Argonne Training Program on Extreme-Scale Computing

ATPESC 2018

Russ Joseph
Associate Professor – Northwestern University

Q Center, St. Charles, IL (USA)
July 30, 2018

Evolution of Multicore Architectures

- Past decade has seen tremendous evolution in multicore designs
  - Increasing sophistication of component cores
  - Numerous ways to compose cores, caches, interconnect
- How can you benefit as a programmer?
  - System architecture has many complex interactions with application
  - Difficult to predict performance
  - Knowledge of the microarchitecture can be useful to programmer!
- Focus on “basic” homogeneous elements
  - No GPU
  - No Vectorization

Bottom Up

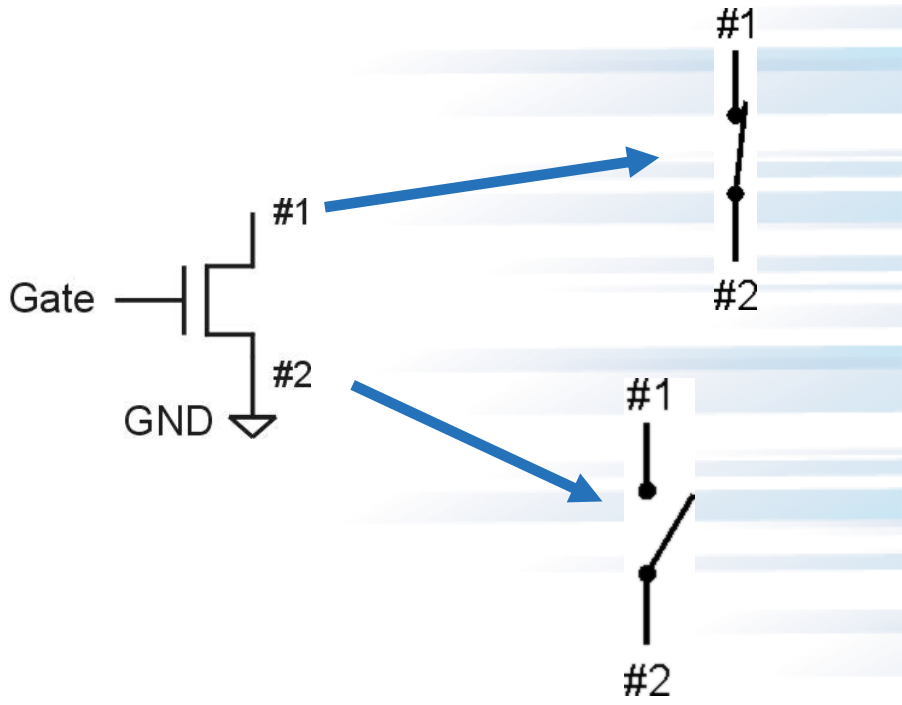
- Lean heavily on system stack concept
  - Layers built on top of layers
  - For this talk start with transistors and work our way up
  - Thinking about transistors…not necessary for day to day activity
- Why start with transistors?
  - Gives you great insight and appreciation for why things are the way they are
  - Helps you to understand trends

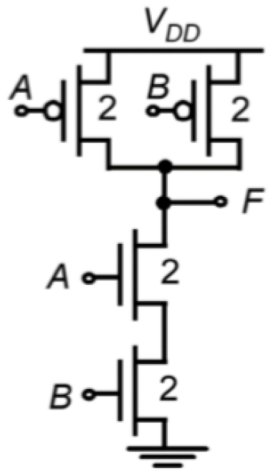
EXASCALE COMPUTING PROJECT

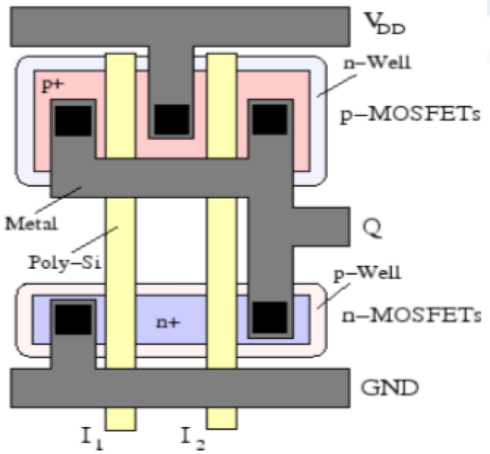
Building With Transistors

Transistor: Switch Model

- Humble transistor: bedrock of our digital world
- Field Effect Transistors (FETs)
- CMOS is dominant design paradigm
  - Two FET variants (PMOS and NMOS)
- As digital devices can be thought of as simple switches
- Put transistors together for compute and memory (state)

