H2

VI. The Memory Hierarchy

- ***Memory system*** -- hierarchy of storage devices with different capacities, costs, and access times
  - **CPU registers** --- hold the most **frequently used** data
  - **Cache memories** --- staging areas for a subset of the data and instructions stored in the relatively slow main memory
  - **Main memory** --- data stored on large, slow disks, serve as staging areas for data stored on the **disks or tapes** or other machines connected by **networks**

- Memory hierarchies work because programs tend to access the storage at any particular level **more frequently than they access the storage at the next lower level**
  - storage at the next level can be **slower, larger, cheaper/bit**

- Memory hierarchy has a big impact on the performance

  e.g. cycles required to access if the data are stored in...

  CPU register -- 0 cycles (during the execution)

  Cache -- 4 ~ 75 cycles

  Main memory -- hundreds of cycles

  Disk -- tens of millions of cycles

- ***Locality*** -- programs with good locality tend to access the **same set of data items over and over again**, or they tend to access **sets of nearby data items**

  --> **run faster**

H3

§6.1 Storage Technologies

H5

¶6.1.1 Random Access Memory (RAM)

- **2 varieties** of **Random Access Memory** (RAM)
  1. **Static** RAM (SRAM)
     - faster, significantly more expensive
     - used for **cache memories** (both on and off the CPU chip)

2. **Dynamic** RAM (DRAM)
    - used for the **main memory** & **frame buffer** (of graphics system)

H6 Static RAM

- stores each bit in a ***bistable*** memory cell
    - each cell is implemented with a six-transistor circuit
    - can stay **indefinitely** in either of **two different voltage configurations** (**states**)
        - any other state will be unstable
            - pendulum could also remain balanced in a vertical position indefinitely (**metastable**)
                - smallest disturbance would make it start to fall, and once it fell it would never return to the vertical position
    - **SRAM** memory cell will **retain its value indefinitely**, as long as it is kept powered
        - even with disturbance (electrical noise), the circuit will return to the stable value when the disturbance is removed

H6 Dynamic RAM

- Stores each bit as **charge on a capacitor**
    - can be made **very dense** -- each cell consists of a capacitor and a single access transistor
- **Very sensitive** to any disturbance
    - when the capacitor voltage is disturbed, it **will never recover**
- **lose its charge within a time period of around 10~100 ms**
    - clock cycle times measured in ns --> **retention time is quite long**

H6 Conventional DRAMs

- Cells (bits) in a DRAM chips are partitioned into d **supercells**, each consisting of w **DRAM cells**
    - $d \times w$ DRAM stores a total of dw bits of information
    - Supercells are organized as a rectangular array with r rows and c columns, where $rc = d$
    - Each supercell has an address of the form (i, j)

- Information **flows in and out** of the chip via external connectors called ***pins***

- each pin carries a **1-bit signal**
- 8x data pins can transfer 1 byte in or out of the chip
- 2x addr pins can carry **two-bit row and column** supercell **addresses**

- Each DRAM chip is connected to ***memory controller***
  - can transfer w bits at a time to and from each DRAM chip
  - to read the contents of supercell (i, j),
    1. Memory controller sends the row address i to the DRAM (**RAS request** (row access strobe))
    2. DRAM responds by copying the **entire contents of i-th row into an internal row buffer**
    3. Memory controller sends the column address j to the DRAM (**CAS request** (column access strobe))
    4. DRAM responds by copying the **8-bits in supercell (i,j) from the row buffer** and sending them to the memory controller

- Reason for DRAMs to be **two-dimensional arrays** instead of linear arrays
  - **reduce the number of address pins on the chip**
  - **However** , since with two-dimensional array organization, addresses must be sent in **two distinct steps** --> increases the access time

H6 Memory Modules

- DRAM chips are packaged in ***memory modules*** that plug into expansion slots on the motherboard
  - Core i7 systems use the **240-pin *dual inline memory module (DIMM)*** , transferring data to and from the **memory controller** in 64-bit chunks

- To retrieve the word at memory address A
  1. the **memory controller** **converts** A to a **supercell address** (i, j) and **sends it to the memory module**
  2. **memory module** **braodcasts** i and j to each **DRAM**
  3. **DRAM** **outputs** the **8-bit contents** of its (i, j) supercell
  4. **memory module** forms them into a 64-bit word and **returns** to the **memory controller**

H6 Enhanced DRAMs

- based on the conventional DRAM cell, with optimizations that improve the speed with which the basic DRAM cells can be accessed

1. Fast page mode DRAM (FPM DRAM)
    - allows consecutive accesses to the same row to be served directly from the row buffer
        - memory controller sends an iniiial RAS/CAS request, followed by three CAS requests
            - initial RAS/CAS request copies specified row into the row buffer and returns the supercell addressed by the CAS, then the next three supercells are served directly from the row buffer

2. Extended data out DRAM (EDO DRAM)
    - enhanced form of FPM DRAM that allows the individual CAS signals to be spaced closer together in time

3. Synchronous DRAM (SDRAM)
    - replaces many of control signals with the rising edges of the same external clock signal that drives the memory controlelr --> can output the contents of its supercells at a faster rate than its asynchronous counterparts (Conventional, FPM, EDO)

