

Memories

CPE380, Fall 2025

Hank Dietz

<http://aggregate.org/hankd/>

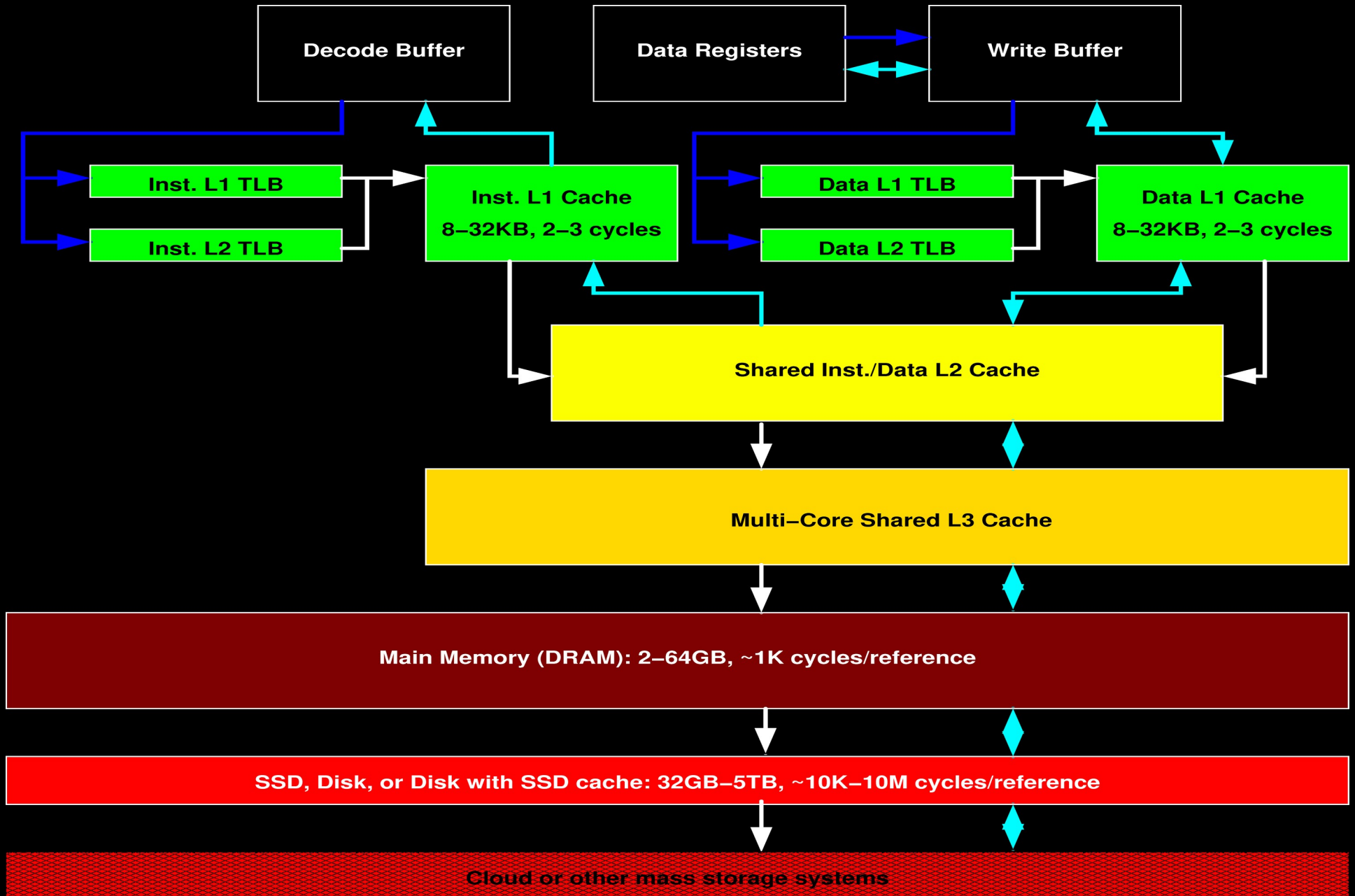
Memory Terminology

- **Volatile** – power off, data fades away
- **ROM** – non-volatile Read Only Memory
- **PROM, EPROM, OTP, EEROM, Flash, 3DXPoint** – types of non-volatile programmable memory
- **RAM** – **R**andom **A**ccess **M**emory (mostly volatile)
 - **Core** – non-volatile magnetic RAM technology
 - **SRAM** – Static RAM, fast but big cells
 - **DRAM** – Dynamic RAM, slow but small cells
 - **EDO, SDRAM, DDR, RamBus** – DRAM types
 - **CXL** – **C**ompute **eX**press **L**ink
- **Registers, Cache** – fast working memories

More Memory Terminology

- Punched cards
- Punched paper tape
- Tape, Magtape
- Drum
- Disks:
Floppy, Hard, Magneto-optical, Compact Disc,
Digital Video (Versatile?) Disc, Blu-ray
- Solid State Disk, Optane