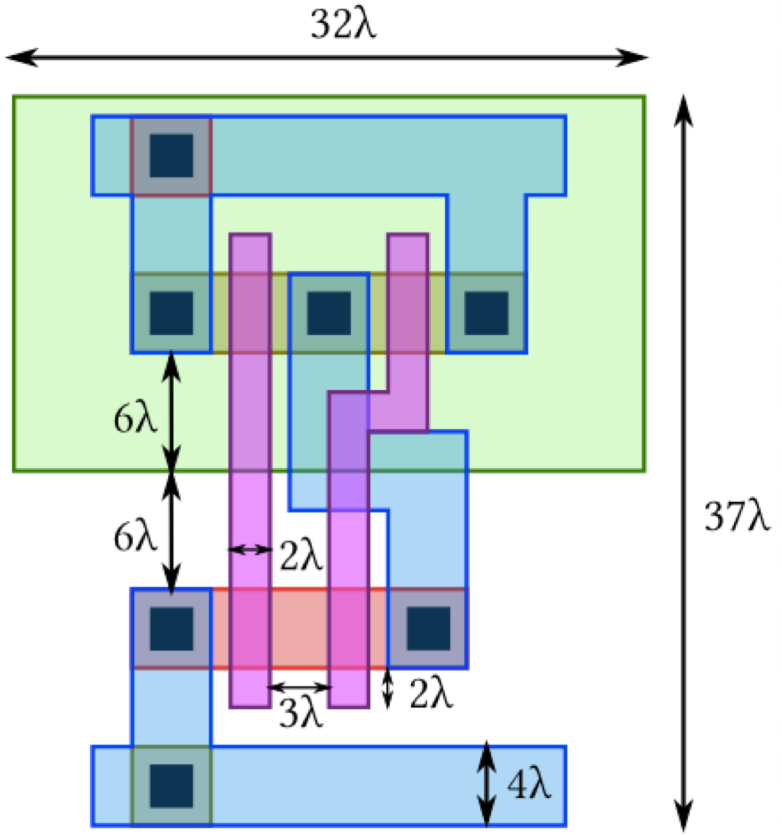


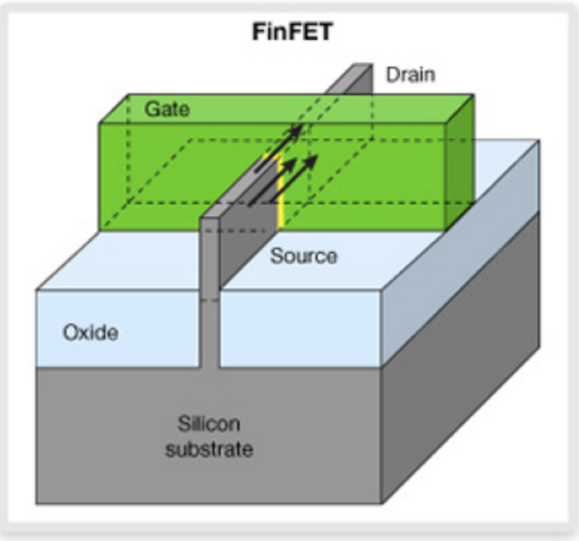




Transistors: Physical Properties

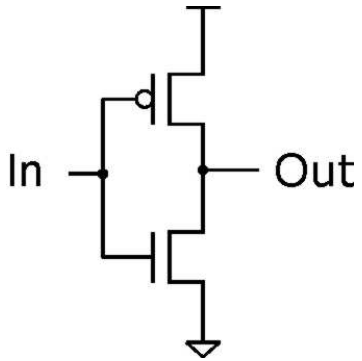
- Physical devices with measurable properties and consequences:
  - Dimensions => Die Area
  - Switching Speed => Latency
  - Power => Battery, Utility Costs, Temperature
- Circuit designer can optimize transistor parameters to make tradeoffs
  - For example, can pick transistors larger to improve speed at some cost in area and power
  - Apply different tradeoffs throughout the system
- Relevance for us: Physical constraints force different design choices

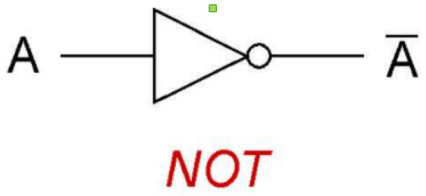


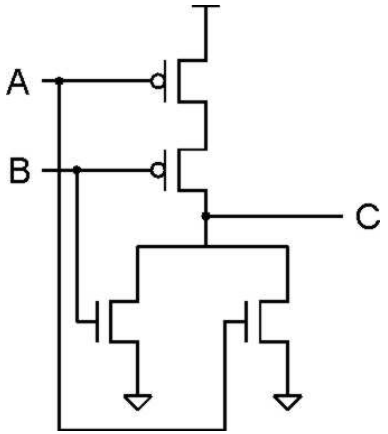


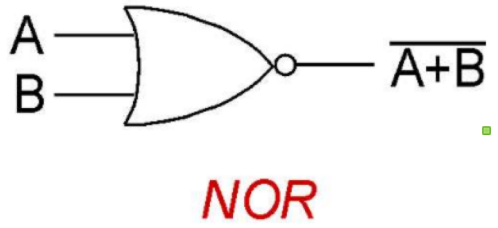
Transistors : Logic Building Blocks

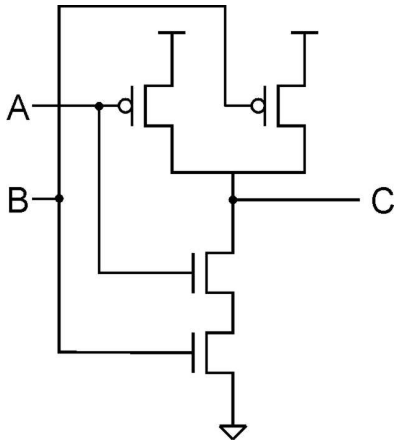
- Build basic logic gates out of transistors:
  - NOT, NAND, NOR, AND, OR etc
- Compose logic gates to compute anything!