4. Double Data-Rate Synchronous DRAM (DDR SDRAM
    - enhancement of SDRA that doubles the speed of the DRAM by using both clock edges as control signals
    - characterized by the size of a small prefetch buffer that increases the effective bandwidth
        - DDR (2bits), DDR2 (4bits), DDR3 (8bits)

5. Video RAM (VRAM)
    - used in the frame buffers of graphics system
    - similar to FPM DRAM but...
        1. VRAM output is produced by shifting the entire contents of the internal buffer in sequence
        2. VRAM allows **concurrent reads and writes** to the memory

H6 Nonvolatile Memory

- DRAMs and SRAMs are ***volatile*** -- lose their information if the supply voltage is turned off

- ***Nonvolatile memories*** -- retain their information even when they are powered off
- For historical reason, nonvolatile memories are called **read-only memories (ROMS)**
  - even though some can be written to

- ROMS are distinguished by the **number of times** they can be reprogrammed (written to) and by the mechanism for reprogramming them
  1. Programmable ROM (PROM)
     - can be programmed exactly once
       - memory cell can be blown once by zapping a high current

  2. Erasable programmable ROM (EPROM)
     - has a transparent quartz window that permits light to reach the storage cells
       - cleared by shining ultraviolet light
     - requires a special device to write to
     - can be erased and reprogrammed on ≈1,000 times

  3. Electrically erasable PROM (EEPROM)
     - does not require a physically separate programming device
     - can be reprogrammed on the order of 10^5 times before it wears out
     - ***Flash memory*** is based on EEPROMS
       - provides fast, durable nonvolatile storage
       - **SSD** (solid state disk)

H6 Accessing Main Memory

- data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called ***buses***
  - **bus transaction** -- steps used for transfer of data
    - ***read transaction*** -- Main memory -> CPU
    - ***write transaction*** -- CPU -> Main memory

- ***bus*** -- collection of parallel wires that carry address, data, and control signals
- control wires -- carry signals that synchronize the transaction and identify what kind of transaction is currently being performed

CPU chip ------- I/O bridge ------- Main memory

[System Bus] [Memory Bus]

Suppose CPU performs a **load operation**

$$movq \ \ A, \%rax$$

--> ***bus interface*** initiates a **read transaction** on the bus by...

1. CPU places the address A on the system bus --> I/O bridge passes the signal along to the memory bus
2. Main memory senses the address signal on the memory bus, reads the addrss from the memory bus, fetches the data from the DRAM, and writes the data to the memory bus --> I/O bridge translates the memory bus signal into a system bus signal and passes it along to the system bus
3. CPU senses the data on the system bus, reads the data from the bus, and copies the data to register **%rax**

Conversely, suppose CPU performs a **store operation**

$$movq \ \ \%rax, A$$

--> ***bus interface*** initiates a **write transaction**

1. CPU places the address on the system bus --> memory reads the address from the memory bus and waits for the data to arrive
2. CPU copies the data in **%rax** to the system bus
3. Main memory reads the data from the memory bus and stores the bits in the DRAM

H5 ¶6.1.2 Disk storage

- ***Disks*** -- hold enormous amounts of data (GB) but takes milliseconds to read information
  - ≈100,000 longer than from DRAM, ≈1,000,000 longer than from SRAM

H6 Disk Geometry

- **Disks** (rotating disks) are constructed from ***platters***

- Each platter consists of **two sides (*surfaces*)** coated with magnetic recording material
    - each **surface** consists of a collection of **concentric rings (*tracks*)**
        - each **track** is partitioned into a collection of ***sectors***
            - each **sector** contains an **equal number of data bits** (512 bytes) encoded in the magnetic material on the sector
                - sectors are separated by ***gaps*** (with no data bits stored)
- Rotating ***spindle*** in the center of the platter **spins** the platter at a fixed rotational rate (5,400 ~ 15,000 RPM)
- ***Cylinders*** -- collection of **tracks on all surfaces** that are **equidistant** from the center of the spindle

H6 Disk Capacity

- Maximum number of bits that can be recorded by a disk is determined by...
    1. **Recording density** (bits/in)
    2. **Track density** (tracks/in)
    3. **Areal density** (bits/ in^2)

- Modern high-capacity disks use ***multiple zone recording***
    - set of cylinders is partitioned into disjoint subsets known as ***recording zones***
        - each zone consists of a contiguous collection of cylinders
            - each track in each cylinder in a zone has the same number of sectors, determined by the number of sectors that can be packed into the innermost track of the zone

$$Capacity = \frac{\#bytes}{sector} \times \frac{average\ \#sectors}{track} \times \frac{\#tracks}{surface} \times \frac{\#surfaces}{platter} \times \frac{\#platters}{disk}$$

H6 Disk Operation

- Disks read and write bits stored on the magnetic surface using a ***read/write head*** connected to the end of an **actuator arm**
    - by moving back and forth along its radial axis, the drive can **position the head over any track on the surface** --> **seek**
    - once the head is **positioned over the desired track** , each bit on the track passes underneath, the head can either **sense the value of the bit (read)** or **alter the value of the bit (write)**