The Memory Hierarchy

Regs: a few kB, 1 cycle

L1 Cache: 64kB, ~4 cycles

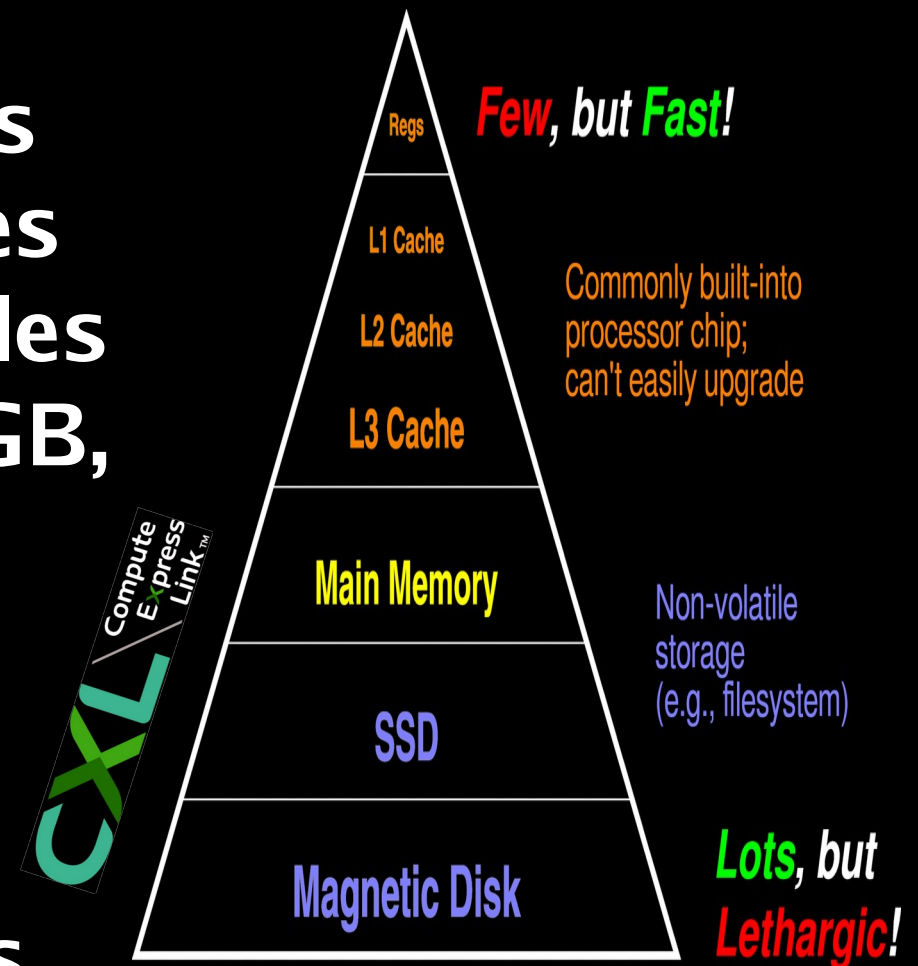
L2 Cache: 2MB, ~12 cycles

L3 Cache: 16MB, ~43 cycles

Main Memory: 32GB, \$4/GB,
~248 cycles

SSD: 512GB, \$0.10/GB,
~200k cycles,

Magnetic Disk: 14TB,
\$0.018/GB, ~20M cycles



How The Hierarchy Helps

- **Main memory** is **too slow** & **too small**; we want:
 - Capacity & cost of the big stuff (e.g., disk)
 - Access speed of the fast stuff (e.g., regs)
- If most things are in the top layers when we want to access them, this works...
this is what we call **good locality of reference**
- Two basic types of **locality**:
 - **Temporal**: same thing accessed again soon
 - **Spatial**: nearby thing accessed soon

Managing The Hierarchy

- Everything “lives” in the bottom layer (e.g., disk)
- Drop copies in higher layers to access faster
 - **SSD** and **disk** are slow enough that **OS software** can manage copying
 - **Caches** need **hardware management**
 - **Register** copy management is explicitly done by the **compiler** via load/store instructions (GPUs & microcontrollers often also have **local memories** managed by the compiler)

Making An Access

- Does this layer have a copy of what I want?
No: a **miss** needs to request thing from below
Yes: a **hit** operates on the copy here
- What high **hit ratio**: $\text{hits}/(\text{hits}+\text{misses}) \approx 1$
- How big a block to copy?
 - Temporal locality \Rightarrow 1 word
 - Spatial locality \Rightarrow a bigger chunk holds more nearby things, but takes longer to copy
 - Can transfer bigger blocks from SSD, disk
 - Usually 32B/64B, but $\geq 512\text{B}$ from SSD, disk

What Does Cache Look Like?

- Cache is basically a hardware hash table
 - Index is `hash (address)`
 - Offset of B in Data is `offs (address)`
 - At least two fields: `tag`, `data`
- Suppose 16kB cache with 16B/line with
`hash (addr) = addr [15:4] ; offs (addr) = addr [3:0] ;`

32' h0000002a
32' h70c00010
32' h00000024
32' h08000003
32' h12300016

Line Index	Tag	Data
0:	28'h0800000	-----
1:	28'h70c0001	-----
2:	28'h0000002	-----
...		
1023:		

Cache Associativity?

- **Set size (ways)** is like hash table bucket size
 - **Direct mapped**: each addr maps to 1 line
 - **Set associative**: select 1 of s lines
 - **Fully associative**: set size = # lines in cache
- Ways > 1 means there are ways choices for where to put a line, might improve hit rate
- Ways > 1 requires comparing to ways tags
 - Read entire set, need ways comparators
 - Check tags sequentially, takes ways clocks

Cache Associativity Example

- 2-way set associative; $1024/2=512$ buckets
- Staying with a very simple hash function

`hash(addr)=addr[14:4]; ofs(addr)=addr[3:0];`

32'h0000002a
32'h70c00010
32'h00000024
32'h08000003
32'h12300016

Line Index	Tag	Data
0:	28'h0800000	-----
1:	28'h70c0001	-----
	28'h1230001	-----
2:	28'h0000002	-----
...		
511:		

Basic cache design issues

- Placement (mapping)
 - the hash function
- Identification
 - which line within the set do I want?
- Replacement policy
 - which line gets kicked-out to make space?
- Write strategy
 - how far back do writes go and when?

Which replacement policy?

