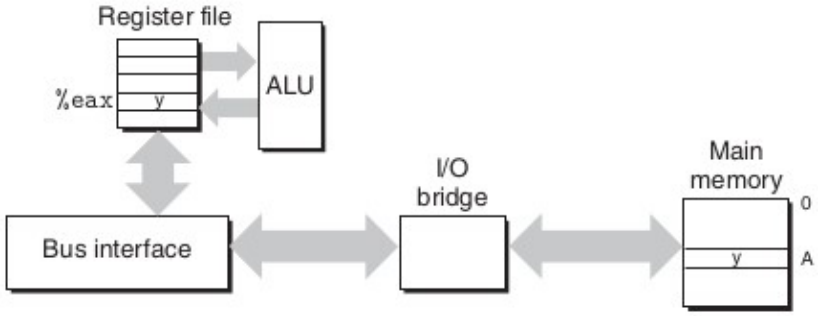
Memory Write Transaction: Step II

- CPU places data word y on the bus..



Memory Write Transaction: Step III

- Main memory reads data word y from the bus and stores it at address A

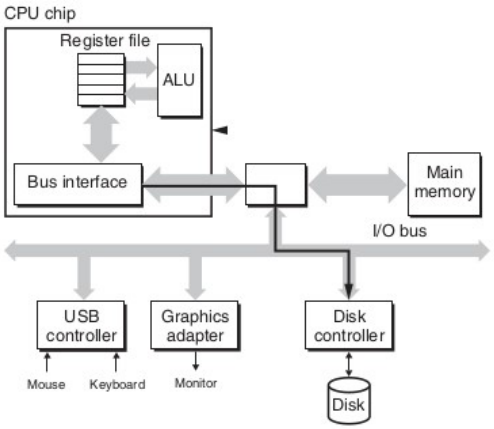


CPU to/from Disk

- The CPU **issues commands to I/O devices using a technique called memory-mapped I/O**.
  - In a system with memory-mapped I/O, a block of addresses in the address space is reserved for communicating with I/O devices.
  - Each of these addresses is known as an **I/O port**.
    - Each device is associated with (or mapped to) one or more ports when it is attached to the bus.
- Then the **CPU might initiate a disk read by executing three store instructions** to address I/O port:
  1. The first of these instructions sends a command word that tells the disk to initiate a read, along with other parameters such as whether to interrupt the CPU when the read is finished.
  2. The second instruction indicates the logical block number that should be read.
  3. The third instruction indicates the main memory address where the contents of the disk sector should be stored.

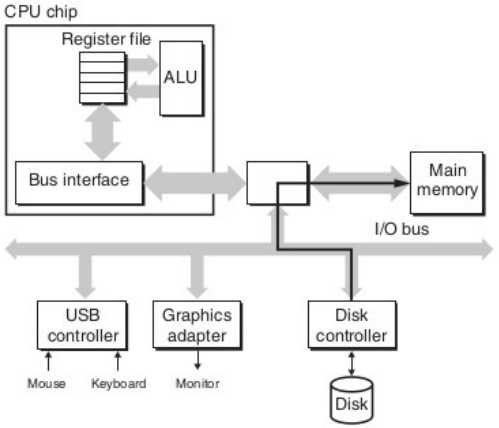
CPU reads from Disk: Step I

- CPU initiates a disk read by writing a command, logical block number, and destination memory address to a port (address) associated with disk controller.



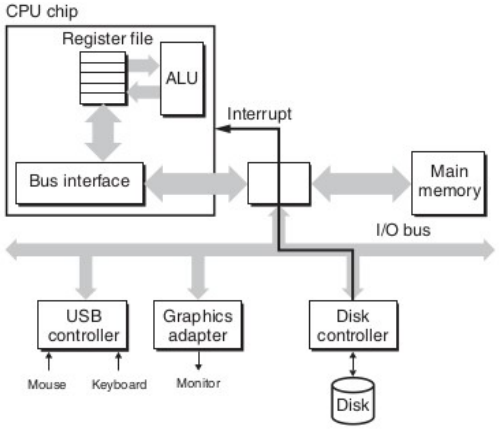
CPU reads from Disk: Step II

- Disk controller reads the sector and performs a **Direct Memory Access** (DMA) transfer into main memory.