- disks with multiple platters have a separate read/write head for each surface, lined up vertically and move in unison
- head flies on about 0.1 microns over the disk surface with a speed of ≈80 km/h --> tiny piece of dust can cause a **head crash** --> disks are always sealed in airtight packages

- **Access time** has three main componenets
  1. **Seek time**
     - time required to move the arm over the track that contains the target sector
     - depends on the **previous position of the head** & **speed that the arm moves** across the surface

     $T_{avg\ seek}$ ≈ 3 to 9 ms $\quad$ T_{max} ≈ 20 ms

  2. **Rotational latency**
     - once the head is in position over the track, the drive waits for the **first bit of the target sector** to pass under the head
     - depends on **position of the surface** when the head arrives at the target track & rotational speed of the disk
     - **worst case??**

       --> when the head just misses the target sector and waits for the disk to make a full rotatoin

       $$T_{max\ rotation} = \frac{1}{RPM} \times \frac{60\ secs}{1\ min}$$
       $$T_{avg\ rotation} = \frac{T_{max\ rotation}}{2}$$

  3. **Transfer time**
     - when the first bit of the target sector is under the head, the driver can **begin to read or write the contents of the sector**
     - depends on the **rotational speed** & **number of sectors per track**

       $$T_{avg\ transfer} = \frac{1}{RPM} \times \frac{1}{(average\ \#\ sectors/track)} \times \frac{60\ secs}{1\ min}$$

- Time to access a disk sector is dominated by the **seek time** and the **rotational latency** ---> **twice the seek time** is good for estimating disk access timee

H6 Logical Disk Blocks

- To hide the complexity from the OS, modern disks present a simpler view of their geometry as a sequence of B **sector-size logical blocks**
  - ***disk controller*** in the disk package maintains the **mappling between logical block numbers and actual (physical) disk sectors**

- When the **OS** wants to perfrom an I/O operation (reading)
  1. **OS** sends a command to the **disk controller** asking it to read a **particular logical block number**
  - **Firmware on the controller** performs a fast table lookup that **translates the logical block number into a *(surface, track, sector)* triple** that uniquely identifies the corresponding **physical sector**
  - **Hardware on the controller** interprets this triple to **move the heads** to the appropriate cylinder, waits for the sector to pass under the head, gathers up the bits sensed by the head into a small memory buffer on the controller, and copies them into main memory

H6 Connecting I/O Devices

- **I/O devices** (graphic cards, monitors, mice, keyboards, disks..) are connected to the CPU and main memory using an ***I/O bus***
  - independent of the underlying CPU (system bus and memory bus are CPU-specific)
  - slower than the system and memory buses, but can accomodate a wide variety of third-party I/O devices

1. **Universal Serial Bus (USB)**
   - popular standard for connecting a variety of I/O devices (keyboards, mice, modems, digital cameras, game controllers, printers, external disk drivers, SSDs)
2. **Graphic card (adapter)**
   - contains hw/sw logic that is reponsible for painting the pixels on the display monitor
3. **Host bus adpater**
   - connects one or more disks to the I/O bus using ***host bus interface***
     - ***SCSI*** , ***SATA***
4. Additional devices
   - can be attached to the I/O bus by plugging the adapter into empty **expansion slots** on the motherboard

H6 Accessing Disks

- CPU issues commands to I/O devices using a technique called ***memory-mapped I/O***
  - a block of addresses is reserved for communicating with I/O devices
    - each of these addresses is known as an **I/O port**
    - each device is associated with one or more ports

- Suppose that the disk controller is mapped to port 0xa0 and **CPU** is initiating a disk read (**three store instructions** to address 0xa0)
  1. 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. **logical block number** that should be read
  3. **main memory address** where the contents of the disk sector should be stored

  --> After it issues the request, the CPU will typically **do other work while the disk is performing the read**

- After the **disk controller** receives the read command from the **CPU**
  1. translates the **logical block number** to a sector address
  2. reads the contents of the sector
  3. transfers the contents **directly** to main memory, **without any intervention from the CPU** ***direct memory access (DMA)*** ---> **DMA transfer**

- After the **DMA transfer** is complete and the contents of the disk sector are safely stored in main memory, the **disk controller** notifies the CPU by sending an **interrupt signal to CPU**

  --> causes the CPU to stop what it is currently working on and jump to an operating system routine

H5 ¶6.1.3 Solid State Disks

- Based on **flash memory**, attractive alternative to the conventional rotating disk
- **SSD package** plugs into a standard disk slot on the **I/O bus** (USB or SATA) and behaves like any other disk, processing requests from the CPU to read and write **logical disk blocks**

- consists of one or more **flash memory chips** (replacing mechanical drive in a conventional rotating disk)
- **flash translation layer** (≈ disk controller), that translates requests for logical blocks into accesses of the underlying physical device

- Reading from SSDs is faster than writing
  1. erasing a block takes a relatively long time
  2. if a write operation attempts to modify a page that contains existing data, then any pages in the same block with useful data must be copied to a new (erased) block before the write to page ocan occur
     - page can be written only after the entire block to which it belongs has been erased (all bits set to 1)

- **Advantages** over rotating disks
  1. no moving parts --> faster random access times
  2. use less power
  3. more durable

- **Disadvantages**
  1. potential to wear out (∵ flash blocks wear out after repeated writes)

     --> **Wear-leveling logic** used to maximize the lifetime of each block by spreading erasures evenly across all blocks
  2. ≈30x more expensive per byte than rotating disks

     --> storage capacities are significantly less than rotating disks

H5 ¶6.1.4 Storage Technology Trends

1. Different storage technologies have different price and performance trade-offs
   - fast storage is always more expensive than slower storage
   - **SSDs** is somewhere in between DRAM and rotating disk

2. The price and performance properties of different storage technologies are changing at dramatically different rates
   - e.g. since 1985...