- Direct mapped → no choice
- Random
- Replace a clean (not dirty) line
- LRU (Least Recently Used): mark when line is accessed, replace not accessed recently
- LFU (Least Frequently Used)
- MRU and MFU: Most “”
- Belady's MIN: replace line not used for the longest time in the future (how to know this?)
- Compiler-driven; e.g., using cache bypass

Write strategy

- Write through
 - Write always goes to main memory
 - Easy; needed for I/O devices in memory
- Write back
 - Write only when line replaced, saving traffic
 - Could do lazy writes when not busy
 - May need to read on miss to get rest of line
- Write allocate: write back, but don't wait for line to be read first; aka pre-arrival caching

Write Buffer

- Sort-of like a “level 0 data cache”
(faster because no TLB in front of it...
but we haven’t discussed TLBs yet)
- Buffer can re-group writes to form write to a
larger fraction of a line (not just one byte or
word)
- Need to be careful about task switches, etc.;
may have to flush write buffer often

What causes a miss?

- **Compulsory**
 - Never touched this block before
 - **Shared fetch effect** can avoid these when another process touches what I want first
- **Capacity**
 - Could have been from cache, but didn't fit
- **Conflict**
 - Could have fit, but cache mapping had a conflict with another line that caused this line to be replaced (e.g., direct mapped)

Cache optimizations

- Larger total cache size
 - Fewer capacity & conflict misses
 - Dumber replacement policy works ok
 - Increases hit time, die space, and power use
- Larger line size
 - Fewer compulsory misses (spatial locality)
 - More capacity & conflict misses
 - Increases miss penalty (block transfer time)

More cache optimizations

- Higher associativity
 - Reduces conflict misses
 - Increases hit time & power use
- More levels of cache
 - Smaller, faster, upper-level caches
 - More complex hardware structure

Still more cache optimizations

- Priority to read misses over writes
 - Reduces miss penalty
 - Modest increase in design complexity
- Avoiding address translation before indexing
 - Reduces hit time
 - Not what operating systems expect
 - Frequent cache flushes or need PID tags

Compiler optimizations

- Restructure code to change data access pattern
 - Group data (**data layout**)
(many languages heavily constrain this)
 - Reorder accesses (**loop transformations**)
- Prevent cache pollution
 - Why cache what you get from a register?
 - Often double-map: cache / bypass
- Avoid saving data that isn't used again

Compiler optimizations

Merging/splitting arrays

- Array elements accessed together can be grouped together to enhance spatial locality
- Also separate those not accessed together

E.g., suppose `a[i]` and `c[i]` accessed together:

```
int a[N], b[N], c[N];  
struct { int a, b, c; } abc[N];  
struct { int a, c; } ac[N]; int b[N];
```

Compiler optimizations

Loop interchange

- Loop nest traversal order matches data layout
- Improves spatial locality

E.g., if `a[0][0]` is next to `a[0][1]`:

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

Compiler optimizations

Loop fusion

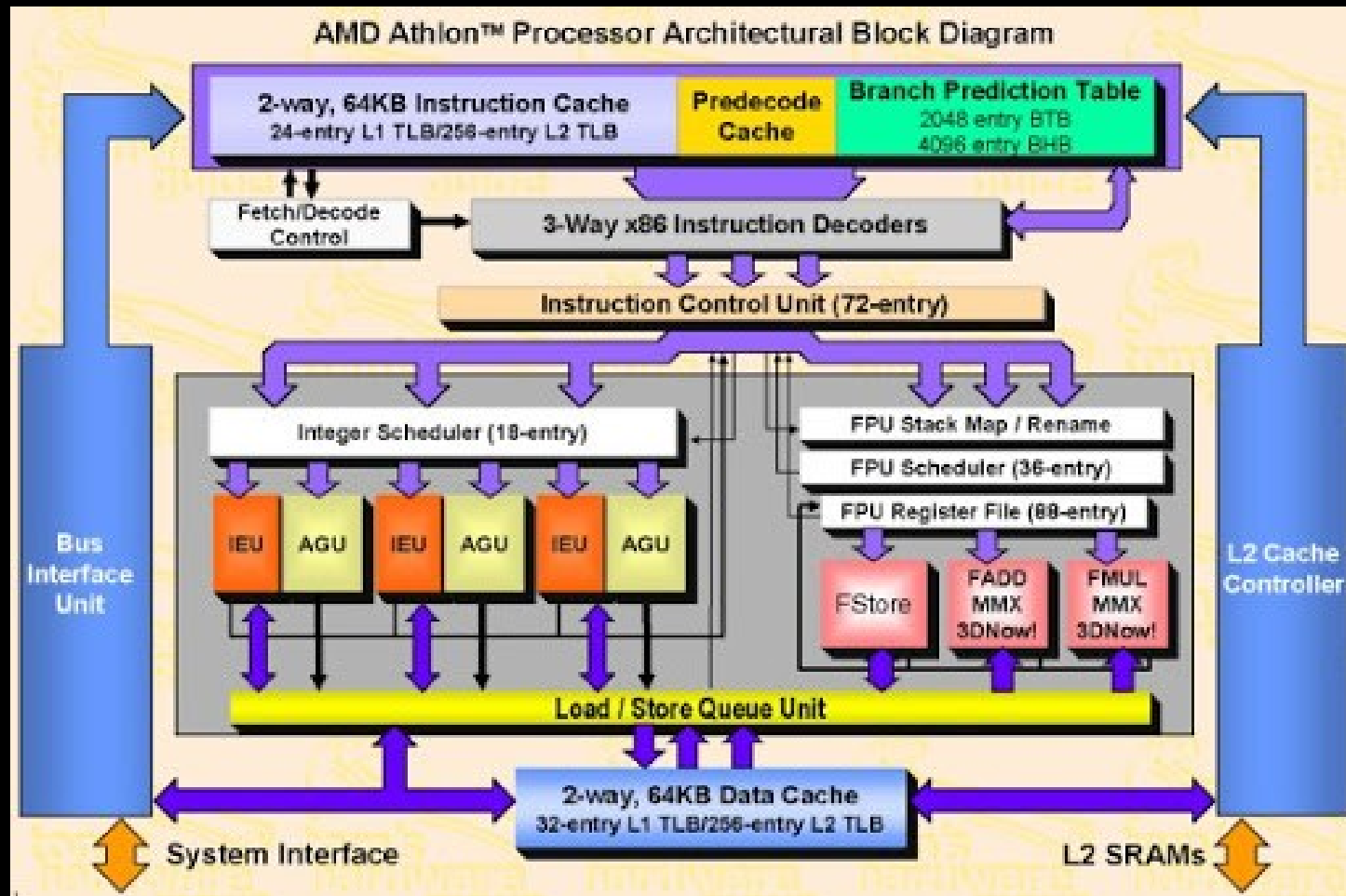
- Fuse loops that work on similar data
- Improves spatial locality

```
for (i=0; i<N; ++i)
    for (j=0; j<M; ++j)
        a[i][j] = b[i][j] + c[i][j];
for (i=0; i<N; ++i)
    for (j=0; j<M; ++j)
        d[i][j] = a[i][j] * c[i][j];
for (i=0; i<N; ++i)
    for (j=0; j<M; ++j) {
        a[i][j] = b[i][j] + c[i][j];
        d[i][j] = a[i][j] * c[i][j]; }
```