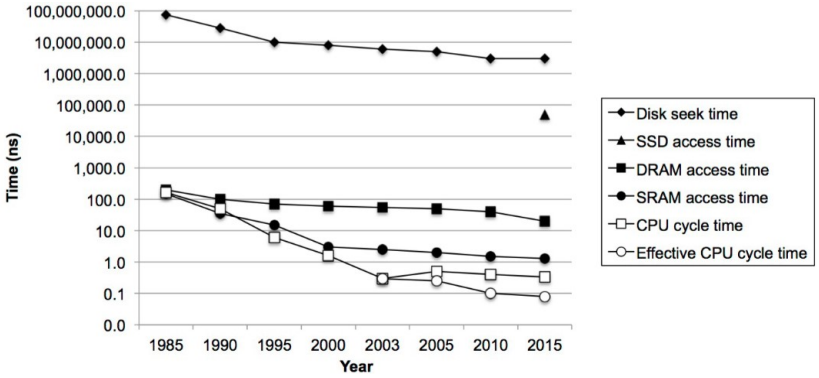
CPU reads from Disk: Step III

- When the DMA transfer completes, the disk controller notifies the CPU with an interrupt (i.e., asserts a special “interrupt” pin on the CPU)



Technology comparison

- The gap widens between DRAM, disk, and CPU speeds.

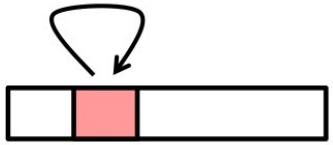


  - Effective CPU cycle time: It is calculated to be the cycle time of an individual CPU divided by the number of its processor cores

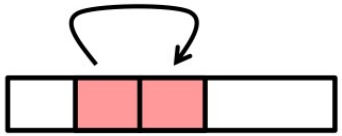
Locality

Locality

- **Temporal locality**
  - Recently referenced items are likely to be referenced again in the near future
  - **Repetition is common**



- **Spatial locality**
  - Items with nearby addresses tend to be referenced close together in time
  - **Sequential access is common**



Principle of Locality

- **Principle of Locality**
  - Programs tend to use data and instructions with addresses near or equal to those they have used recently.
- Principle of locality **has an enormous impact on the design and performance of hardware and software systems.**
- In general, programs with **good locality run faster than programs with poor locality.**
- This principle **is used by all levels of a modern computer system**: hardware, OSes, and application programs.

Stride Access Patterns

- **Sequential Access**
  - Accessing memory in sequential order (contiguous addresses)
- **Stride-k Access**
  - Accessing every $k - th$ memory address.
    - Note that stride-1 access patterns is the same as sequential access
- Stride-1 reference patterns are a common and important source of spatial locality in programs.
- In general, **as the stride increases, the spatial locality decreases**.

Instruction locality

- Normal execution exhibits **spatial locality**
  - Instructions execute in sequence
  - Long jumps exhibit poor locality (this includes calls)
- Loops exhibit both **temporal and spatial locality**
  - The body statements execute repeatedly (**temporal locality**) and in sequence (**spatial locality**)
  - Short loops are better

Data locality

- **Temporal locality**
  - keep often-used values in higher tiers of the memory hierarchy
- **Spatial locality**
  - Use predictable access patterns
  - Stride-1 reference pattern (sequential access)
  - Stride-k reference pattern (every k elements)
  - Closely related to row-major vs. column-major
  - Allows for prefetching (predicting the next needed element and preloading it)

Locality of References to Program Data (I)

- Does this function **have good locality**?