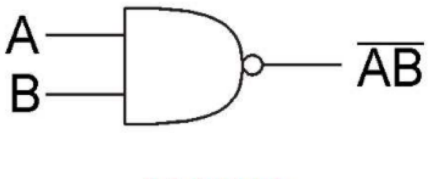




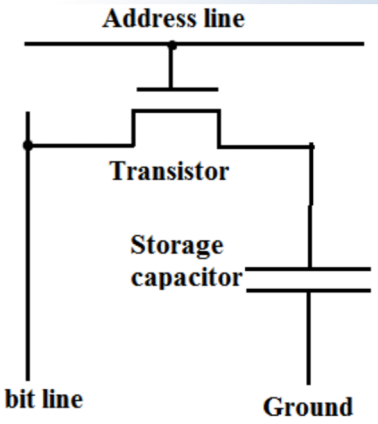




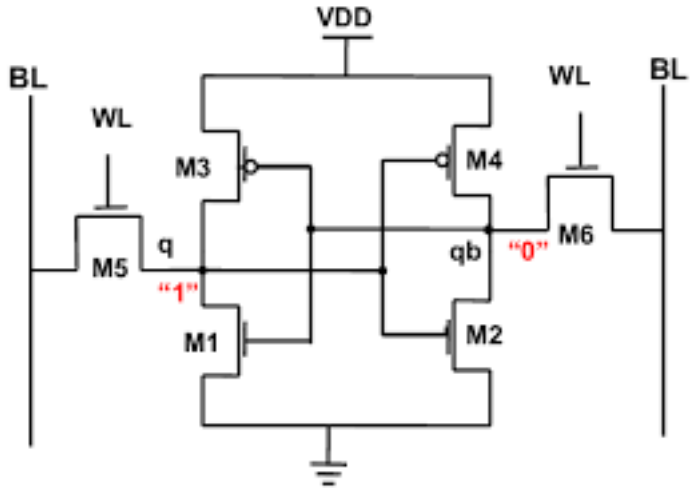




Transistor Building Blocks : State



- Use transistors as storage elements
  - Retain state (hold data)
  - 6-T Design for Static RAM (SRAM)
  - 1-T Design for Dynamic RAM (DRAM)
- Many tradeoffs between latency, power, density
  - Leads to different choice of state elements throughout system
- Wires have delay, too
  - Dominant latency component for caches



EXASCALE COMPUTING PROJECT

Inside The Core

Simple CPU

- Idea: Implement CPU with logic and state building blocks
- Obvious way to do this is sequential
- Bare minimum hardware resources = economical
- Fits naturally with programmer's view

Simple Pipeline

- Idea: Overlap steps in instruction execution
  - Known as a scalar pipeline
- Works because many of the steps are independent
  - May stall when there are dependencies
- Increases throughput – no impact on latency
- Relatively low hardware demands over simple cpu

Superscalar Pipeline

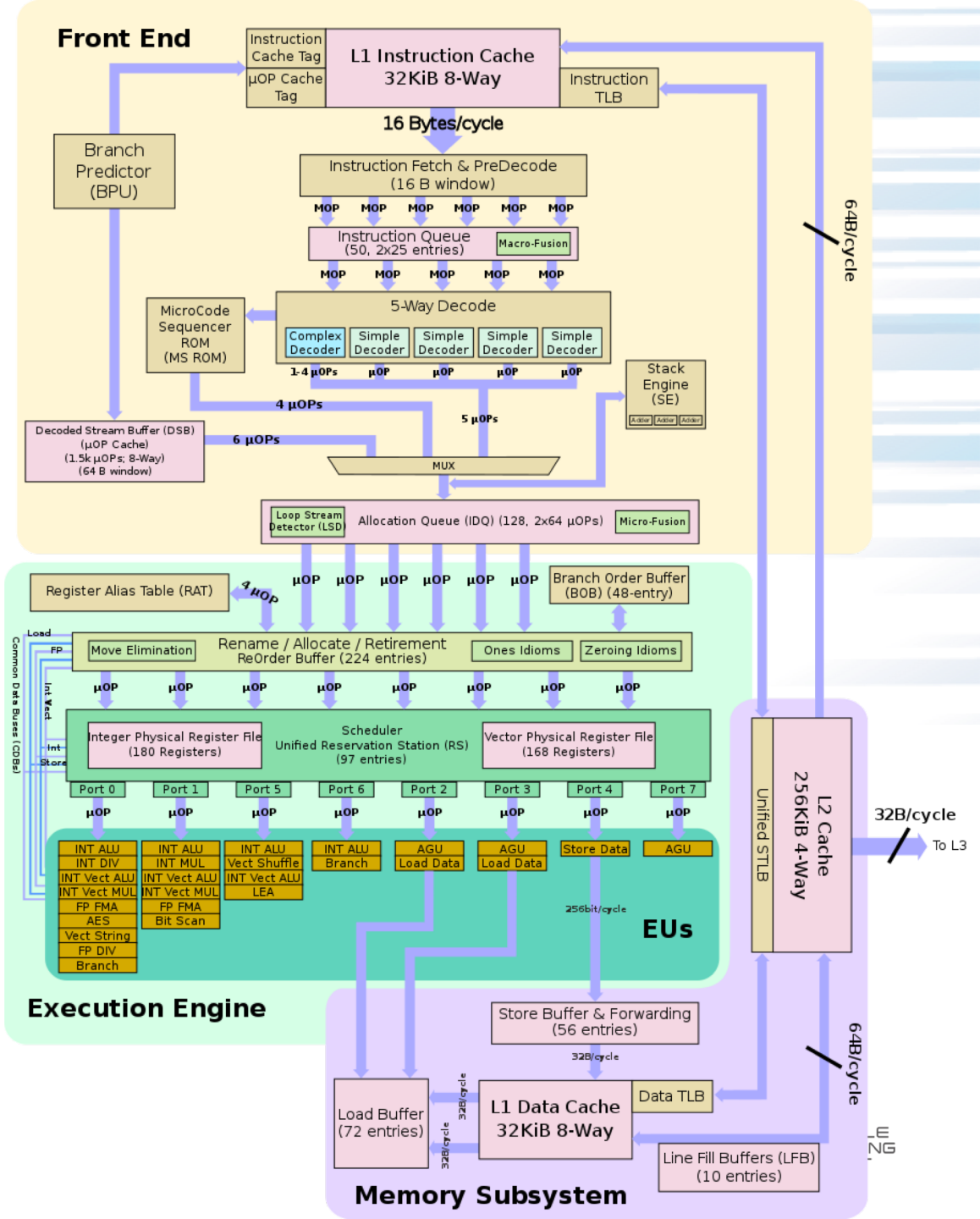
- Idea: Some of these instructions are independent and can be executed in parallel
  - Known as superscalar execution
- Always limited by dependencies between instructions
- Demands more hardware resources
  - Some structures scalar linearly (e.g. execution units)
  - Some quadratic (e.g. dependency check logic)