Prefetching

- Software (by compiler)
 - Hoist load to earlier position in program
 - *Suggest* hardware load into cache
- Hardware
 - Assume or recognize reference pattern and request expected next early
 - Line +/-1, strided, other patterns
- Works better for instructions than data
- Generally can abort a prefetch to cache, prefetches can't fault (no exceptions)

A Real Processor: AMD Athlon



3 caches: split Instruction/Data L1, unified L2

Consistency Models

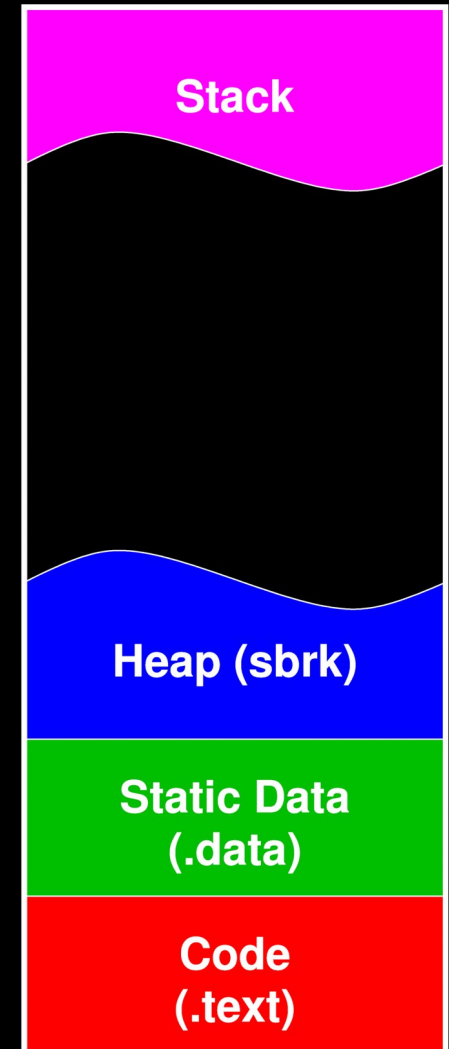
- The **volatile** keyword in C/C++ gives potential memory order constraints
- **Strict**: everybody sees result at next tick
- **Sequential**: everybody sees things as if they happened in a sequential order
- **Weak Ordering**: memory barriers/fences force ordering of before vs. after

Cache Coherence

- How one maintains consistency
- What to do when something writes?
 - **Invalidate**: mark/discard old entries
 - **Update**: use the write data to update
- Who to notify?
 - **Snooping**: everybody watches
 - **Ownership**: only talk to owner
 - **Directory**: permissions, who to notify
- **MESI Protocol**: Modified (dirty), Exclusive, Shared (clean), Invalid – 4 line states

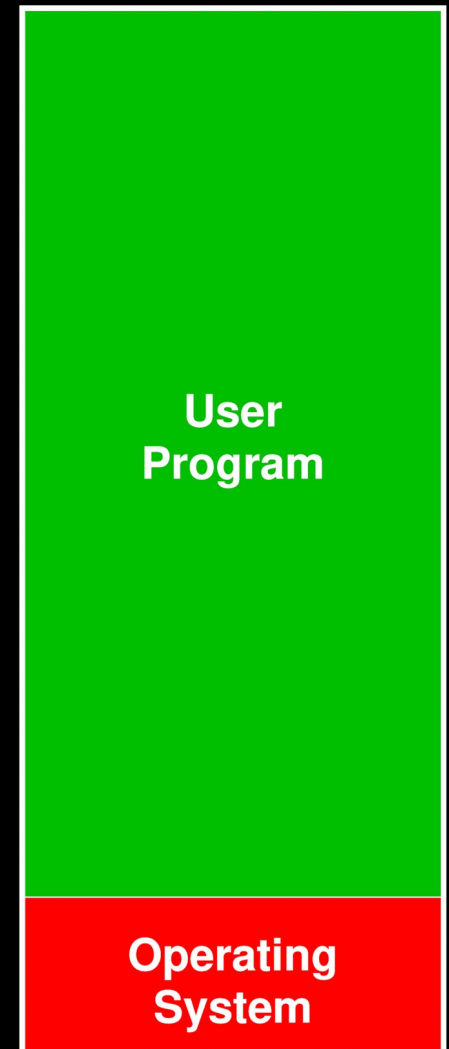
Memory Map of a Process

- Arranging stuff in memory:
 - Code starts at low address (0)
 - Static (fixed address) data
 - Heap typically grows up
 - Stack typically grows down
- Very bad if stack meets heap
 - Stack grows to cover SP
 - Heap grows by explicit calls to `sbrk()`, `malloc()`, `new`, etc.