    - cost & performance of **SRAM** tech have improved at roughly the same rate (116x & 115x)
    - for **DRAM** cost & performance improvement (44,000x & 10x)
    - for **Rotating disk** (3,333,333x & 25x)
  - it is much easier to **increase density** (reduce cost) than to decrease access time

3. DRAM and disk performance are lagging behind CPU performance
   - gap between DRAM and disk performance and CPU performance is widening
     - until the advent of multi-core processors, performance gap was a function of latency
     - after the multi-core processors, performance gap becomes a function of throughput (multiple cores issuing requests to the DRAM and disk in parallel)
   - modern computers make heavy use of **SRAM-based** caches to try to bridge the processor-memory gap (**locality**)

H3 §6.2 Locality

- ***Principle of locality*** -- tend to reference data items that are near other recently referenced data items or that were recently referenced themselves
  - enormous impact on the design and performance

1. **Temporal locality**
   - memory location that is referenced once is likely to be **referenced** again **multiple times** in the near future

2. **Spatial locality**
   - if a memory location is referenced once, then the program is likely to **reference a nearby location** in the nearby future

--> programs with good locality run faster than programs with poor locality

- All levels of modern computer systems are designed to exploit locality
  - **Hardware level**
    - ***cache memories*** -- hold blocks of the most recently referenced instructions and data items
  - **Operating system level**

- allows the system to use the **main memory as a cache** of the most recently referenced chunks of the virtual address space

H5 ¶6.2.1 Locality of References to Program Data

[Example C code]

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

- **sum** variable is referenced once in each loop iteration

  --> **good temporal locality**
- elements of vector **v** are read sequentially (in the order they are stored in memory)

  --> **good spatial locality**

--> **sumvec** uses a **good locality**

- **sumvec** is said to have a **stride-1 reference pattern** (**= sequential reference patterns**)
  - visits each element of a vector sequentially
- Visiting every k-th element of a contiguous vector is called a **stride-k reference pattern**
  - as the stride increases, the **spatial locality decreases**

- **Doubly nested loop** reads the elements of the array in ***row-major order***
  - inner loop reads the elements of the first row, then the second row, and so on

[Example C codes]

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

```
}

int smarraycols(int a[M][N]) {
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];
  return sum;
}
```

- **sumarrayrows** function enjoys **good spatial locality**

  --> references the array in the same row-major order that the array is stored (**stride-1 reference pattern**)

- **sumarraycols** function computes the same result but has **poor spatial locality**

  --> (**stride-N reference pattern**)

H5 §6.2.2 Locality of Instruction Fetches

- Since program instructions are stored in memory and must be fetched (read) by the CPU, we can also evaluate the locality of a program with respect to its **instruction fetches**
- e.g.) **sumvec** function from ¶6.2.1
  - **for** loops are executed in sequential memory order --> **spatial locality**
  - **loop body** is executed multiple times --> **temporal locality**

H5 ¶6.2.3 Summary of Locality

1. Programs that repeatedly reference the same variables enjoy good temporal locality
2. For programs with stride-k reference patterns, the smaller the stride, the better the spatial locality.
   - stride-1 reference patterns have good spatial locality
   - programs that hop around memory with large strides have poor spatial locality
3. 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

§6.3 The Memory Hierarchy

¶6.3.1 Caching in the Memory Hierarchy

- **Cache** - small, fast storage device that acts as a **staging area** for the data objects stored in a larger, slower device
  - proces of using a cache = **caching**
- Central idea of a **memory hierarchy**
  - 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 $k+1$
  - each level in the hierarchy caches data objects from the next lower level

    e.g. local disk serves as a cache for files retrieved from remote disks over the network

- Storage at level $k+1$ is partitioned into contiguous chunks of data objects called ***blocks***
  - each block has a **unique address or name**
- Similarly, storage at level k is partitioned into a **smaller set of blocks** that are the **same size as the blocks at level** $k+1$
- Data are always copied back and forth between level k and level $k+1$ in block-size ***transfer units***
  - block size is **fixed** between any particular pair of **adjacent levels in the hierarchy**, **BUT** other pairs of levels can have **different block sizes**
    - e.g. L1 - L0 use word-size blocks, while L2 - L1 use blocks of tens of bytes

Cache Hits

- When a program needs a particular data object d from level $k+1$, it first looks for d in one of the blocks currently stored at level k
  - If d is already **cached at** level k --> ***cache hit***
    - program reads d **directly** from level k (faster)
    - program with **good temporal locality?**