Out-of-Order Pipeline

- Idea: Detect dependencies dynamically and schedule around them
  - Known as out-of-order execution
- Can schedule execution in any order that maximizes performance
- Hides long latency operations (e.g. some cache misses, long FP operations)
- Expensive hardware requirements
  - Queues and buffers hold waiting instructions
  - Lots of temporary storage for in-flight operations
  - Complicated logic to detect dependencies and schedule

Modern Core

- Pipeline is both deep and wide
- Huge buffers and queues
- Aggressive support for speculation



Hyperthreading

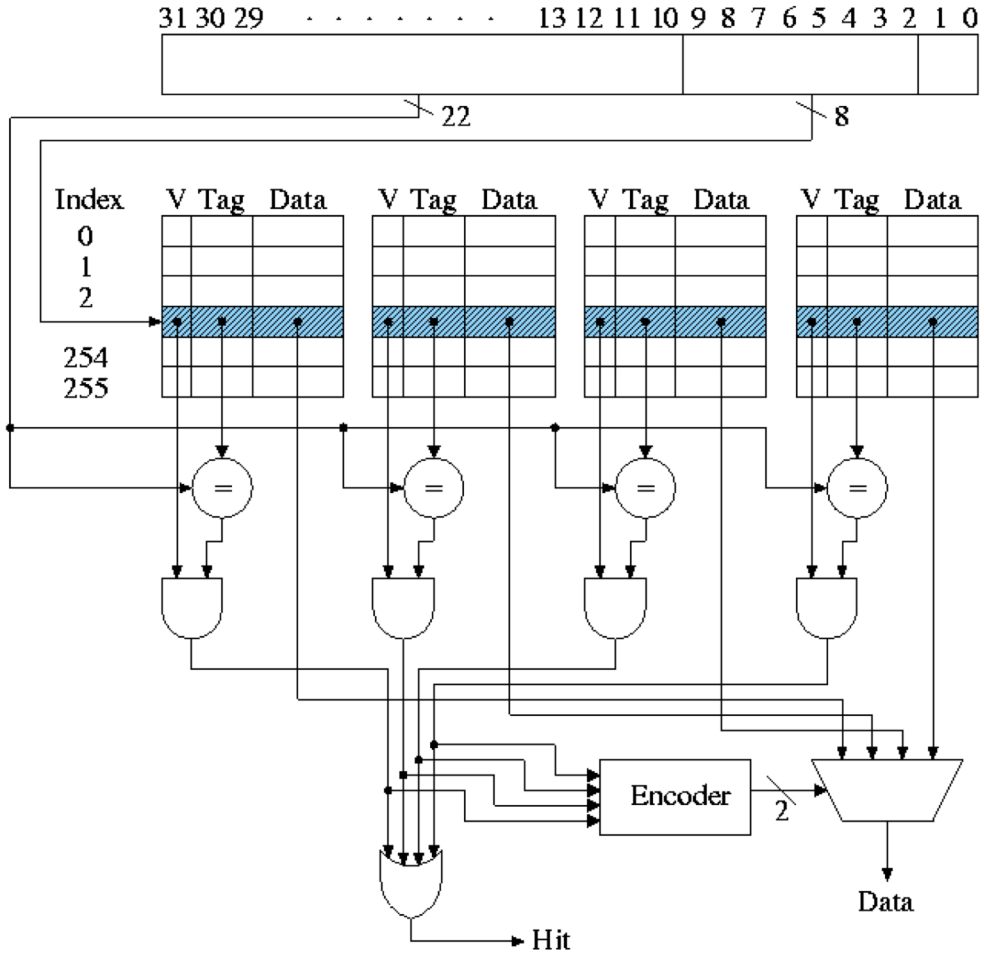
- Idea: Threads can share pipeline resources and execute simultaneously
  - Known originally as simultaneous multithreading (SMT)
- Multiple logical cores map to same physical core
- Works because many core resources are under utilized
- Slows down each thread by some amount but improves throughput
- Modest hardware cost over non-HT

