ECE411: Computer Architecture

Because I Didn't Go To Lecture (-ish)

Pradyun Narkadamilli

Contents

1 Midterm 1

1.1 Background and General Architecture

- Role of computer architecture: falls between software (OS and above) and hardware (circuitry and below)
  - Encompasses ISA and Computer System Organization
- Clock Frequency used to be good indicator of performance (still is)
  - Had to stop increasing because it slowed down semiconductor scaling and had excessive power draw
- **Moore's Law**: the number of transistors in a dense integrated circuit doubles every two years
  - 1965 to 1980 it doubled every year, after 1980 is slowed to every two years, and since 2010 there has been further slowdown
- **Dennard's Scaling**: When transistors get smaller, their power density remains constant so that power consumption is still proportionate to area
  - Every technology generation, transistor dimensions were 0.7xed while area halved
  - Voltage reduced by 30%, Circuit delay down by 30%, Power halved, Capacitance down by 30%
  - Comes out to **2.8x chip capability at same power**
  - *Ended in 2005-2007*
- After Dennard's scaling ended, transistor feature size 0.7x per generation, w/ 1.4x more chip capability at same power
- How to increase perf if we don't keep cranking clock frequency?
  - Increase core count, specialize hardware, add GPU, make accelerators (TPU, AI chips, etc.)
- Memory-Compute gap is increasing (storage not speeding up at same rate that CPU perf is)
- **Von Neumann Architecture**
  - CPU contains a *control unit* and an ALU - it communicates with a *memory unit*
  - Computer communicates with Input Devices and Output Devices
  - Instructions are executed in a **fetch-decode-execute-store** cycle
    - Corresponds with RAM, CPU, ALU, RAM, where the 4th store is potentially optional
  - Contains regfile to hold data referenced by instructions, has a program counter to retain where in the code it is, and has random-access to memory with address to select location
- Memory byte ordering
  - *Big endian* - highest memory address contains right-most byte
  - *Little Endian* - lowest memory address contains right-most byte
- **Single Cycle Design** (e.g MIPS) - fetch/decode/execute all happens in *one* clock cycle. Thus, cycle time is constrained by the critical path, i.e the ***longest instruction***
  - These designs tend to be very inefficiently clocked and have a high hardware cost (potentially duplicated functional units since they cannot be shared during an instruction)
  - Very simple/easy to understand (look no further than CS 233)
- **Multi Cycle Design** (any modern processor) - each instruction is separated into multiple *steps*, and the clock is timed to the slowest *step*.
  - Allows functional units to be used more than once an instruction by muxing inputs with a control unit

- Requires more state registers, more muxes, and more complex control FSMs
- Due to state register overhead and bloat from the critical path, the clock of a multicycle is not perfectly $\frac{\text{single cycle clock}}{\text{\# of steps}}$
- *Goal*: want to balance amount of work done each cycle (no one step should be obscenely long)

1.2 ISA

- Purpose/Function of ISA?
  - Serves as an interface between SW/HW, provides mechanism for software to control hardware
- **ISA Considerations**
  - Operand Location: registers, memory, stack, accumulator
  - Number of operands
  - How is operand location specified? (register, immediate, indirect, etc.)
  - Operand size/type
  - Supported operations
- How to pick number of registers?
  - Fewer registers requires fewer address bits in ISA, less hardware, faster access (less fanout), faster context switch (when saving all registers)
  - Larger registers result in fewer load/stores from memory, and can make operation parallelism easier
- Why are GPRs so popular in newer architectures?
  - Much faster than cache/memory, values are available immediately
  - Convenient for variable storage, especially since compiler can choose to use fastcall-esque register assignments
  - Deterministic (no such thing as a "miss")
- Cons of GPR
  - context switching (think of 391)
  - No such thing as "register address" - anything using pointers can't be mapped to registers
  - fixed size (can't store structs)
  - Compiler needs to manage them, and there aren't many!
  - *Operation Types*
    - Arithmetic, Data Transfer, Control (`br/jmp`), System (`os call`, `vm`), Float, Decimal, String, Graphics
- *You need to have bits that specify what addressing mode you're using bozo*