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

- To answer this question, we look at the reference pattern for each variable.
  - The `sum` variable is referenced once in each loop iteration, and thus there is **good temporal locality**.
  - Since `sum` is a scalar, there is **no spatial locality**.
  - The elements of vector `v` are read sequentially, one after the other, in the order they are stored in memory, thus, with respect to variable `v`, the function has **good spatial locality** but **poor temporal locality** since each vector element is accessed exactly once.

Locality of References to Program Data (II)

- Does this function **have good locality**?

```
1  int sumvec(int v[N])
2  {
3      int i, sum = 0;
4
5      for (i = 0; i < N; i++)
6          sum += v[i];
7      return sum;
8  }
```

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

- To answer this question, we look at the reference pattern for each variable **(cont.)**.
  - Since the function has either good spatial or temporal locality with respect to each variable in the loop body, we can conclude that the `sumvec` function **enjoys good locality**.
  - A function such as `sumvec` that visits each element of a vector sequentially is said to have a **stride-1 reference pattern** (with respect to the element size).

Locality of References to Program Data (III)

```
1  int sumarrayrows(int a[M][N])
2  {
3      int i, j, sum = 0;
4
5      for (i = 0; i < M; i++)
6          for (j = 0; j < N; j++)
7              sum += a[i][j];
8      return sum;
9  }
```

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

- The doubly nested loop reads the elements of the array in **row-major order**.
  - That is, the inner loop reads the elements of the first row, then the second row, and so on.
- The `sumarrayrows` function enjoys **good spatial locality** because it references the array in the same **row-major order** that the array is stored.
  - The result is a nice **stride-1 reference pattern with excellent spatial locality**.

Locality of References to Program Data (IV)

```
1  int sumarraycols(int a[M][N])
2  {
3      int i, j, sum = 0;
4
5      for (j = 0; j < N; j++)
6          for (i = 0; i < M; i++)
7              sum += a[i][j];
8      return sum;
9  }
```

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

- The `sumarraycols` function suffers from **poor spatial locality** because it scans the array column-wise instead of row-wise.
  - The result is a **stride-N reference pattern**.

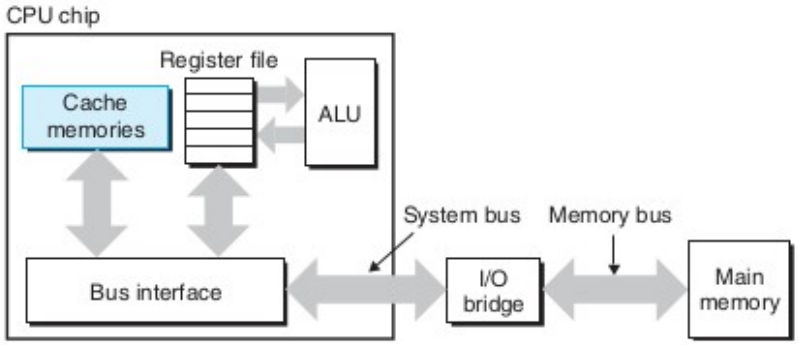
Summary Locality

- Programs that repeatedly reference the same variables enjoy **good temporal locality**.
- For programs with stride-k reference patterns, **the smaller the stride the better the spatial locality**.
  - Programs with stride-1 reference patterns have good spatial locality.
  - Programs that hop around memory with large strides have poor spatial locality.
- **Loops have good temporal and spatial locality** with respect to instruction fetches.
  - **The smaller the loop body and the greater the number of loop iterations, the better the locality**.

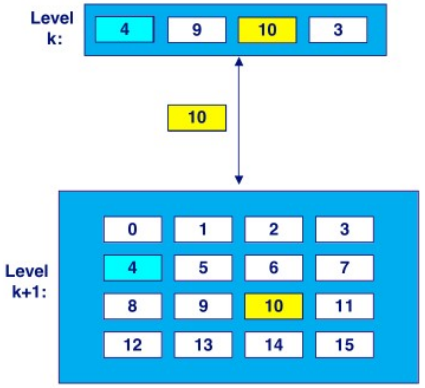
Cache Memories

Why Caches?