EXASCALE COMPUTING PROJECT

Cache Organization

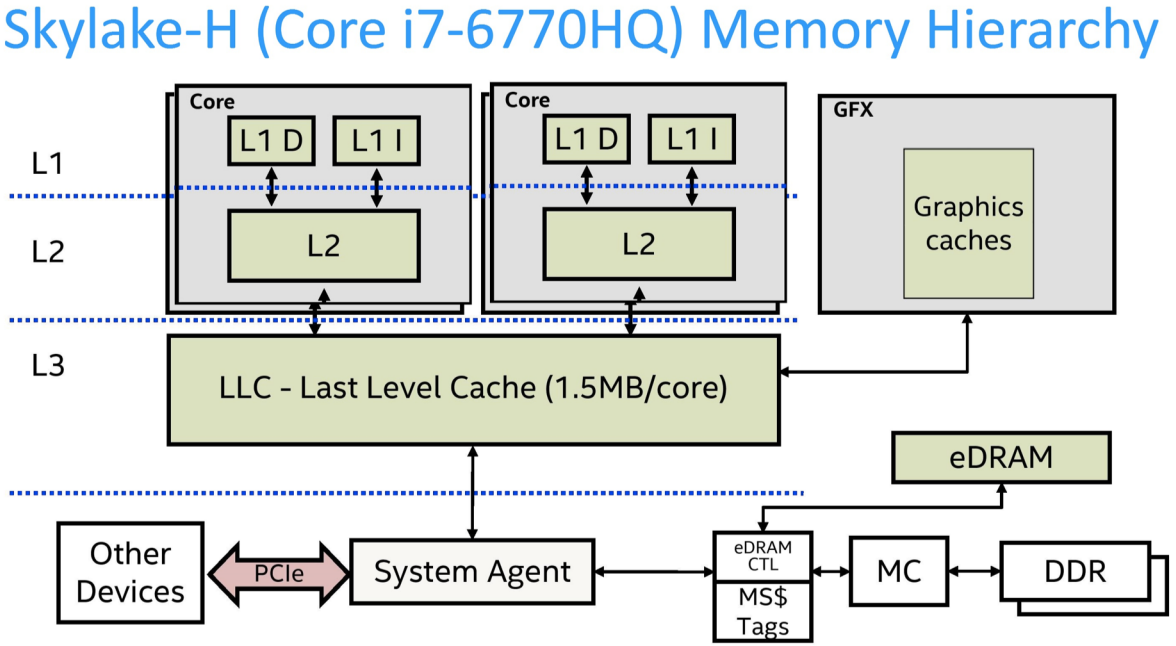
Caches

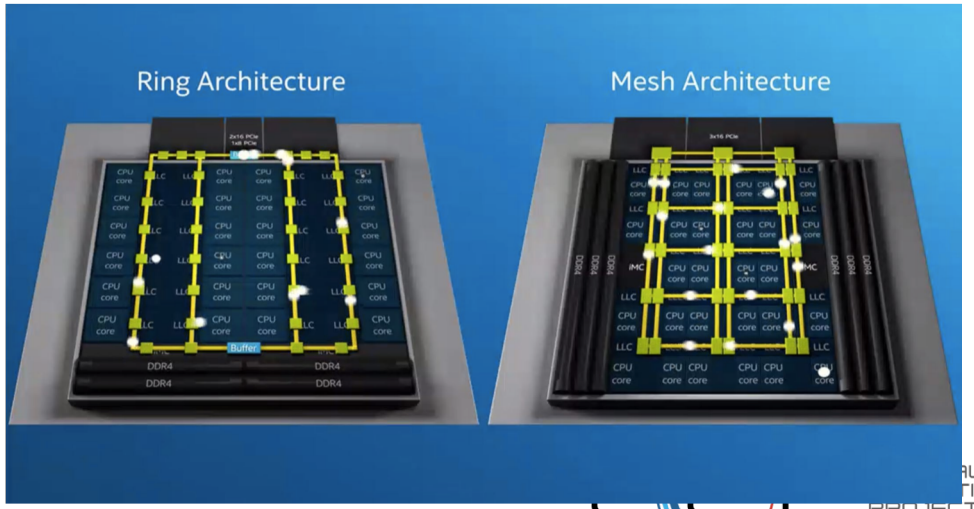
- Fast local storage structures which hold instructions/data
- Most caches use SRAM (most commonly 6-transistor cells)
- Primary benefits:
  - Reduce average memory access time
  - Reduce interconnect traffic
  - Reduce contention on memory
- Fundamental tradeoff in cache capacity (size) and access time
  - Recall: Importance of wire delay



Cache Hierarchy

- Most program data does not fit in a low access latency cache
  - Quick access also implies small size
- Build memory hierarchy with increasingly larger but slower caches
  - Choose associativity and capacity of each level
- In multi-core systems we have additional choices:
  - What parts of the hierarchy are shared?
  - What type of interconnect?



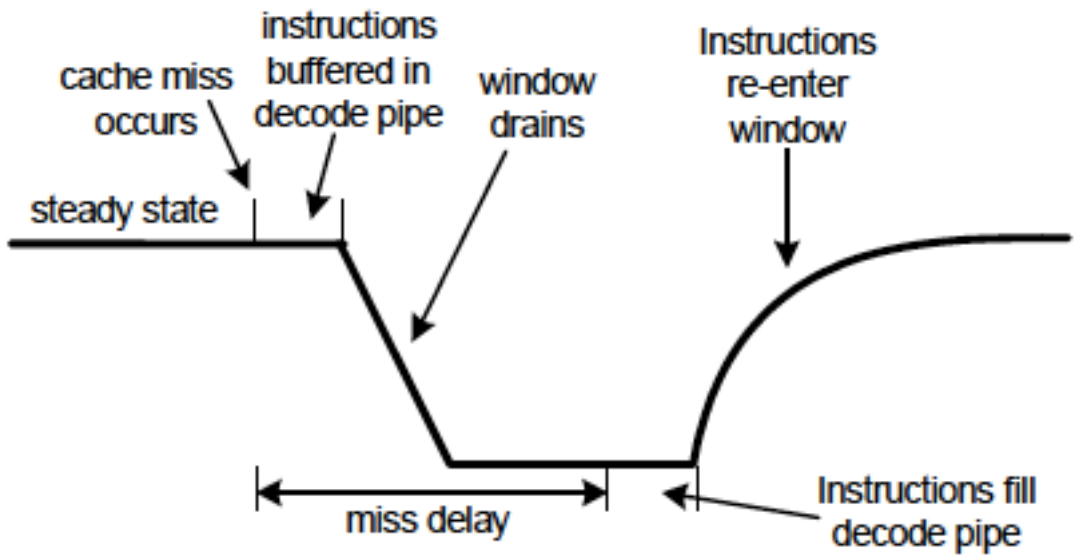


Main Memory

- Contemporary systems use DRAM main memory
  - Very slow…at least 200 cycles per access
  - Off-chip (usually) and uses dense but slow 1-T cells
- Memory architecture has been a very vibrant area of architecture research over last decade
  - For some important workloads, there is no hope of fitting everything within even a very large cache
  - Many different ways to organize DRAM and schedule requests
  - Don’t have time to do this justice here