1.2.1 Variable vs Fixed Instruction Length

- **Variable Length Instructions** - May vary from 1-17 bytes long (e.g x86)
  - Requires multi-step, complex fetch-decode [**ANNOYING FOR DESIGNER**]
  - More accurately captures logic of program, leading to smaller program size [**GOOD FOR MEMORY**]
    - Consequently, less disk storage, less DRAM at runtime, less memory bandwidth, better cache efficiency
- **Fixed Length Instructions** - All instructions are 4 bytes long (e.g MIPS, PowerPC, most RISCs)
  - Easy fetch/decode, simplifies pipelining and parallelism [**GREAT FOR DESIGNER**]
  - Can lead to more bloated programs [**BAD FOR MEMORY**]
    - Invert above reasoning

1.2.2 ISA Classes

1. Stack `push` loads memory into first register, moves other registers down, and `pop` does reverse. An `add` will combine first two regs, move the rest up.
   - 0-address instructions, both operands are implicit from stack ordering
   - *Instruction Set*: `add`, `sub`, `mult`, `div`, `pop`, `push`
   - **Pros**: good code density, low hardware overhead, easy to write compiler
   - **Cons**: stack structure becomes bottleneck, not much parallelism or pipelining, data not always at top of stack (might need to `swap`), hard to write optimization in compiler
2. Accumulator Only one register called `accumulator`.
   - 1-address operations since accumulator is an implicit operand.
   - *Instruction Set*: `add`, `sub`, `mult`, `div`, `load`, `store`
   - **Pros**: *very* low hardware overhead, very easy to design/understand
   - **Cons**: accumulator is the bottleneck, not much parallelism or pipelining, high mem traffic
3. Reg-Mem Arithmetic instructions can use data in GPR (general purpose registers) and/or memory.
   - 2-address operations, one operand is an implicit dest
   - Think of x86 ISA
   - **Pros**: some data can be accessed w/o a load, instructions easy to encode, good code density
   - **Cons**: operands have asymetric load/store time, slower operand is bottleneck
4. Load-Store (Reg-Reg) Arithmetic instructions are only performed on GPR, load/stores are used to pull data out of memory.
   - 3-operand operations, dest is explicit
   - Think of RISC-V ISA
   - **Pros**: simple fixed-length instructions, instructions take similar num of cycles, pretty easy to pipeline/make superscalar
   - **Cons**: higher instruction count (code bloat), not all instructions *need* three operands, you need a good compiler

1.3 Performance

- *Latency* - How long does it take to execute a task?
  - denoted as T_{exe}
- *Throughput* - How many jobs can the machine finish in a minute?
  - potential metrics: millions instructions per second (MIPS), millions floating ops per second (MFLOPS), TOPS
  - if a machine can finish 90 jobs in a minute, but they all take a minute and are performed in parallel, it has better throughput than a machine that finishes 1 job every 15 seconds, and its latency is worse.
- CPU Execution time: $\frac{instructions}{program}\frac{cycles}{instruction}\frac{seconds}{cycle}$
  - I/P is determined by software side (programmers, algos, compilers) and ISA
  - C/I (*CPI*) is determined by microarchitecture/system architecture
  - S/I is determined by microarchitecture and circuit characteristics
- *CISC* minimizes *I/P* whereas *RISC* minimizes *C/I*
- Potential Benchmarks

  - Real Programs, Kernels, Toy Benchmarks, Synthetic Benchmarks, Benchmark Suites (ex: SPEC for scientific/engineering/GP)

- Reporting performance is difficult - simply averaging everything together can weight benchmarks inequally, and any arithmetic mean can cause "reference machine" biases.
  - **Solution**: use the geometric mean as follows, considering an example case of T_i for n different benchmarks:

$$\sqrt[n]{\prod_{i=1}^{n} \text{normalized execution time i}}$$

  - Normalize execution times by dividing each one by the corresponding runtime on a selected reference machine
  - In the event that weights are *provided* for different benchmarks, it is permissible to use a weighted average instead. To find *speedup*, you will want to use the

- **Amdahl's Law** (diminishing returns) - make the common case faster. Assume the following where f is the proportion of the whole task involving the optimized task, and P is the proportion that the optimized task was sped up by.