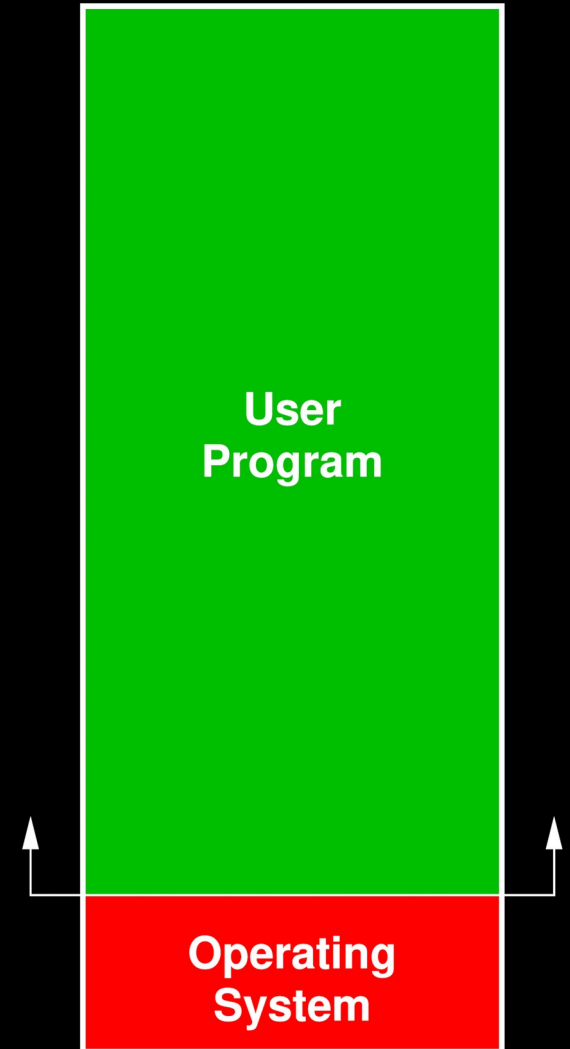
Memory Map of a Computer

- Originally, loaded one program at a time
 - OS was mostly a “loader”
 - User code could do anything
- Still a fairly common model for embedded computers and various microcontrollers



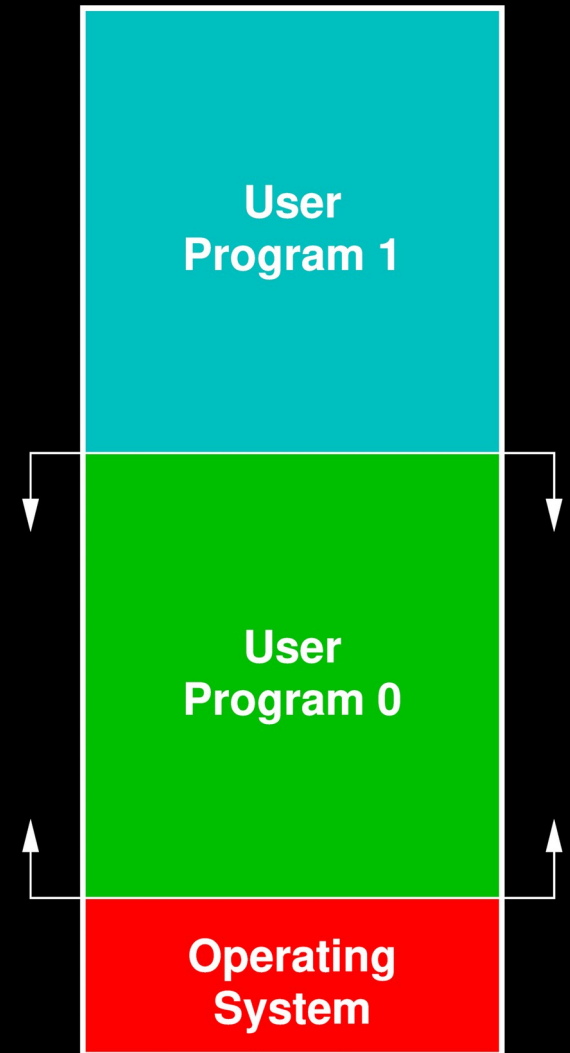
Protection

- A stray user program could corrupt the OS... add a **fence register** to protect it
- Processor respects fence unless in privileged mode
 - Become priv by system call or interrupt to trusted address
 - Surrender priv when return to user program