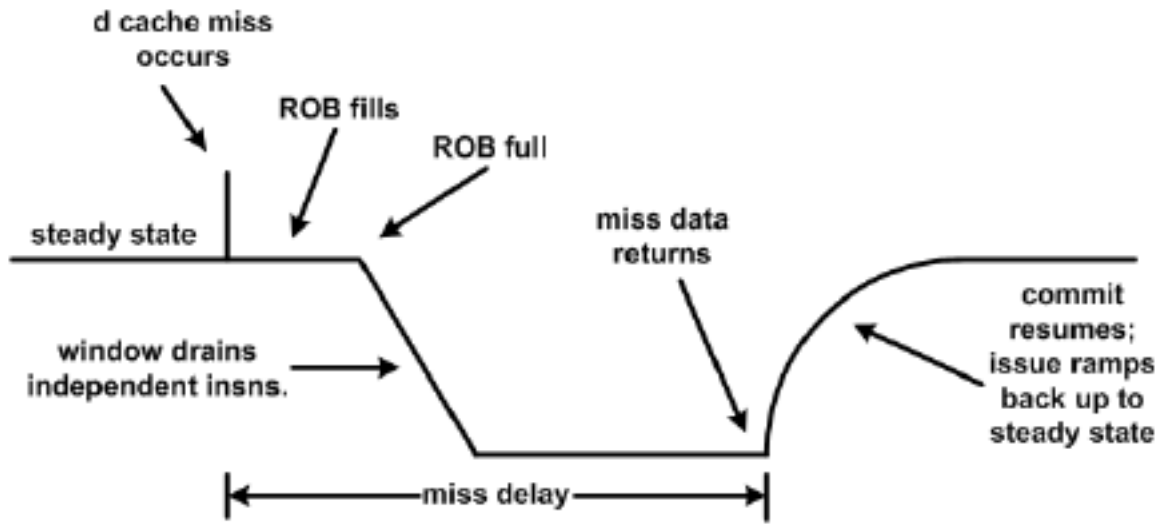
I-Cache: What happens on a miss?

- Instruction stream is disrupted…no instructions to feed pipeline
- This is true for any I-Cache miss (even if it hits in next level)



D-Cache: What happens on a miss?

- If miss event is short…maybe nothing
  - OOO easily hides the penalty
- If miss event is long…latency cannot be hidden
  - Buffers and queues fill up
  - Cannot make progress until miss serviced

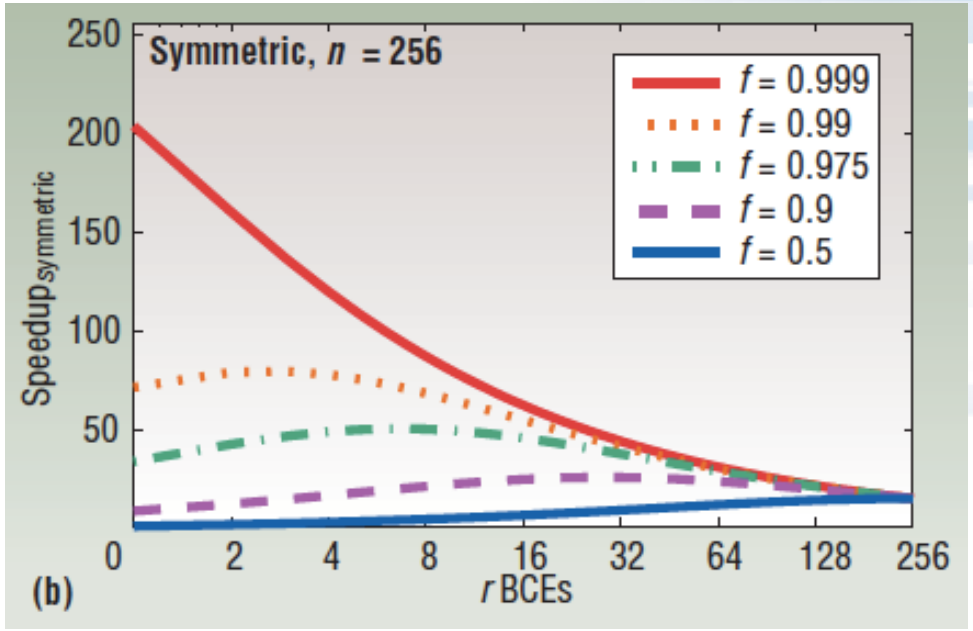


EXASCALE COMPUTING PROJECT

Programming for Multicores

Limits to Parallelism

- Ideally, we'd like linear speedup
- But because of Amdahl's Law we won't get there
  - $p=T(\alpha+(1-\alpha)/p)$
  - Where α represents the serial part of the computation
- Classic sources of inefficiency:
  - Load imbalance
  - Synchronization
  - Communication