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{P}}$$

  - *ex*: if FLOPS are 10% of a program and are optimized to 2x faster, then $f = 0.1$ and $P = 2$

1.3.1 Power

- performance/Watt - perf achievable with same cooling budget
  - Power (energy/time) poses constraints on how much cooling is necessary, the noise margin, how ground needs to be wired, etc.
- performance/Joule - per achievable with same energy source (ex: battery)
  - Reciprocal of energy-delay product (ED)
- The energy required for a task is generally considered its 'true cost', thus energy is the 'ultimate metric'
- Common sense, but $E = P \cdot t_{exe}$
- A general estimate of power, where C_L is load capacitance, and α is probability of a low-high transition on a given clock:

$$P = \alpha_{0\to1} f(C_L V_{DD}^2 + t_{sc} V_{DD} I_{peak}) + V_{DD} I_{\text{leakage}}$$

  - Dynamic power and short-circuit power are going down, while leakage power continues to increase

1.3.2 How to evaluate an ISA?

- *Design-Time Metrics* - [feasibility, time, cost] of implementation, software support
- *Static Metrics* - how many bytes in memory?
- *Dynamic Metrics* - how many instructions, how many bytes fetched for program, CPI
- **Best Metric**: program execution time

1.4 Pipelining

- Basic concept - keep as many stages populated as possible every clock cycle. Concurrency good!
- *Balanced Pipeline* - time btwn instructions (pipelined) = $\frac{\text{time between instructions (non-pipelined)}}{\text{\# of stages}}$
  - maximizes speedup from pipelining
- RISC-V is feasibly pipelined since it
  - has fixed-size instructions, so fetch-decode can be done in a single cycle
  - instruction formats can be decoded and registers read in one step
  - load/store addressing - can calculate address in 3rd stage, access memory in 4th
- **RISC-V Pipeline Stages**
  - *IF* - Instruction Fetch from instruction memory/cache
  - *ID* - Instruction Decode (pull out regfile values)
  - *EX* - execute (ALU funstuffs)
  - *MEM* - potential read/write from register
  - *WB* - write back into regfile
  - in below notation, `[STAGE1]/[STAGE2]` denotes the register sitting between those two stages
- *Maximum Speedup* calculated as $\frac{\text{instruction latency}}{\text{pipeline stage latency}}$
  - Your *throughput* is significantly higher (hence the speedup!)
  - Latency may be longer for some instructions, because they have to go through extra pipeline stages (think of `add`)
- *Steady State* - all pipeline stages are fully utilized

1. Pipeline Hazards
   - *Structure Hazards* - resource is busy (this is why we separate instruction/data memory)
   - *Data Hazard* - need to wait for previous instruction to complete data read/write
   - *Control Hazard* - deciding on control action depends on previous instruction
     - *Branch Hazard* - branch determines flow of control, so we end up with a "bubble" in the pipeline while the branch finishes
   - To insure pipeline execution, we must *detect hazards* then *stall the pipeline*
     - the fetch/decode stages should not progress, and `nop`'s should be inserted into later stages.
     - Stalls reduce performance, but are necessary to guarantee correctness
     - Reordering code to avoid using load results in the very next instruction can help avoid stalls
   - *Load-Use Hazard* - a type of data hazard. Occurs when a mem read is followed by an ALU operation using its result
     - Detected when `ID/EX.rd = IF/ID.rs[1/2]`. Need to stall pipeline in this case.
2. Data Hazards
   - Can be handled in software by adding `nop`, or can stall the pipeline w/ bubbles in hardware
   - Can be avoided with **forwarding** - directly pass ALU output from previous op into ALU input for next one (avoid waiting for the register store)
     - Forwarding can only solve *code organization* data hazards - it *cannot* account for data hazards caused by memory latency
   - Forwarding requires tracking register numbers in the pipeline to detect data hazards
     - EX/MEM.rd = ID/EX.rs1 will forward from EX/MEM register
     - MEM/WB.rd = ID/EX.rs1 will forward from MEM/WB register

  - The above cases should only be handled if rd is *nonzero* and *the instruction performs a reg write*
  - There are also potentially *double data hazards* where `rd` is overwritten *twice* - we want to use the *most recent* result.

