Bitwise operators and the Memory Hierarchy

William Hendrix

Lecture 21

Today

- Digital representation
- Bitwise operations
- Memory hierarchy
- Caching

Data representation

- Integer data: binary numbers
  - 0, 1, 10, 11, 100, etc.
  - Unsigned `int` go up to 4B (32 bit)
  - Signed `int` represent -2B to 2B

- Decimal numbers: IEEE floating point (32 or 64 bits)
  - 1^{st} bit: positive/negative
  - 2^{nd}-8^{th} bit: exponent ($6.02 * 10^{\underline{\mathbf{23}}}$)
  - 9^{th}-32^{nd} bit: mantissa ($\underline{\mathbf{6.02}} * 10^{23}$)
  - Special values to represent +/- infinity and NaN (not a number)
    - `<cmath>:  isinf(x), isnan(x)`
  - Double precision has 11-bit exponent and 52-bit mantissa

- Text data: ASCII codes
  - Codes in the range 0-255 that represent all standard characters

Bitwise operations

- Operations that directly affect the bits of data
- Code is efficient but hard to read/understand
  - Use at your own risk
- Left and right shift (<< and >>)
  - Bits move left or right the given number of places
    - Precedence is very low; use () as necessary
  - **Example:** `0x68 << 4 = 0x680; 0x68 >> 1 = 0x34`
  - Since bits worth twice as much moving left, left shift effectively multiplies by 2^n (barring overflow)
    - Right shift divides by 2^n (integer division/floor behavior)
    - Much more efficient than "true" multiplication
  - Negative integers may copy top bit when right shifting
    - Left shift always fills in zeros
- Bitwise negation (~)
  - Flip all bits
  - **Example:** `~0xf0 = 0x0f`

Bitwise operations (cont'd)

- Bitwise and (`&`) and or (`|`)
  - Each bit of result is the and (or) of corresponding input bits
  - **Example:** `0x6f & 0x30 = 0x20, 0x6f | 0x30 = 0x7f`

```
    01101111          01101111
  & 00110000        | 00110000
    --------          --------
    00100000          01111111
```

  - Does not support short-circuit evaluation
    - Do not use in place of logical and (`&&`) and or (`||`)
- Use bitwise or to turn bits on or test
  - `x |= 0x80` will set top bit of smallest byte
- Use bitwise and to test bits or turn bits off
  - `x & 1` is true (non-zero) if the last bit of `x` is set
  - `x &= ~0x80` will turn off the top bit of the last byte
- Bitwise xor (`^`)
  - Exclusive or (one or the other, but not both)
  - Some clever applications

Flags

- Binary data can be treated as a sequence of `bool` (0/1) values
- Represent a combined state by taking bitwise or (`|`) of conditions
- Example

```
//Constants should always be powers of 2
#define TWO_DIGIT_YEAR   0x1
#define HYPHEN_SEPARATOR 0x2
#define NAME_OF_MONTH    0x4
#define PRINT_NEWLINE    0x8
void Date::print(char flags);
//Use flags & constant to test printing options

//In main():
Date today;
today.print(NAME_OF_MONTH | PRINT_NEWLINE);
```

Negative integers

- Negative numbers represented using *two's complement*
  - Highest bit value is negated
    - **Example** (8 bits, signed):
      - `0x80`: -128
      - `0xff`: -1
  - Largest negative number: only first bit set
  - Largest positive number: all but first bit set
  - All bits set: -1
  - Flip all bits of x: $-x - 1$
- Advantage
  - No special cases needed to add positive and negative numbers
    - Easier to design circuits to quickly add numbers
  - **Example:** `0xff` + `0x01` = `0x100`
    - Carry bit is lost due to max size of data type
- **Overflow:** exceeding the max size of data
  - Occurs when carrying into/borrowing from the topmost bit
  - Eventually, integers will "wrap around" and become negative

CPU organization

- All operations take place on data in registers
  - 32 or 64 registers in CPU
  - Each CPU cycle, instructions are loaded, interpreted, and run (*fetch*, *decode*, *execute*)
- Different types of instructions
  - Arithmetic
    - Addition, multiplication, division, etc. (integer and floating point)
  - Control flow
    - Decide what to do next based on comparison (equal, less than, etc.)
  - Memory operations
    - Store or retrieve information from memory/disk
- Fetch-decode-execute cycle occurs ~2B times per second
  - **Major challenge:** How do we supply data to CPU fast enough for it to keep running?