Cache Misses

- If d is **not cached** at level k --> ***cache miss***
  - cache at level k **fetches the block containing** d from level $k+1$, possibly **overwriting an existing block** if the level k is already full

    --> **replacing** or **evicting**
    - block that is evicted is called a ***victim block***

        - decision made by the cache's **replacement policy**
            1. **random replacement policy**
                - choose a random victim block
            2. **least recently used (LRU) replacement policy**
                - choose the block that was last accessed the furthest in the past
- After the cache at level k has fetched the block from level $k+1$, the program can read d from level k as before
    - once it has been copied, remain in level k in expectation of later accesses

H6 Kinds of Cache Misses

1. **Compulsory miss (cold miss)**
    - if the cache at level k is **empty (cold cache), any access of any data object will miss**

      --> temporary event, might not occur in steady state, after the cahce has been *warmed up* by repeated memory access

2. **Conflict miss**
    - whenever there is a miss, the cache at level k must implement some ***placement policy*** that determines where to place the block it has retrieved from level $k+1$
        - if to allow any block from level $k+1$ to be stored in any block at level k, too expensive to locate!
            - hence, caches typically **restricts a particular block** at level $k+1$ to a **small subset** of the blocks at level k
                - **Conflict Miss!!** --> cache is large enough to hold the referenced data objects, but because they map to the same cache block, the cache keeps missing

3. **Capacity miss**
    - **Sequence of phases** -- each phase accesses some reasonably constant set of phases (**loops**) --> ***working set***
        - If the cache is too small to handle this particular working set

H6 Cache Management

- Storage device at each level is a **cache for the next lower level**
  - ∃ some **logic** that **manages** the cache
    - partitioning the cache storage into blocks
    - tranferring blocks between different levels
    - deciding when there are hits and misses
    - etc.

  e.g. **Compiler** -- Register files

  Hardware logic -- L1, L2, L3

  OS + HW -- DRAM main memory

H5 ¶6.3.2 Summary of Memory Hierarchy Concepts

- Memory hierarchies based on caching work because **slower storage is cheaper than faster storage** & **programs tend to exhibit locality**

  1. Exploiting **temporal locality**
     - same data objects are likely to be **reused multiple times**
     - once a data has been copied into the cache on the first miss, can expect a number of subsequent hits on that object
  2. Exploiting **spatial locality**
     - blocks usually contain multiple data objects
     - cost of copying a block after a miss will be amortized by subsequent references to other objects within that block

H3 §6.4 Cache Memories

- Early computer systems consisted of **only three levels of memory hierarchies** (CPU registers, main memory, disk storage)
- However, due to the **increasing gap between CPU and main memory,** small SRAM ***cache memory (L1 cache)*** was inserted between the CPU register file and main memory
  - L1 can be accessed nearly as fast as the registers
- As the gap between CPU and main memory continued to increase, additional larger caches, **L2, L3** were added between the L1 cache and main memory
- CSAPP uses an assumption about a simple memory hierarchy with a single L1 cache between the CPU and main memory

H5 ¶6.4.1 Generic Cache Memory Organization

Consider a computer system where each memory address has m bits that form $M = 2^m$ unique addresses

A cache is organized as an array of $S = 2^s$ **cache sets**

Each cache 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** that uniquely identify the block stored in the cache line

- Cache's organization can be characterized by the tuple **(S, E, B, m)**
  - size of a cache is stated in terms of the aggregate size of all the blocks (excluding tag bits and valid bit)

    $$C = S \times E \times B$$

- When the CPU is instructed by a **load instruction to read a word from address A** of main memory, it **sends address A to the cache**
  - if the cache is holding a copy of the word at address A, it sends the word immediately back to the CPU
  - but... how does the cache know whether it contains a copy of the word at address A?
    - Cache is **organized** --> can find the requested word by simply inspecting the bits of the address (≈ hash table)

      s **set index bits** in **address** A form an index into the array of S sets

      --> tells which set the word must be stored in

      t **tag bits** in **address** A

      --> tells which line (if any) in the set contains the word

      --> line in the set **contains the word iff** the **valid bit is set** and the **tag bits** in the line match the **tag bit in the address A**

      b **block offset bits**

      --> tells offset of the word in the B-byte data block

H5 ¶6.4.2 Direct-Mapped Caches

- Caches are grouped into different classes based on E, the **number of cache lines per**

set

- A cache with exactly **one line per set ($E = 1$)** is known as ***direct-mapped cache***

Suppose we have a system with a **CPU, Register file, L1 cache,** and **main memory**

- CPU executes an instruction that **reads a memory word** w, it requests the word from the **L1 cache**
  - If the **L1 cache has a cached copy of** w, --> **cache hit**
    - cache quickly extracts w and returns it to the CPU
  - If **cache miss,**
    - **CPU** **must wait** while the **L1 cache** **requests** a copy of the block containing w from the main memory
    - when the requested block arrives from **memory** the **L1 cache** **stores** the block in one of its cahce lines
    - **extracts** word w from the stored block
    - **returns** it to the CPU

- Process that a cache goes thorugh of determining whether a request is a hit or a miss and then extracting the requested word consists of **three steps**
  1. **Set selection**
  2. **Line matching**
  3. **Word extraction**