3. Branch Hazards

- Branches separate into direct conditional branches (if-else, loops, etc.) and unconditional branches (function calls, goto, return, etc.)
  - Around 80% of branches are *conditional*
- Branch hazards are caused due to the indeterminacy of PC location after the instruction - how to handle?
  - Could stall until PC is known - but branching is so frequent that this would cause a large performance hit. At least it's easy to implement?
  - *Static* branch prediction - always assume branch taken or not taken, and execute successor instructions.
    - Squash/flush subsequent pipeline instructions if the assumption is incorrect
  - *Dynamic* branch prediction - fun stuff!
  - Delay the branch - by placing an instruction directly after the branch instruction to fill the 1-slot delay, we can avoid a stal
- **Filling Branch Delay Slot**
  - Can place an unrelated instruction (no data hazard) logically occuring before the branch operation *after it* in the compiled code. This will allow it to execute in the indeterminate pipeline "slot" right after the branch
  - Can place an instruction from target or fall-through - benefits if branch result matches prediction
  - If compiler can't find useful instruction, we just insert a nop (amounts to a stall)
- **Simple Dynamic Branch Predictor**
  - Create a **branch history table** - each entry represents the state for a different branch prediction FSM
  - After potentially putting the PC through a hash, some of its LSBs are used to index the BHT - every time that index's prediction is used (and proved right or wrong), the relevant table entry's FSM is updated
  - Depending on the predictor, the size of each entry will change
  - The simplest predictor in each entry of the table is 1 bit storing the previous branch decision
  - A slightly more complex predictor is the *2 bit saturating up/down counter*, where you can either strongly or weakly predict taken vs not-taken.
- **Branch Target Buffer** - also called BTB, you use all PC bits for a lookup into table.
  - Each entry contains predicted PC and branch prediction
  - If there is a match in table and predict bits are set, set PC to BTB's predicted PC
    - If prediction wrong, do the same old branch mispredict recovery (flush pipeline, reset PC)
    - The prediction bit is simply the old success or failure of the branch
  - If decode indicates branch w/o BTB match, we can either:
    - look up prediction now, act on it
    - statically predict fallthrough
  - When branch resolves, update predicted PC and prediction bits as necessary
- Branch predictor does well with *conditional* jumps, BTB is very good at *unconditional* jumps (in the sense that you can ignore the branch prediction for unconditionals)
- Indirect jumps change their PC location frequently, and are often used for return instructions - we need a mechanism to know what the return address is

  - Call instructions deterministic - we can keep a stack of PC address when call instruction occurs
  - If return instruction seen, pop off of caller address stack
- *Note*: newer BTB implementations will *not* have a prediction bit, since there is a branch predictor in addition to the BTB. The branch predictor **only predicts a yes/no**, but **the BTB is always where the CPU will pull the 'potential' PC destination**
- *Aliasing* - a new PC value causes reuse of older branch prediction/BTB values that had the same indexing value
  - This is why hash functions are sometimes used on the PC value - to more evenly distribute table/buffer indexing with branch prediction constructs

---

For pipelining, anything under here is *likely* optional for the exam, so no need to obsess over it. Good to know in general though!

- **Branch Correlation** - Idea that if you have a set of clustered branches and know the result of few earlier ones, you might reasonably be able to know the result of later branches in the cluster.
  - *Correlated Branch Predictor* - correlation scheme marked as (M, N) for M shift register bits and N bit counter
    - Bottom of branch PC (potentially hashed) is used to select a table with 2^M entries of N bit counters.
    - An M bit global branch history shift register is used to select which entry in that table to use/update
    - Idea is that you read/update the branch prediction in the case that M previous branches were in the same state that they currently are!
- *Two-Level Branch Predictor* - a generalized correlated branch predictor
  - One shift register called *Branch History Register* tracks the previous N branch outcomes
  - Index into *Pattern History Table* with BHR as an index, each entry in PHT contains a counter FSM (assume 2-bit saturating counter by default)
- *Gshare Branch Predictor* - pretty much same logic as TLBP, but there is one table indexed by `pc xor BHR` instead of an array of tables for different PC values.
- *Tournament Predictor* - also called Hybrid Predictor. Has multiple (suppose 2 in this case) branch predictors all generating predictions, then will have a *meta-predictor* to pick which predictor to use. Meta-Predictor will have its own counter strongly/weakly choosing to use either predictor

1.5 Memory/Caching

- Memory is not just a black box
  - Pyramid of memory from top-down: Registers, On-Chip SRAM, Off-Chip SRAM, DRAM, Disk
  - The higher up on the pyramid, the faster/more costly the memory is. This leads to the phenomenon of lower capacity higher up in the pyramid.
- *Temporal Locality* - referring to the same memory location many times in the sam general timespan
  - Want to keep recently referenced items at higher level of memory hierarchy to make future references faster
- *Spatial Locality* - referring to many memory locations that are clustered near each other in the same general timespan
  - Want to keep neighbors of recently referenced memory locations in higher levels to make future references faster