The Memory Hierarchy

- Organization of data storage hardware
  - Higher levels are faster, more expensive, and hold less

Memory level	Access time	Capacity
Registers	~1 ns	< 1 kB
L1 cache	10-100 ns	10-50 kB
L2 cache	100-1000 ns	1-5 MB
L3 cache*	0.5-10 μs	10-1000 MB
Main memory (RAM)	1-10 μs	1-10 GB
Nonvolatile memory (e.g., flash)*	10-100 μs†	100-1000 GB
Hard disk	1-10 ms	100-1000 GB
Tertiary storage (tape)*	Sec to min	1-10 TB per tape

CPU is here

Variables are here

Files are here

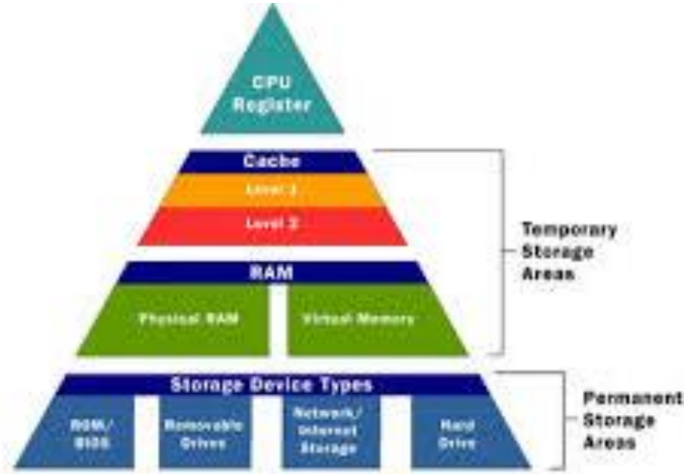
* Not common or present on all systems

†Writing data to flash memory is much slower

- Values in table are approximate; will change over time

Caching

- **Problem**: main memory is ~1000 times slower than CPU!
  - Files are even worse!
- **Solution**: caching
  - Store recent data in higher levels of memory hierarchy
    - E.g., *buffering* file data in memory
  - Subsequent accesses much faster
  - Retrieves a *page* of memory at once



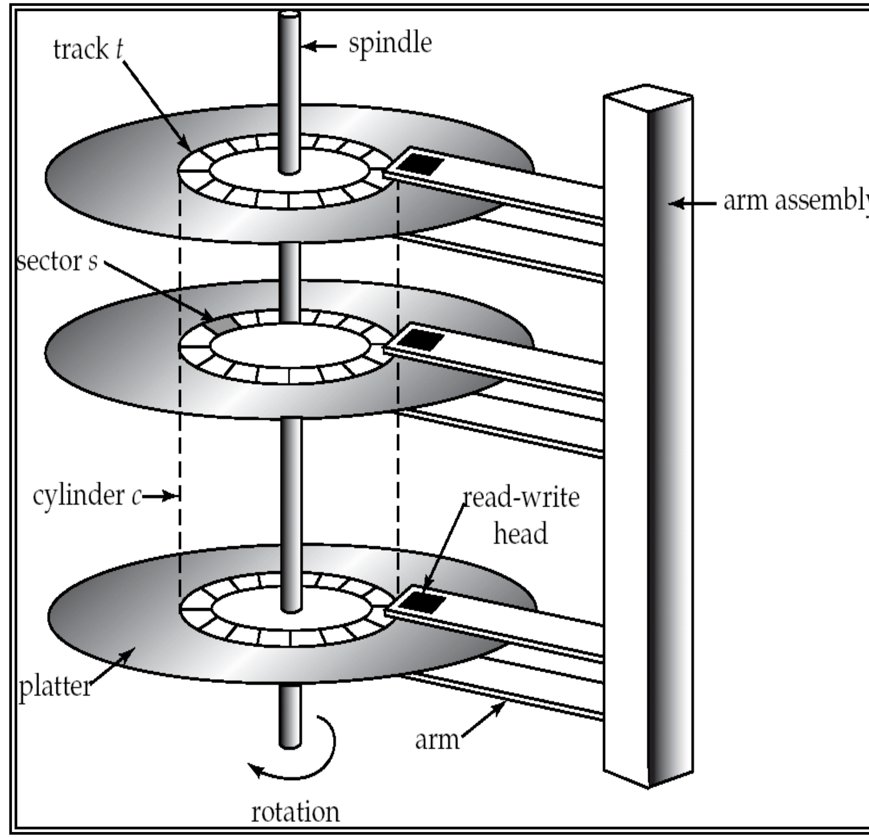
- **Example**

```
for (int i = 0; i < nrow; i++)
  for (int j = 0; j < ncol; j++)
    sum += arr[i*nrow+j];
```
(consecutive memory access)

```
for (int i = 0; i < ncol; i++)
  for (int j = 0; j < nrow; j++)
    sum += arr[j*nrow+i);
```
(random access)

Hard disks

- Magnetic disks
  - Multiple *platters* per drive
  - Divided into concentric *tracks*
  - Tracks split into radial *sectors*
  - Access to outer tracks is faster



- Latency
  - Two main sources
  - Seek time
    - Moving to correct track
  - Rotation latency
    - Moving to correct sector

- Throughput
  - Data after the first sector is retrieved much faster
  - Can be improved even further by reading from multiple disks at once
    - RAID