- Because of the increasing **gap between CPU and main memory**, system designers were compelled to insert a small SRAM cache memory between the CPU register file and main memory.

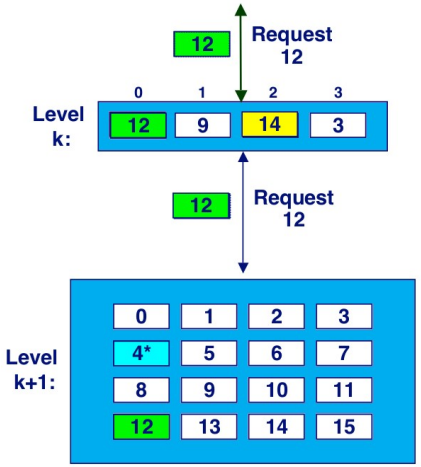


- **Cache**
  - Fast storage that stores temporary copies of data from slower storage
- Caches **allow us to store data closer to where it is actively being used**, and therefore allow retrieval of the data to be faster.
- The central idea is that **for each level k in the memory hierarchy, the faster and larger storage device serves as a cache for the larger and slower storage devices at level $k + 1$.**



Cache Hits & Miss

- Program needs object d, which is stored in some block b.
- **Cache hit**
  - Program finds b in the cache at level k. E.g., block 14.
- **Cache miss**
  - b is not at level k, so level k cache must fetch it from level $k+1$. E.g., block 12.
  - If level k **cache is full**, then some current block must be replaced (evicted).
    - **Placement policy**: where can the new block go?
    - **Replacement policy**: which block should be evicted?

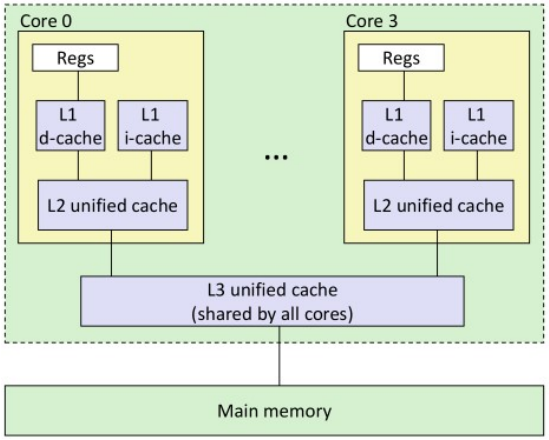


Examples of caching

Cache Type	What is Cached?	Where is it Cached?	Latency (cycles)	Managed By
Registers	4-8 bytes words	CPU core	0	Compiler
TLB	Address translations	On-Chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-Chip L1	4	Hardware
L2 cache	64-byte blocks	On-Chip L2	10	Hardware
Virtual Memory	4-KB pages	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Disk firmware
Network buffer cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Example: Intel Core i7 Cache Hierarchy