- *General Memory Hierarchy* - **I/D L1 Cache**, **L2 Cache**, main memory, disk
  - *Benefits of Multilevel Cache* - on a cache miss from the top level L1 cache, you are *less likely* to incur the entire miss penalty of waiting for the main memory or disk
- *Guiding principle of Cache*: hit time « miss penalty. Still true of lower level cache, albeit to a lesser degree
- **Average Memory Access Time (AMAT)** - hit time + (miss rate)(miss penalty)
  - hit time is same as *cache access time*, which is still incurred even on a cache miss
  - in a multilevel cache, the miss penalty is the AMAT of the next layer
- *Cache Terminology*
  - Cacheline/Block - minimum unit of data that cache can allocate/evict from next memory level
  - Tag Bits - an identifier for each cacheline to identify which memory block it is (usually unused bits of memory address)
  - Cache Lookup - step of finding whether data is in the cache
    - After finding correct set, tag bits of address must be compared against tag bits in tag array
  - Set - a cache will assign each lower level memory block to a *set*, where each set will have a constant number of *blocks*
    - Note that from a data perspective, we arbitrate between sets of blocks. At a microarchitectural level (e.g MP3), we construct an SRAM block per way and make "mini-direct-mapped-caches", so to speak, then arbitrate between those.
- **Types of Caches**
  - *Direct Mapped* - N sets with one block each. Each block of memory in lower level can only be assigned to one block (but multiple lower level memory blocks will be mapped to same block)
    - Easy to implement, easy to index, but causes a higher eviction rate
  - *Set-Associative* - A block can be placed in one of W locations in a set - each of these locations is called a *way* (i.e a $4-way$ cache has N sets with 4 ways each, and a total capacity of $4N$ blocks)
    - Search mechanism per set is more involved, harder to implement, but eviction rate is lowered considerably
  - *Fully Associative* - A block can be placed in *any* cache location (i.e 1 set with N blocks)
    - Search mechanism is very complex/hard to implement, but your eviction rate is now only a result of cache being at capacity
- **Miss Types**
  - *Compulsory Misses* - cold start misses from beginning of program (no valid data in cache yet)
  - *Capacity Misses* - cache is full. can be avoided by increasing cache size
  - *Conflict Misses* - the set you would have pulled this block from is full, so you have to evict one (cycle continues). Mitigated by increasing size/associativity
    - Most felt by direct mapped caches. Alternative organizations reduce these
    - Also can be called "collision" or "aliasing"
- *Design Considerations of a Cache* - placement (associativity), identification (how to find a block), replacement policy, write policy
  - In this class, we primarily discuss LRU replacement policies
- Describing Caches - the essential parameters
  - T_{hit} (access time), capacity, blocksize, replacement policy, associativity, unified vs I/D

- **Write Policies**
  - Allocation: *Write-Allocate* vs *No-Write-Allocate* - a write miss bring block into cache
    - Pick which one to use depending on whether you want to exploit temporal locality
  - Update Policy: *Write-Back* vs *Write-Through* - memory updated in *cache* and main memory updated on cache eviction vs. direct main memory update
    - Write-Through has the benefit of maintaining memory and cache consistency. Consistency becomes difficult in multicore systems
- **Replacement Policies** - which cache line do we choose to evict?
  - LRU, Random, NMRU (randomly pick excluding the MRU)
- True LRU tracking requires a stack per set - this will in turn icnrease cache access times
  - stacks also become expensive to implement at a high set size (higher associativity)
  - In true LRU, there are $\log_2(W)$ LRU bits per tag entry in the tag store
- **Pseudo LRU** - create a BST-like structure for every set at the top level of cache. Since you need $W-1$ PLRU bits for an W-way associative cache, you create $W-1$ vectors at the top level sized to S each, where S is the number of sets.
  - After each access of a set, we update the relevant bit in each vector to denote whether we took the left path (0) or right path (1) for that split of blocks. PLRU assumes a 2^n way size due to the BST structure.
  - PLRU guarantees the removal of an *old* way but not necessarily the *oldest* way
- L1 cache can either be *unified* or be separated into *instruction* and *data* caches
  - A *unified* cache is simpler to construct in a non-pipelined system
  - For pipelined or single-cycle processors, concurrent access to instruction fetch information and mem fetch information is necessary - thus either a two-port unified cache or separated cache must be constructed
    - A two-port unified cache is more are-intensive than separated cache - thus that is more common
- **Cache Microarchitecture**
  - On a way-by-way basis, address is used to index data array and search tag array
    - If tag hit, the output is used as a mux select input to the data output
    - The tag array being smaller will allow it to propagate faster than data array out - output of data array not really necessary until hit-miss detected (i.e hit-miss detection+tag array propagation will absorb some of the extended data access time caused by denser routing)
- *Tag Store* - refers to the array of all the tags the cache is currently holding. Each entry in the tag store/array will have a valid bit, a dirty bit, (potentially) LRU bits, and then the tag bits.
  - Dirty bit is only included in *write-back* configurations
  - Tag store size *might* include pLRU bits, but confirm with your proctor or label it on your sheet