H6 Set Selection in Direct-Mapped Caches

- **Cache extracts** the s **index bits** from the **middle of the address** for w
  - these bits are **interpreted** as an unsigned integer that corresponds to a **set number**
    - think of the cache as a one-dimensional array of sets, then the set index bits form an index into this array

H6 Line Matching in Direct-Mapped Caches

- After some set i has been selected, need to determine if a copy of the word w is **stored in one of the cache lines** contained in set i
- In a **direct-mapped cache**, there is only one line per set
  - hence, a copy of w is **contained** iff..
    1. **valid bit is set** (set to 1)
    2. **tag in the cache line matches the tag in the address of** w

--> If either the valid bit were not set or the tags did not match, we would have had a cache miss

H6 Word Selection in Direct-Mapped Caches

- Once we have a hit, we know that w is somewhere in the block
- Need to determine **where the desired word starts in the block**
  - **Block offset bits** provide the **offset of the first byte** in the desired word
  - think of a block as an array of bytes, and the byte offset as an index into that array

H6 Line Replacement on Misses in Direct-Mapped Caches

- If the cache **misses**, it needs to **retrieve the requested block from the next level in memory hierarchy** and **store the new block in one of the cache lines** of the set indicated by the **set index bits**
- if the set is **full of valid cache lines, one of the existing lines must be evicted**
  - for a **direct-mapped cache** , (each set has exactly one line), **the current line is replaced by newly fetched line**

H6 Conflict Misses in Direct-Mapped Caches

- Conflict misses in direct-mapped caches typically occur when programs **access arrays whose sizes are a power of 2**

[Example C code]

```
1  float dotprod(float x[8], float y[8]) {
2    float sum = 0.0;
3    int i;
4
5    for (i = 0; i < 8; i++)
6      sum += x[i] * y[i];
7    return sum;
8  }
```

- since the function has good spatial locality with respect to x and y, we might expect it to achieve a good number of cache hits
- **HOWEVER...**

  Suppose that floats are 4 bytes, x is loaded into the 32 bytes of contiguous memory starting at address 0, and y starts immidiately after x at address 32

Suppose that a block is 16 bytes ($B = 16$, holding 4 floats), and the cache consists of 2 sets ($S = 2$) --> $C = 32$ bytes

----> will cause a **conflict miss**: each subsequent reference to x and y causess thrashing back and forth between blocks of x and y

--> **Thrashing!**: **cache is repeatedly loading and evicting the same sets of cache blocks**

- To fix **Thrashing,** put B bytes of **padding at the end of each aary**

  for example above, instead of defining x to be $float\ x[8]$, set it to be $float\ x[12]$ --> x[i] and y[i] map to different sets

H5 ¶6.4.3 Set Associative Caches

- **Main reason** for **conflict misses** in **direct-mapped caches** = each set has **exactly one line**
- **Set associative cache** -- each set holds **more than one cache line**

  A cache with $1 < E < C/B$ --> is called an E**-way set associative cache**

H6 Set Selection in Set Associative Caches

- Identical to a direct-mapped cache

H6 Line Matching and Word Selection in Set Associative Caches

- MUST **check** the **tags** and **valid bits** of multiple lines in order to determine if the requested word is in the set
- Think of each set in a **set associative cache** as a **small associative memory** where **keys = tag + valid bits** & **values = contents of a block**
  - c.f. ***associative memory*** is an array of (key, value) pairs that takes key as an input and returns a value from one of the (key, value) pairs that matches the input key
- Any line in the set can contain any of the memory blocks that map to the set --> **Cache must search each line in the set for a valid line whose tag matches the tag in the address**

H6 Line Replacement on Misses in Set Associative Caches

- Cache miss! --> **cache must fetch the block** that contains the word from memory

- Once the cache has **retrieved the block,** which line should it replace?
  1. If $\exists$ an empty line --> fill the empty line with a newly fetched blocks
  2. If $\nexists$ empty lines in the set --> must choose one to be replaced
     - **Cache replacement policies**
       1. replace at random
       2. **Least frequently used (LFU)**
       3. **Least recently used (LRU)**

          --> these "more sophisticated" policies require additional time and hardware, but as we move further down the memory hierarchy (away from CPU), the cost of a miss becomes more expensive, making it worthwhile to minimize misses with good replacement policies

H5 §6.4.4 Fully Associative Caches

- Consists of a **single set that contains all of the cache lines** ($E = C/B$)
  - since it has only one cache set, there are **no set index bits**

H6 Set Selection in Fully Associative Caches

- **No set index bits** ($\because$ there is only one set)

  --> address is partitioned into only a **tag & block offset**

H6 Line Matching and Word Selection in Fully Associative Caches

- Work the same as with a set associatvie cache
- Since the cache circuitry **must search for many matching tags in parallel**, it is **difficult & expensive** to build an associative cache that is both large and fast

  --> Fully associative caches are only appropriate for small caches

H5 ¶6.4.5 Issues with Writes

- Suppose we write a word w that is **already cached** (**write hit**)
  1. **Write-through**
     - immediately write w's cache block to the next lower level
     - Disadvantage!

