

Memory

CSE 361S

Memory Technologies

- First we will cover several memory technologies and their primary properties
- They tend to be known by their acronyms (e.g., SRAM, DRAM, SDRAM)

SRAM (Static Random Access Memory)

- Semiconductor memory that retains its contents as long as power is provided
 - can be made very fast
 - not the highest density
 - very simple to use and interface
 - same fabrication process as logic

DRAM (Dynamic RAM)

- Semiconductor memory that must be periodically refreshed
 - longer access latency than SRAM
 - highest density of semiconductor memories
 - more involved interface
 - some differences in fabrication process
 - SDRAM is synchronous DRAM, with higher transfer rates than traditional DRAM

Non-volatile Memory

- ROM (read only memory) – contents set once and then not altered again, typically set during fabrication
- PROM (programmable ROM) – can have contents set in the field instead of factory, but still only once
- EPROM (erasable PROM) – can erase contents, but it is high-overhead process (e.g., UV light)

More Recent NV Memory

- EEPROM (electrically erasable PROM) – can erase contents electrically, but still a slow process (ms vs. ns)
- Flash – form of EEPROM which is block erasable (cannot erase individual addresses)
 - Used extensively in solid-state drives

Non-semiconductor Memory

- MRAM – magnetic memory using thin films organized to look and act like flash
- Disk – magnetic storage arranged on rotating platters
- Tape – magnetic storage arranged linearly

Important Tradeoff

- With a few minor exceptions, previous list is ordered from fastest to slowest
- It is also ordered from lowest to highest density
- We can store more data per unit volume if we are willing to pay longer lookup (access) times
- Conversely, we can access data faster if we are willing (or able) to retain less

Locality

- *The principle of locality* allows us to have our cake and eat it too
 - Using small fast memories and
 - Using large slow memories,
 - We can mimic large fast memories

Spatial Locality

- If a program accesses a memory location, it is likely to access nearby memory locations soon

```
int sumvec() {
    int sum, i;

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

- Both code and vec exhibit spatial locality in this example

Temporal Locality

- If a program accesses a memory location, it is likely to access the same location again soon

```
int sumvec() {
    int sum, i;

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

- sum exhibits temporal locality in this example

Memory Hierarchy

- Principle of locality enables one to build small fast memories in front of larger slower memories and have most references be serviced by the fast memories. The resulting memory system is hierarchical:
 - A – register file
 - B – L1 cache (small on-chip SRAM)
 - C – L2 cache (larger on-chip SRAM)
 - D – L3 cache (larger-yet on-chip SRAM)
 - E – main memory (very large off-chip DRAM)
 - F – disk (either magnetic or solid-state)

- Put frequently used data closer to the processor, diminishing the time required to access them when they are needed
- Use assumption of locality to decide what is positioned at each level of the hierarchy
- From point of view of the ISA, the only two levels that are visible are A and E (we're ignoring the file system right now); however, all of B through F are used to implement the view of E presented by the ISA

- Movement of data between levels is in fixed size units called "blocks"
 - $A \leftrightarrow B$ word (32 or 64 bits) reg load/store
 - $B \leftrightarrow C$ 4 to 8 words cache miss
 - $E \leftrightarrow F$ 4 Kbyte or more page fault
- On miss, move block from level $i+1$ (where it is found) to level i (where it is needed)
- This movement might displace something from level i
- Therefore, we need a replacement policy

Replacement Policies

- Random
 - Choose a random block in level i and boot it out
- Least Recently Used (LRU)
 - Choose the least recently used (accessed) block in level i as the one to boot out
- Cheap implementations that approximate LRU work surprisingly well

Understanding Cache Function

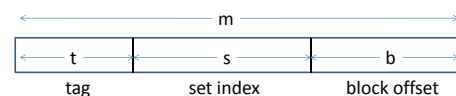
- Consider a two-level cache hierarchy
 - $B \leftrightarrow E$
 - Cache \leftrightarrow Main memory
- Cache is organized as collection of "sets"
 - If $S = 2^s$ is # of sets, s is # of bits to ID set
- Each set contains 1 or more cache "lines"
 - Each cache line stores 1 block
 - E is number of lines per set
 - If $B = 2^b$ is block size (in bytes), b is # bits to ID byte within a block