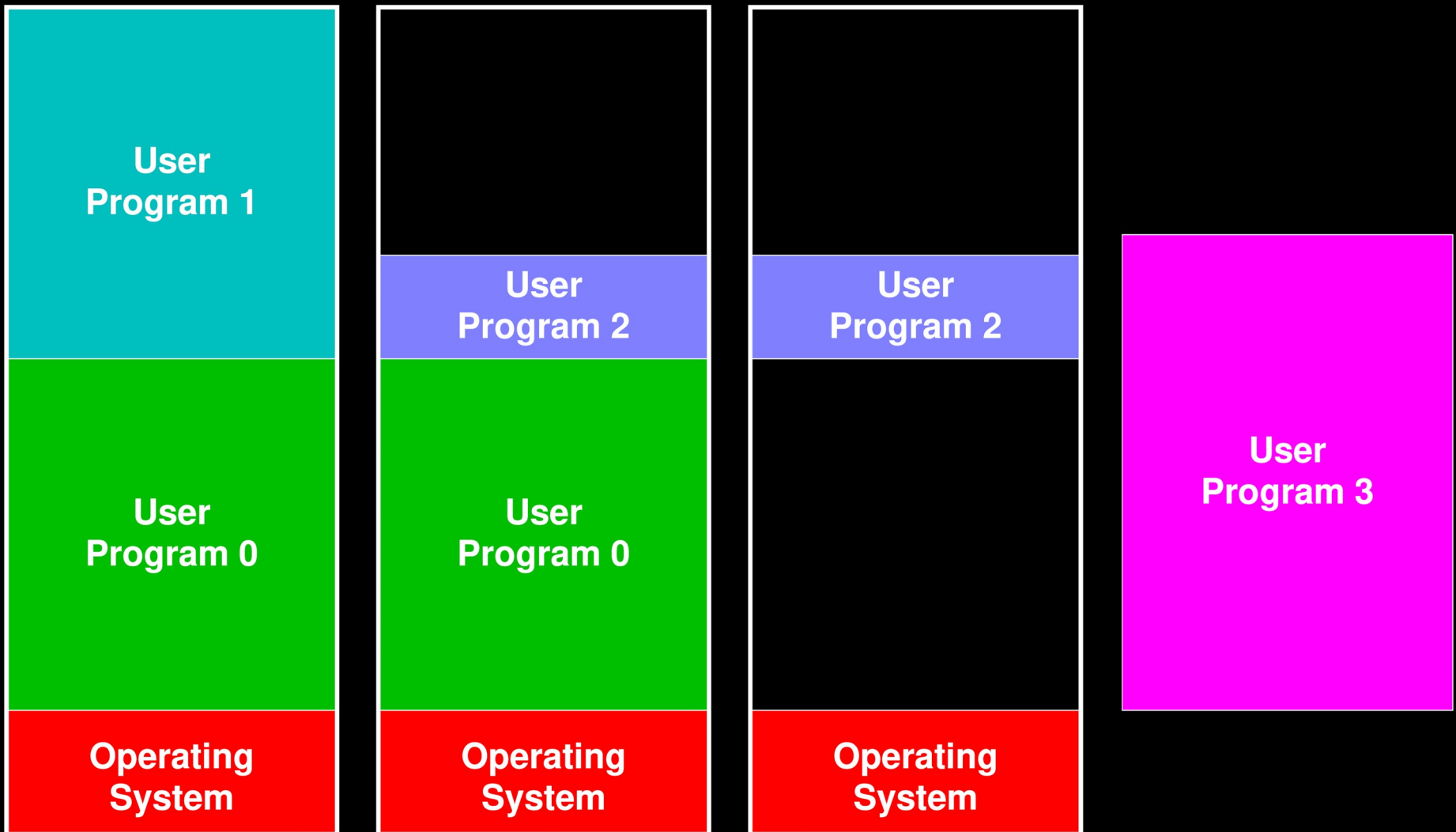


Batch Scheduling and Timesharing

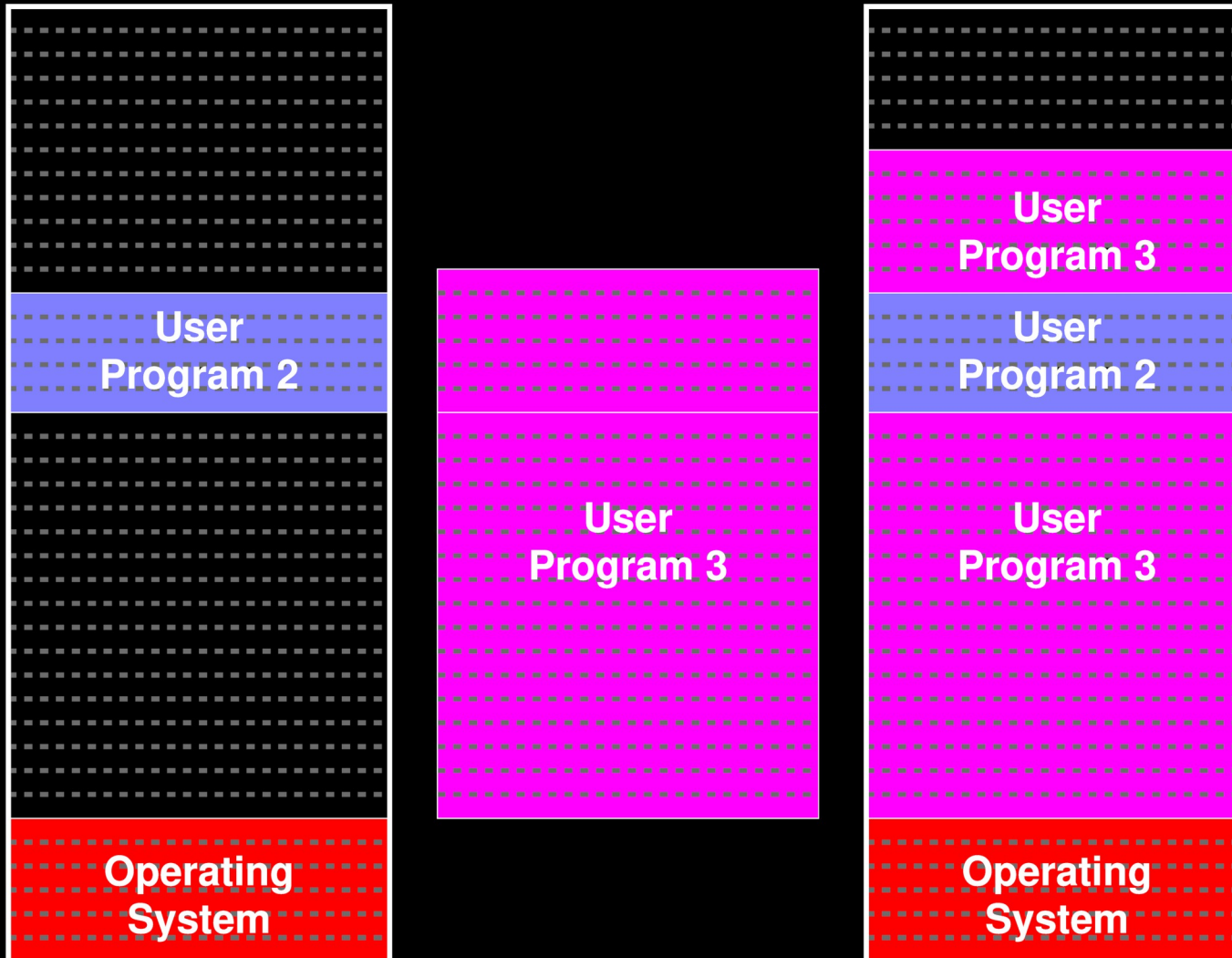
- Don't want expensive computer idle while waiting for printer, etc.
 - Load multiple jobs
 - Run 1 while 0 is waiting
- **Timesharing**: alternate running so all processes make progress
- Want two fence registers...