        - causing bus traffic with every write
    - **No-write-allocate**
        - bypasses the cache and writes the word directly to the next lower level
2. **Write-back**
    - defers the update as long as possible by writing the updated block to the next lower level **only when it is evicted from the cache by the replacement algorithm**
    - Disadvantage!
        - additional complexity --> **must maintiain an additional dirty bit** for each cache line that indicates whether or not the cache block has been modified
    - **Write-allocate**
        - loads the corresponding from the next lower level into the cache and then updates the cache block
            - tries to exploit spatial locality
            - Disadvantage!
                - every miss results in a block transfer from the next lower level to the cache

- Assume **write-back, write-allocate caches** when trying to write reasonably cache-friendly programs
    - **caches at lower levels of the memory hierarchy** are more likely to use **write-back** instead of write-through **because of the larger transfer times**
    - **symmetric to the way reads are handled** (exploiting locality)

H5 ¶6.4.6 Anatomy of a Real Cache Hierarchy

- ***i-cache*** - cache that holds **instructions only**
- ***d-cache*** - cache that holds **program data only**
- ***unified cache*** - cache that holds **both instructions & data**

- Modern processors include **separate i-cache & d-cache**
    - with two separate caches, the processor **can read** an instruction word and a data word **at the same time**
    - **i-caches** are typically **read-only** --> simpler
    - two caches are often optimized to different access patterns and can have different block sizes, associativities, and capapcities

- ensures that **data accesses do not create conflict misses with instruction accesses (vice versa)** , at a cost of a potential increase in capacity misses

- **Core i7 Processors**
  - each CPU chip has 4 cores
    - each core has its own **private L1 i-cache, L1 d-cache**
      - the two L1 caches share **L2 unified cache**
    - all of the cores share an **on-chip L3 unified cache**

H5 ¶6.4.7 Performance Impact of Cache Parameters

- Cache performance is evaluated with...
  - **Miss rate**
    - #misses / #references
  - **Hit rate**
    - 1 - miss rate
  - **Hit time**
    - the time to deliver a word in the cache to the CPU
      - set selection + line identification + word selection
  - **Miss penalty**
    - any additional time required because of a miss

- Some of the qualitative trade-offs are...

H6 Impact of Cache Size

- Larger cache will tend to **increase the hit rate**, but it is **always harder** to make large memories run faster --> **increase in hit time**
  - Reason why L1 cache is smaller than L2, L2 is smaller than L3 ...

H6 Impact of Block Size

- Larger blocks can help **increase the hit rate by exploiting any spatial locality**

- However, larger blocks imply a **smaller number of cache lines** --> **hurt the hit rate** in programs with **more temporal locality** than spatial locality
    - Also, larger blocks have a negative impact on the **miss penalty** (∵ larger transfer times)

H6 Impact of Associativity

- **Associativity** -- number of **cache lines per set** (E)
- **Higher associativity decreases** the vulnerability of the cache to **thrashing** due to **conflict misses**
- **However, higher associativity...**
    - is expensive to implement and hard to make fast
    - requires **more tag bits per line** , **additional LRU (Least Recently Used)** state bits per lin, and **additional control logic**

    --> can **increase hit time** (∵ increased complexity)

    --> can **increase miss penalty** (∵ increased complexity of choosing a victim line)

H6 Impact of Write Strategy

- **Write-through caches** are **simpler to implement** and can use a ***write- buffer*** that works independently of the cache to update memory
    - read misses are less expensive because they do not trigger a memory write
- **Write-back caches** result in **fewer transfers**, allowing more bandwidth to memory for I/O devices that perform **DMA (Direct Memory Access)**
    - **reducing the number of transfers** becomes increasingly **important** as we move down the hierarchy (transfer time increases) --> caches further down the hierarchy are more likely to use **write-back**

H3 §6.5 Writing Cache-Friendly Code

- Quick review:
    - **Block** - a fixed-size packet of information that moves back and forth between a cache and main memory (or a lower-level cache)
    - **Line** - a container in a cache that stores a block, as well as other information such as the valid bit and the tag bits
    - **Set** - a collection of one or more lines. Sets in direct-mapped caches consist of a single line.

- Programs with **better locality** will tend to have **lower miss rates**

  --> tend to **run faster**

- To make our code "**Cache friendly**",
  1. **Make the common case go fast**
     - focus on the inner loops of the core functions
  2. **Minimize the number of cache misses in each inner loop**
     - loops with better miss rates

[Example C code]

```c
int sumvec(int v[N]) {
  int i, sum = 0;

  for (i = 0; i < N; i++)
    sum += v[i];
  return sum;
}
```

- **Temporal locality** -- local variables **i**, **sum**

  --> optimizing compiler will **cache** them in the **register file**

  - repeated references to local variables --> cache in register file
- **Spatial locality** -- **stride-1** reference pattern
  - caches at all levels of the memory hierarchy store data as contiguous blocks
  - important in programs operate on **multi-dimensional arrays**

[Example C codes]

```c
int sumarrayrows(int a[M][N]) {
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
}
```

--> will have the same desirable **stride-1 access pattern**

```
int sumarraycols(int a[M][N]) {
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];
  return sum;
}
```

--> if the array is larger than the cache, each and every access will miss

--> siginificant impact on running time (25x performance diff)