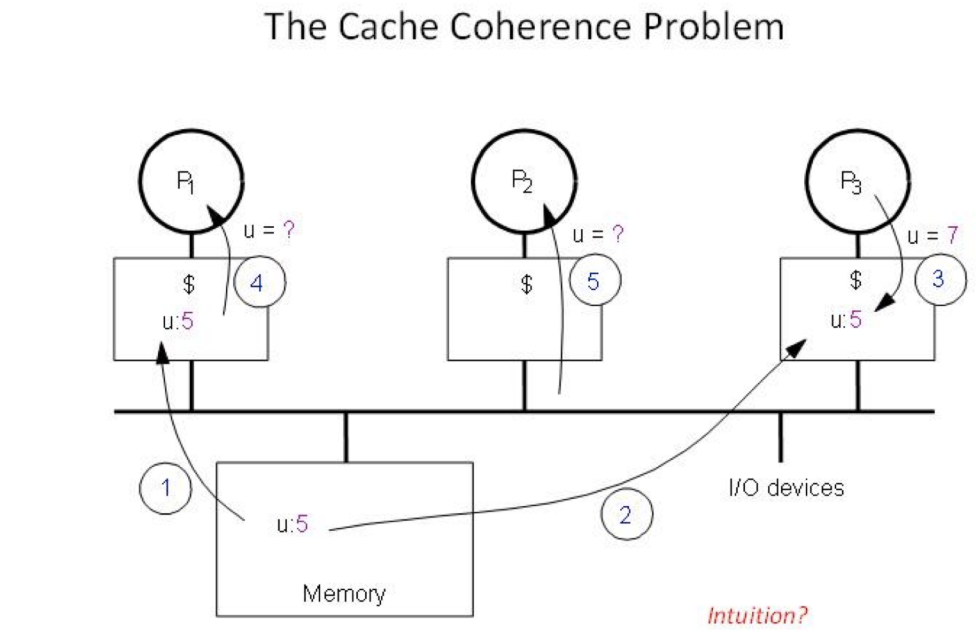


Types of Misses

- Most communication in the system comes through memory hierarchy
- Revisit cache misses:
  - Cold / Compulsory – Data has not been accessed yet
  - Conflict – Insufficient associativity
  - Capacity – Data does not fit [Sharing Resources]
  - Coherence – Data is being shared…system forces misses to maintain correct semantics (RAW) [Sharing Data]
- In parallel workloads, sharing is critical

Cache Coherence

- Blocks are replicated throughout system to facilitate local caching of data
- But when there are writes to data the system needs to maintain correct version
- Track state of cache blocks
- Introduce invalidations (forcibly evict blocks) when writes occur
- No programmer intervention required for correctness



Coherence Misses

- Misses introduced by data sharing and enforcement of coherence
  - Cache block shared across P1 and P2
  - P1 does write to block => system invalidates P2's copy
  - P2 misses on next read of block
- True Sharing: P1 writes to data word that P2 actually uses
- False Sharing: P1 writes to one data word; P2 reads from different word

Assumptions

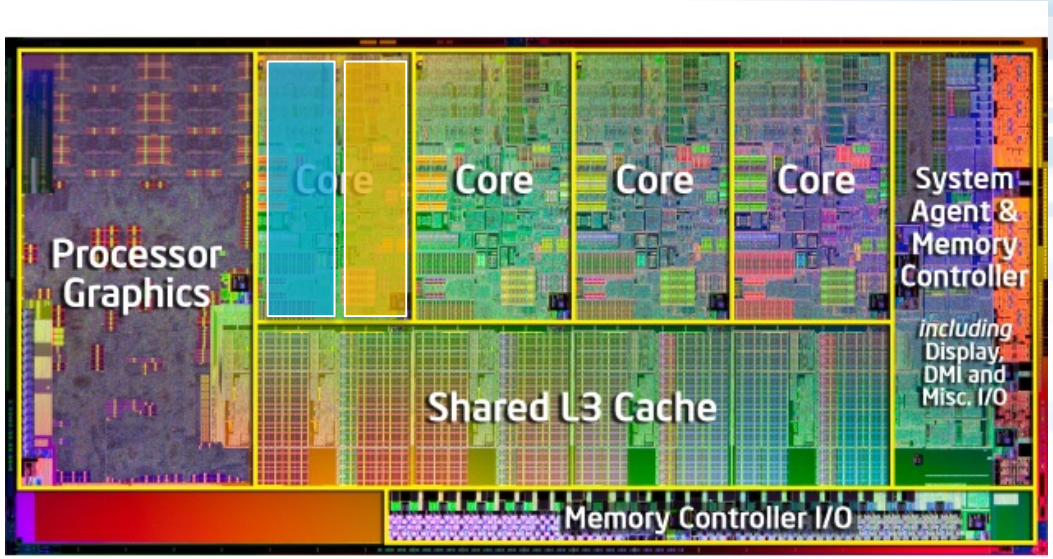
- Have already applied all the classic cache optimizations
  - Loop interchange
  - Loop fusion
  - Blocking
- Minimize load imbalance, apply appropriate task assignment, usually:
  - Static: When tasks are homogeneous and work is predictable
  - Dynamic: When there is variance and uncertainty

Maximizing Performance

- Start with basic implementation: Don’t get too cute!
- Understand key properties: Definitely profile!
  - For each task: Latency, Data
  - Relationship between tasks
- Try to figure out bottlenecks
- Will be an iterative process
- Obvious knobs you can turn:
  - Thread count
  - Mapping
  - Data placement / arrangement

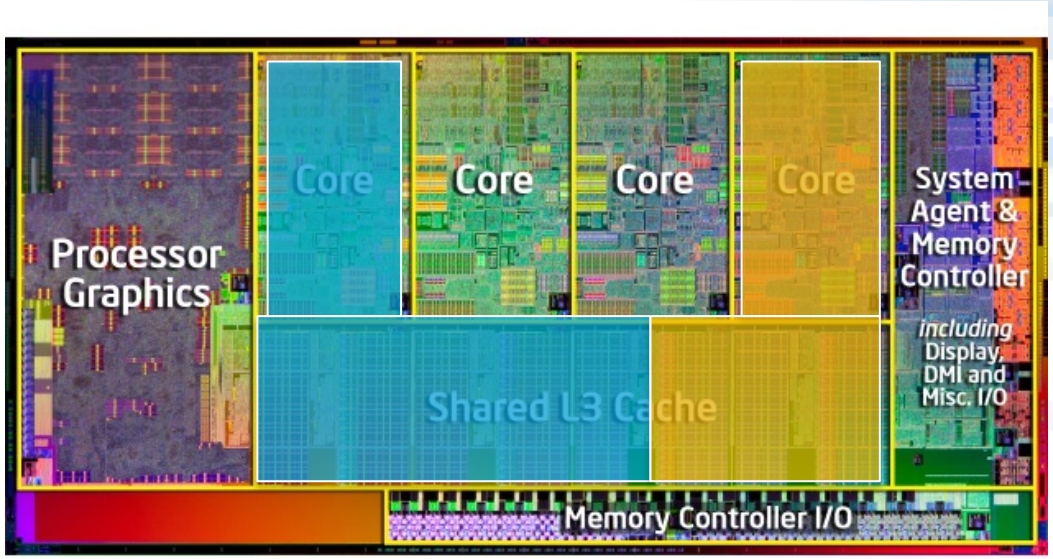
Fits Within Private Caches (L1/L2)