Memory Fragmentation



Memory Page Tables



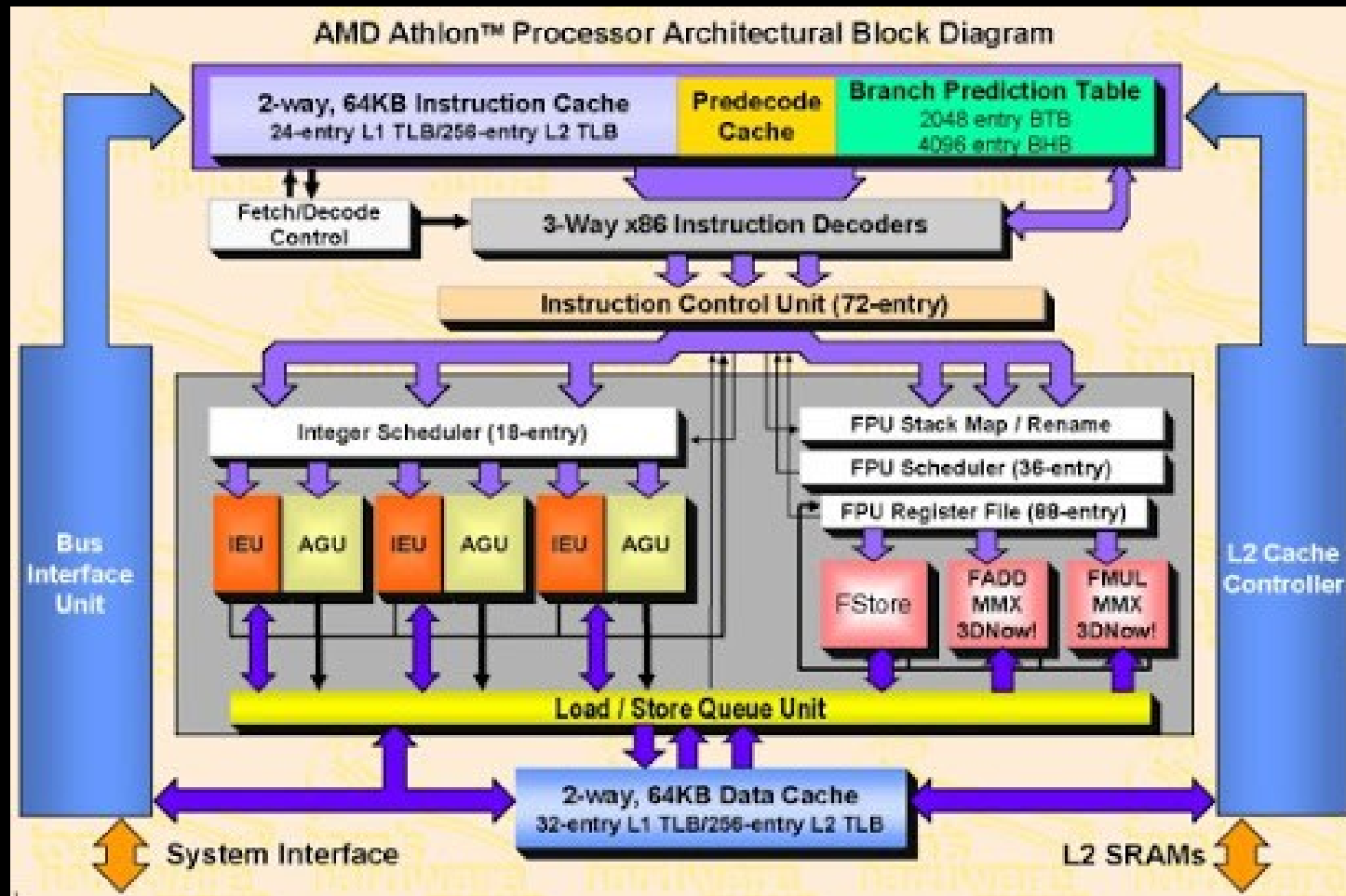
Logical vs. Physical Addresses

- Memory is divided into **pages**
 - Classically, **each page is 4kB**
 - Most systems also support **4MB pages**
- Processor outputs **logical** (aka **virtual**) address
 - Top bits identify page number, bottom offset
 - **Page table** says where each page number is
 - **Physical address** substitutes page address in memory for logical page number

Page Table Issues

- 4kB pages are quite small...
 - IBM PC had 128KB memory, so 32 entries
 - With 4GB memory, need 1M page entries!
 - Each process needs a page table!
- Translation Lookaside Buffer (TLB)
 - Essentially a cache for page table entries
 - Translation typically before L1 cache...
so the TLB needs to be fast, hence small
 - Can make L1/L2 TLBs, separate for I/D;
don't wait for L1 miss to start search of L2