§6.6 Putting It Together: The Impact of Caches on Program Performance

¶6.6.1 The Memory Mountain

- **Read throughput (read bandwidth)** -- rate that a program reads data from the main memory
- **Memory mountain** -- two-dimensional function of **read throughput** vs **temporal & spatial locality**
  - slope of spatial locality decreases as the stride increases

- Performance of the memory system is not characterized by a single number
  - exploit...
    - temporal locality so that heavily used words are fetched from the L1 cache
    - spatial locality so that as many words as possible are accessed from a single L1 cache line

¶6.6.2 Rearranging Loops to Increase Spatial Locality

Consider the program of multiplying a pair of $n \times n$ matrices: $C = AB$

If $n = 2$, then

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

where

$$c_{11} = a_{11}b_{11} + a_{12}b_{21}$$

$$c_{12} = a_{11}b_{12} + a_{12}b_{22}$$

$$c_{21} = a_{21}b_{11} + a_{22}b_{21}$$

$$c_{22} = a_{21}b_{12} + a_{22}b_{22}$$

--> typically, matrix multiply function is implemented using **three nested loops** with indices i, j, k

For analysis, assume...

- Each array is an $n \times n$ array of **double** , with **sizeof(double)** =8
- There is a single cache with a 32-byte block size (B =32)
- The array size n is so large that a single matrix row does not fit in the $L1$ cache
- The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions

[6 Functionally equivalent versions of matrix multiplication]

```
/* Version ijk */
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] += sum;
  }

/* Version jik */
for (j = 0; j < n; j++)
  for (i = 0; i < n; i++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += A[i][k] * B[k][j];
    C[i][j] += sum;
  }

/* Version jki */
```

```
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++) {
      r = B[k][j];
      for (i = 0; i < n; i++)
        C[i][j] += A[i][k]*r;
    }

  /* Version kji */
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++) {
      r = B[k][j];
      for (i = 0; i < n; i++)
        C[i][j] += A[i][k]*r;
    }

  /* Version kij */
  for (k = 0; k < n; k++)
    for (i = 0; i < n; i++) {
      r = A[i][k];
      for (j = 0; j < n; j++)
        C[i][j] += r*B[k][j];
    }

  /* Version ikj */
  for (i = 0; i < n; i++)
    for (k = 0; k < n; k++) {
      r = A[i][k];
      for (j = 0; j < n; j++)
        C[i][j] += r*B[k][j];
    }
```

[Analysis of matrix multiplication (inner loops)]

Version	Per Iteration					
(Class)	Loads	Stores	A_miss	B_miss	C_miss	Tot_miss
ijk & jik	2	0	0.25	1.00	0.00	1.25
jki & kji	2	1	1.00	0.00	1.00	2.00
kij & ikj	2	1	0.00	0.25	0.25	0.50

- ijk, jik - versions
  - scan a row of array A with a **stride of 1**

    - miss rate for A = 0.25 misses / iteration
  - scan a column of B with a **stride of** n
    - each access of array B results in a miss (1 misses / iteration)

  --> Total **1.25 misses** / iteration

- jki, kji - versions
  - each iteration performs 2 loads and 1 store
  - scan the columns of A and C with a **stride of** n
    - miss on each load

  --> Total **2 misses** / iteration

- kij, ikj - versions
  - each iteration performs 2 loads and 1 store
  - scan B, C row-wise with a **stride-1 access patten**
    - each with 0.25 misses / iteration

  --> Total **0.50 misses** / iteration
  - In this case, results in the greatest performance
    - miss rate being a better predictor of performance than the total number of memory access
    - **prefetching hardware** recognizes the **stride-1** access pattern, and keeps up with memory accesses in the tight inner loop

H5 ¶6.6.3 Exploiting Locality in Your Programs

- Programs with good locality access most of their data from fast cache memories ,while programs with poor locality access most of their data from the relativley slow DRAM main memory

1. Focus on the **inner loops** --> bulk of the computations and memory accesses
2. Maximize the **spatial locality** by reading data objects **sequentially (Stride-1** reference pattern)
3. Maximize the **temporal locality** by using a data objects **as often as possible** once it has been read from memory

H3 §6.7 Summary

- Basic storage techniques
  - **RAMs** (Random Access Memory)
    - **Static** RAM (SRAM)
      - faster, more expensive
      - used for cache memories
    - **Dynamic** RAM (DRAM)
      - slower, less expensive
      - used for the main memory & graphic frame buffers
  - **ROMs** -- Nonvolatile memory
    - retain information even if the supply voltage is turned off
    - used to store **firmware**
  - **Disks**
    - Rotating disks -- mechanical nonvolatile storage that hold enormous amounts of data at a low cost per bit, but with much longer access times than DRAM
    - Solid state disks (SSDs) -- based on nonvolatile flash memory, somewhere in between DRAM and Rotating disk
- Price & Performance properties of storage technologies are changing at dramatically different rates
  - DRAM and disk access times are much larger than CPU cycle times
    - systems bridge these gaps by organizing memory as a **hierarchy of storage devices**
    - Programs with good locality **run at the rate of the hiher levels, but at the cost and capacity of the lower levels**