2 Midterm 2

2.1 Memory

- **Virtual Memory** - separates logical memory from physical memory
  - Can make logical address space much larger than physical address space, share address space between multiple processes
  - Requires a *virtual address* to be translated into a *physical address*
- Two approaches to Vmem
  - Segmentation and Paging - nowadays we use paging, and refer to that as "virtual memory"
  - Virtual memory requires both hardware and software support
    - Remember 391 - we had to set up the paging *mappings*. The hardware component decoding those mappings is called **MMU** (memory management unit)
- **Segmentation**
  - Separate address apce into segments, where `phys_addr = BASE + v_addr`.
  - Simple translation, gives us isolation and protection
  - Annoying to manage (different segment sizes, need to manage segment base/limit), can lead to fragmentation (external fragmentation from large segment size?), and only few segments are addressable at the same time

2.1.1 Paging

- Large contiguous *imaginary* virtual address space
- Allocates fixed-size chunks of address space called "pages"
- Maps virtual pages of memory to physical chunk, then uses offset
- Hierarchical page tables to reduce space usage, use TLB (translation lookaside buffer) to reduce translation latency
- x86 uses `CR3` as the PDBR (page directory base register) - stores base address of top-level page directory
  - On context switch, we update to new process's PD physical address
  - The "oh we'll just edit paging" shortcut is *just* for 391
- If virtual page is not mapped to physical memory, (0th bit set to 0 in PTE/PDE)
  - Hardware throws exception (pagefault), which OS needs to handle
  - Can happen due to program errror or virtual page is mapped into *disk* rather than memory
    - OS can save you in latter case by allocating PDE/PTE, loading disk data into RAM, then resuming program
    - A page might be mapped to the disk for **demand paging**, where you don't need to load disk data into memory until the program actually tries to use it (to conserve RAM)
    - or for **swapping** where you run out of RAM, and you need to "set aside" physical pages temporarily
      - If you swap between memory and physical memory frequently it is called *thrashing*, and this can cause extreme performance degradation (long disk waits)
- *Page Sizing*
  - Large page sizes will make your paging setup easier (fewer PTEs, fewer TLB misses), but can have memory impacts (no fine-grained permissions, large disk transfers, internal fragmentation)

1. Protection Most of this section uses x86 as a case study.
    - x86 has four privilege levels in x86 (Ring 0-3)
        - We mainly use Ring 0 for kernel and Ring 3 for user, but technically Ring 0-2 are all supervisor (OS stuff)
        - *Current Privilege Level (CPL)* determined by address of instruction you are executing (*Descriptor Privilege Level (DPL)* of code segment)
    - PDE and PTEs are each 32-bit
        - PDE entry flags will protect all subpages (or the fat page if you allocate 4MB)
            - Main things to look at are RW permission (set to 0 means no writes), and U/S (do you give Ring-3 access? only if this flag is 1)
        - PTE flags will only protect the relevant 4kB page
            - same RW/US bits
    - Processor will check segment protections then check paging protections - allows for different granularities of access protection.
2. TLB The Long Boi
    - Hardware structure to cache PTEs
        - Literally like a cache, you search virtual address here first before attempting translation
    - Need to flush on context switches to avoid stale translations, or need to associate TLB entries with processes (extra field in TLB entry)
        - Can invalidate entries of old process
    - TLB can either be managed by hardware, and hardware can traverse PTs on a TLB miss, or software can manage TLB
        - HW throws exception, software does the page walk, OS fetches PTE then inserts/evicts entries in TLB
        - HW-managed TLB means no exceptions, independent instructions can continue, and has small footprint (but fixes the paging schema)
        - SW-managed TLB allows OS to design paging schema or implement more advanced TLB replacement policies that are hard to make hardware for (but causes pipeline flushes and has a performance overhead)
3. Caches
    - Caches can be virtually or physically indexed/tagged.
        - Creates VIVT, PIPT, VIPT schemes
        - PIVT is possible, but pointless - you already have the physical address, just use it!
        - VIPT creates the optimization of not requiring cache flush, but we can translate physical address in parallel to cache lookup
        - Any virtually tagged cache must be flushed on a context switch, since the tags are now all invalid