A Real Processor: AMD Athlon



4 TLBs: L1+L2 for each of code and data

Page Table Issues

- What happens for a TLB miss?
 - Instruction gets stopped, then restarted when the TLB has the appropriate entry...
this requires hardware support
 - Must fetch page table entry (from memory)
- Thus, data in cache might not be accessible because TLB can't translate the address:

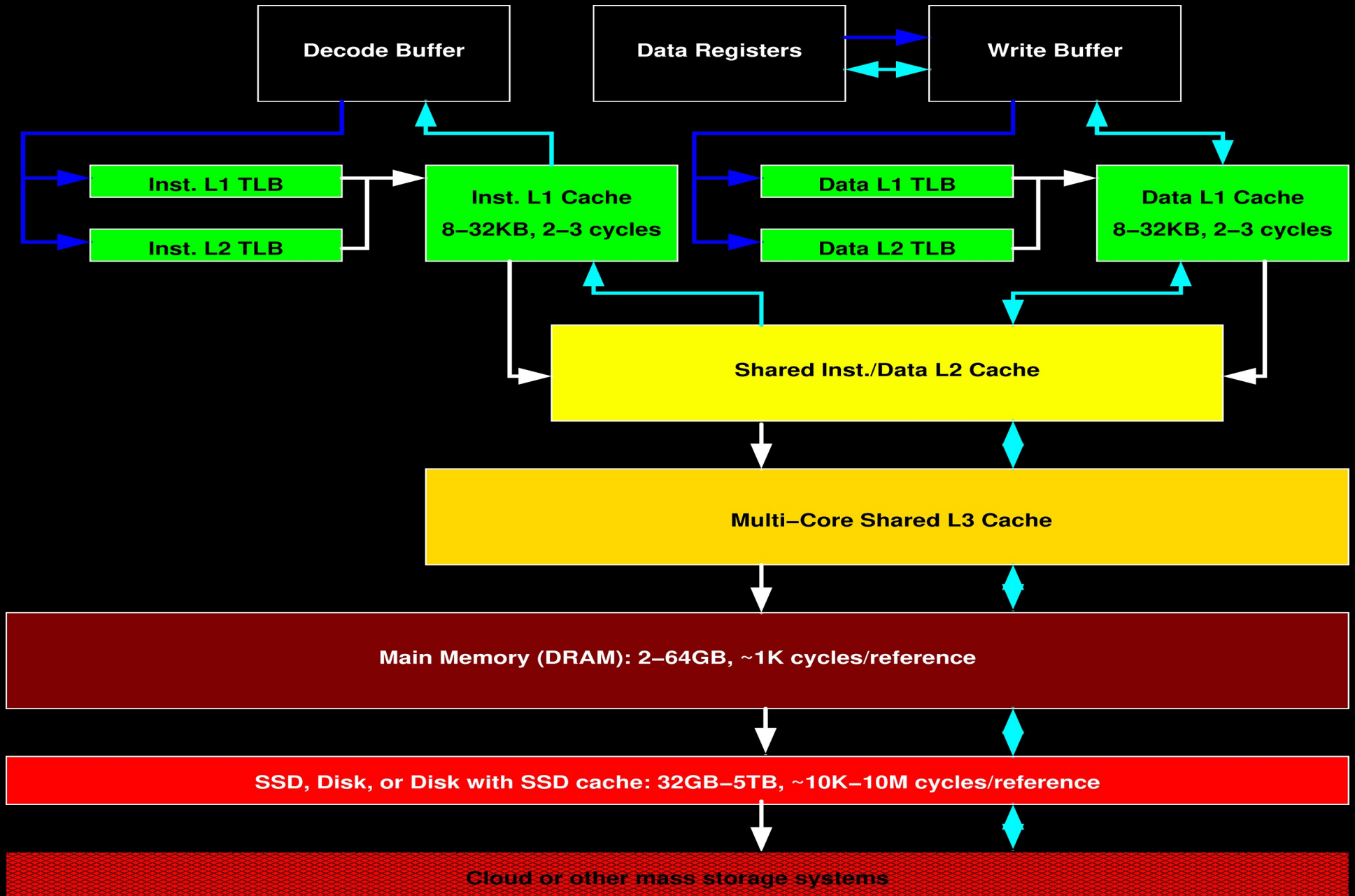
e.g., L1 64kB cache has 1024 64B lines, but L1+L2 TLB might only have 256 entries!

Page Table Use

- Prevents memory fragmentation
- Allows per-page access protection (e.g., **rwX**)
- Don't need to have everything in main memory!
 - Pages can **not yet exist**
 - Pages can **be shared between processes**
 - Pages can **exist on disk**
 - Pages can **exist in a networked machine**
- Pages can be **slow to access from elsewhere**

Page Table Benefits

- Pages can **not yet exist**
 - Stack, heap, and space between
- Pages can **be shared between processes**
 - **DLLs: Dynamic Link Libraries**
 - Inter-process communication
- Pages can **exist on disk**
 - Bigger than main memory
 - **Fault** in stuff as needed, **mapped file I/O**
- Pages can **exist in a networked machine**
 - **DSM: Distributed Shared Memory**



What we want, what we have

- What we want:
 - Unlimited memory space
 - Fast, constant, access time
(UMA: Uniform Memory Access)
- What we have:
 - Memories are getting bigger
 - Growing complexity memory hierarchy
 - Temporal and spatial locality issues
(NUMA: Non-Uniform Memory Access)

Verilog Implementations?

- A cache or TLB is a memory with the usual address decode logic, but:
 - Address used is hash(memory_address)
 - Each memory cell contains a set with...
Cache: (tag, data, dirty, valid, ...)/line
TLB: (tag, physical_address, status)
 - Tag match and replacement algorithm
 - Partial read/write of data field
- State machine sequences operations
- Can be pipelined (even out of order)