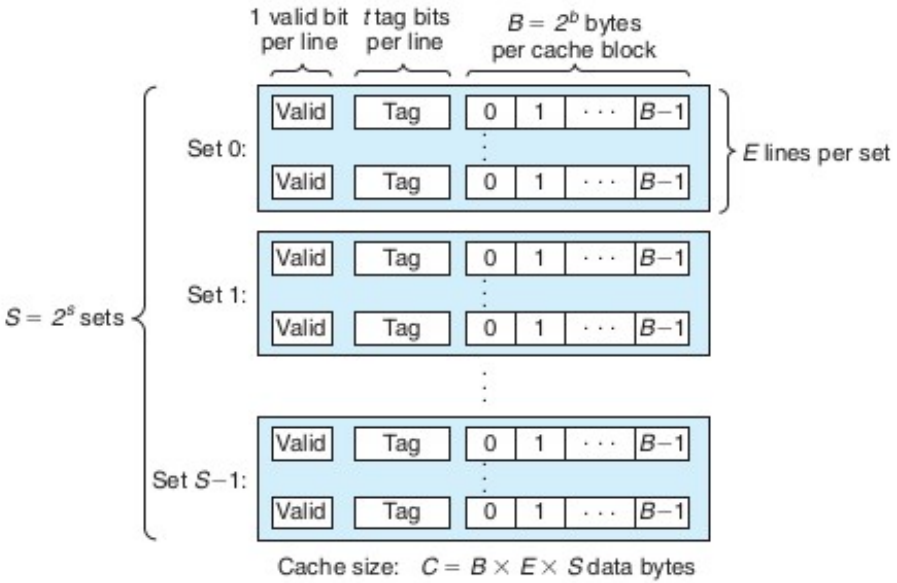
- **Block size**: 64 bytes for all caches
- **L1 i-cache and d-cache**:
  - 32 KiB
  - Access: 4 cycles
- **L2 unified cache**:
  - 256 KiB
  - Access: 11 cycles
- **L3 unified cache**:
  - 8 MiB
  - Access: 30-40 cycles



Generic cache memory organization (I)

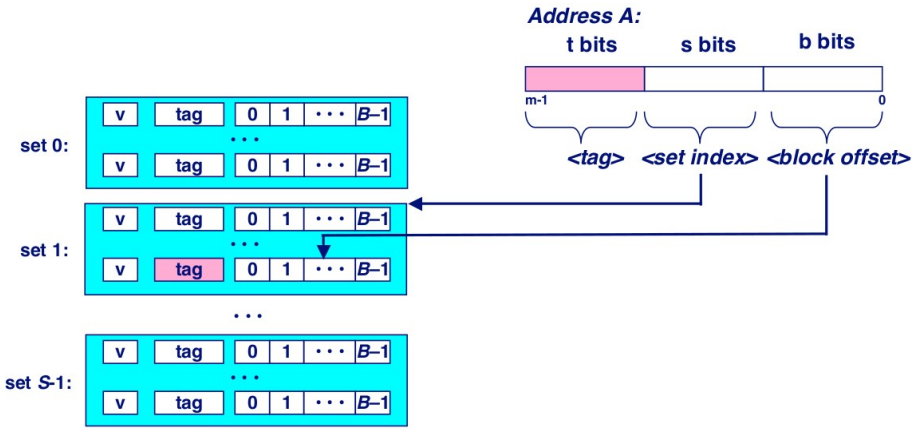
- Consider a computer system where each memory address has m bits that form $M = 2^m$ unique addresses.
- A cache for such a machine is organized as an array of $S = 2^s$ **cache sets**.
- Each set consists of E **cache lines**.
- Each line consists of a data **block** of $B = 2^b$ bytes, a valid bit that indicates whether or not the line contains meaningful information, and $t = m - (b + s)$ tag bits (a subset of the bits from the current block's memory address) that uniquely identify the block stored in the cache line.

Generic cache memory organization (II)



Addressing Caches

- The word at address `A` is in the cache if the `tag` bits in one of the `valid` lines in set `index` match `tag`.
  - The tag bits in the cache line must match the tag bits in the address.
- The word `A` contents begin at offset `block offset` bytes from the beginning of the block.



Writing Cache-Friendly Code

- Programs with **better locality tend to have lower miss rates** and programs with lower miss rates will tend to run faster than programs with higher miss rates.
- Good programmers should always try to write code that is cache friendly, in the sense that **it has good locality**.

Approach to Cache Friendly Code

- Make the **common case go fast**.
- Programs often spend most of their time in a few core functions.
- These functions often spend most of their time in a few loops.
- So **focus on the inner loops** of the core function and ignore the rest.
- Minimize the number of **cache misses for each inner loop**.