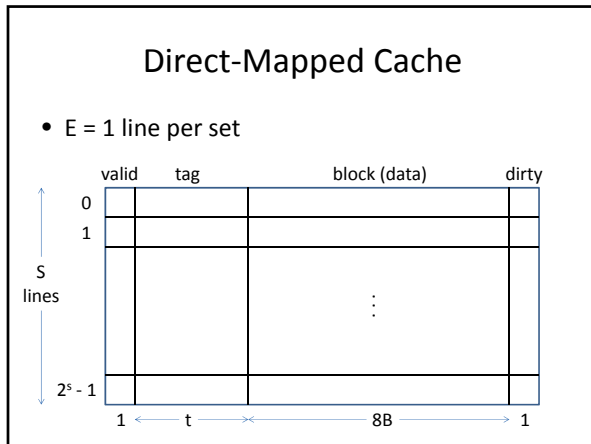
$S = 2^s$ is # of sets, $E = \#$ lines per set,
 $B = 2^b$ is block size

- Each cache line stores:
 - Block (data)
 - Valid bit
 - Tag bits (where is main memory does this block reside?)
 - Dirty bit (is memory up to date?)
- If $M=2^m$ is size of addressable memory, m is # of bits of logical address

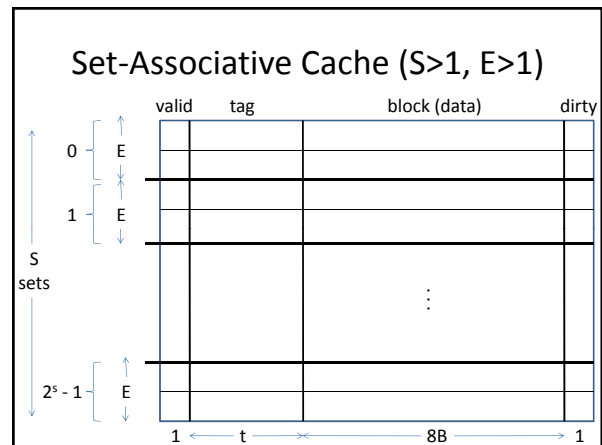
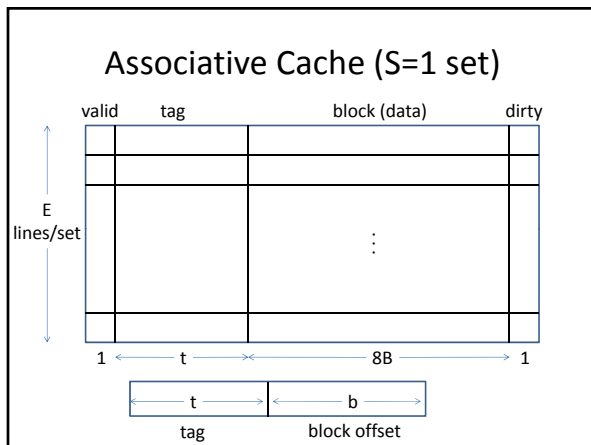
4-tuple: (S, E, B, m)

- Used to characterize cache organization:
 - $S = 2^s$ is # of sets
 - $E = \#$ lines per set
 - $B = 2^b$ is block size (in bytes)
 - m is # of bits of logical address
- Divide m address bits into 3 components:





- ### Write Policy
- *write through* (write to cache and memory both)
 - *write back* (write to cache and later write back to memory, if dirty when evicted)
 - On write miss – *write allocate* or *write no-allocate*
 - Classic pairs:
 - write through, no-allocate
 - write back, allocate



- ### Some Example Caches
- AMD Opteron, ca. 2006-2007:
 - 64 KB L1 i-cache, 2-way set assoc., 64 bytes/block
 - 64 KB L1 d-cache, 2-way assoc., 64 B/block
 - 1 MB L2 cache, unified, 16-way assoc., 64 B/block
 - AMD “Bulldozer”, 8 cores, ca. 2011:
 - 64 KB L1 i-cache, 2-way assoc., shared w/ 2 cores
 - 16 KB L1 d-cache, 4-way assoc., not shared
 - 2 MB L2 cache, shared w/ 2 cores
 - 8 MB L3 cache, shared w/ 8 cores