- Map threads to same physical core
- Separate logical cores with Hyperthreading
- Low miss rate for private cache
- If data parallel will benefit from common instruction working set
  - Good instruction cache behavior



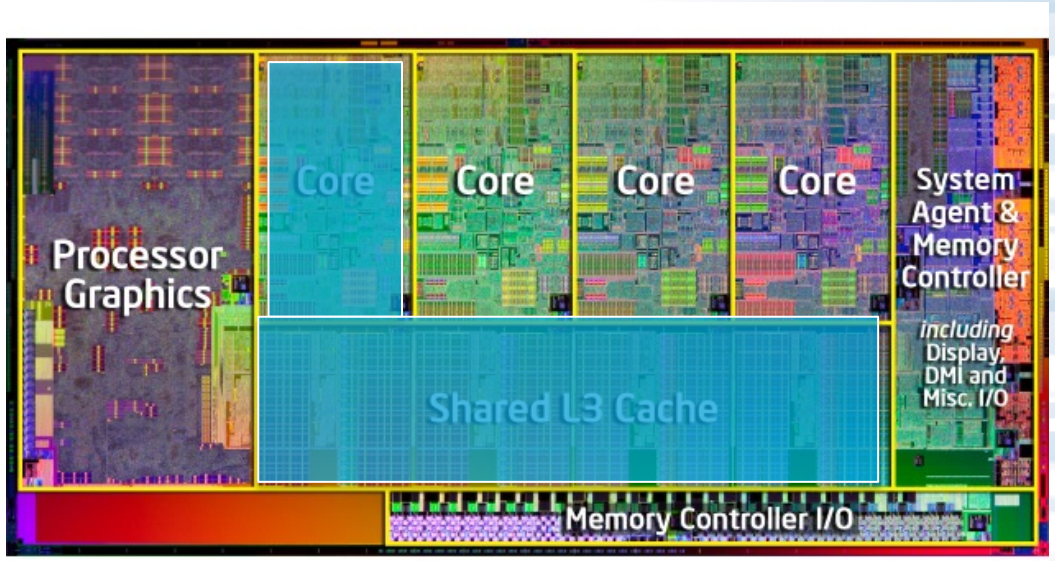
Fits Within Same LLC

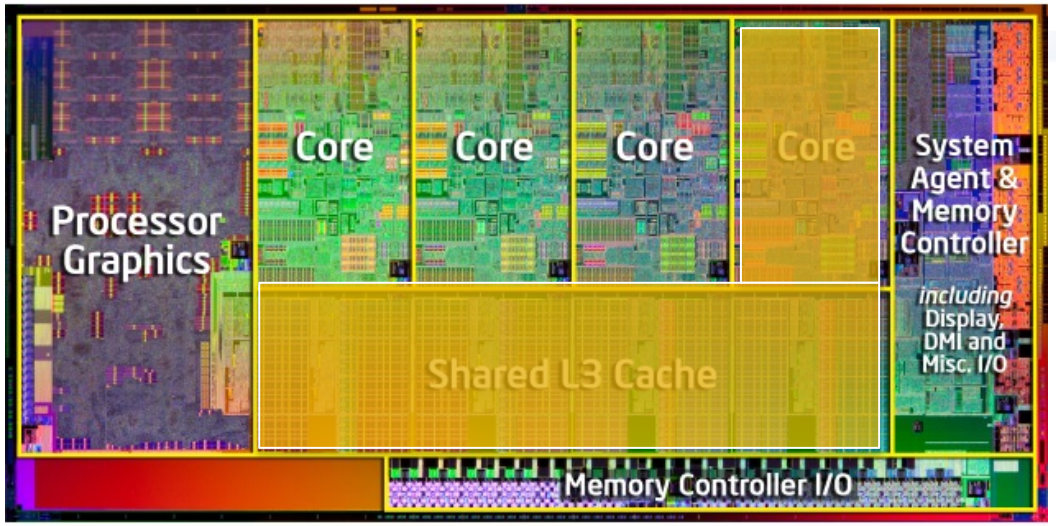
- Map threads to different physical cores within same chip
- Minimize off-chip accesses
- May still pay penalty for distant on-chip access



Does NOT Fit Within LLC

- Map threads to different chips/socket
- Do your best to make most of LLC on each socket





Managing Communication Costs

- Some aspects of communication are intrinsic to the algorithm
  - Example: Data sharing in matrix multiplication
  - Try to reduce the associated costs (e.g. lower latency of sharing)
- Other aspects of communication are introduced by system
  - Often caused by a mis-match between system architecture and implementation
  - Try to eliminate these when we can

Managing Communication: Locality Awareness

- Identify threads that share objects / critical sections
  - Coherence misses (true sharing) with data and synchronization variables
  - Associated communication is intrinsic to algorithm (cannot avoid)
- Place threads as close as you can
  - But still try to respect capacity issues
  - Tradeoff of capacity misses versus coherence misses

Managing Communication: False Sharing

- False sharing occurs when threads have read/write access to same cache block without exchanging data
- Architecture dependent and leads to unexpected performance issues
- Artifact of the system architecture
  - Coherence messages try to maintain correct RAW semantics
  - Block size is too coarse => mis-matched with structure of data
- Need to tune data layout of arrays/objects to suit block size

Thank you!