2.1.2 DRAM Architecture

- A memory channel refers to a single bus from a set of memory modules to the processor
- A DIMM is *dual inline memory module*, usually how PC RAM modules are structured
- Front and back of a DIMM make up *memory ranks*, which are a collection of chips.
    - Each rank can be individually indexed and issued commands
- Each chip in a rank stores parts of the bits of the final rank output
    - Ex: 8 chips in a rank each storing 8 bits to form a 64-bit output

- Each chip subdivides into many banks, each of which stores a different version of the N bits that chip outputs
  - Banks are structured in a row-column format, where the final output of a bank muxes between all the outputs of the current row in that bank. The address will map directly into a row-column pair
    * *DRAM Page* refers to a single row in this format
    * To access a "closed row", we use the following protocol
      · Activate (place row into row buffer, "opening" it), R/W (modify buffer), and Precharge (close row, writeback, prepare for next row access)
    * Open rows can simply be issued the latter 2 command types
- *General Hierarchy*
  - Channel → DIMM → Rank → Chip → Bank → Row/Column
- *Static RAM* (SRAM) vs *Dynamic RAM* (DRAM)
  - Static RAM, or combinational ram, will retain its value while powered
  - DRAM stores its values in a capacitor, and thus requires refreshes over time to retain the value in those capacitors - also has a *destructive read*
    * Can either burst all rows *together* one after another, or distribute the refreshes (each row refreshes at a different time, at a regular interval)
    * Distributed scheme will eliminate long pause times, but is difficult to maintain bookkeeping for/prioritization
  - SRAM is 6 transistors per bit, whereas DRAM is only 1 transistor per bit
- Each DRAM bit is structured (at least in this class) as an NMOS transistor with bitline at drain, capacitor at source, and a "wordline" driving the gate
  - On *write*, the bitline is charged either high or low, and wordline is set *high* to trigger the capacitor overwrite
  - On *read*, we wait for bitline to become precharged, and wordline is set - depending on the charge in capacitor, bitline becomes slightly higher or lower
  - After a read, the capacitive cell enters an indeterminate state where it is neither logic high or low
  - Sense amp is used to threshold the cell to either logic high or low (is about 1/2 after a read)
  - Sense amp will drive the row buffer, which then overwrites the row of DRAM cells back to a full low/high signal
- DRAM row buffer means that the DRAM chip will try to read many bits at once, then read and reorder them based on different column addresses
- DRAM read accesses are fully asynchronous (RAS/CAS signals) whereas SDRAM transfers signal sycnhronous with the clock
  - DDR DRAM offers dual edged data transfer
- *Burst Access* - one one command access, multiple bytes are read or written
  - Hardware can provide multiple burst lengths, which software then chooses from
- Memory controller on the DRAM bank may choose to coalesce or reorder memory access to reduce number of row accesses
- *DRAM Latency*: CPU-to-controller → controller latency → controller-to-DRAM bank → bank latency → dram-to-controller-to-cpu latency
  - Controller latency depends on queuing/scheduling scheme, and how it converts accesses to basic bank commands

  - Bank latency depends on if row is already open, if array is precharged, or if the previous row buffer needs to precharge + open new row

- **DRAM Page Management Policies**
  - *Open Page* - keep row buffer open until a conflict
  - *Close Page* - close the row buffer after an access

- **DRAM Scheduling Policies**
  - *FCFS* is basically just a FIFO of requests, *FR-FCFS* will prioritize row-hits, then the oldest request (i.e when picking which row to open next, or which request to service within current row)
  - Can prioritize based on request age, row buffer status, request type (tag as prefetch, read, write), requestor type (load/store miss), or how critical the request is (more metadata passed to memory controller)

- Multiple banks allow for pipelining of memory requests (reduces impact of DRAM access latency)

- Multiple channels can increase bandwidth or allow for concurrent access on indepedent channels, but requires more complex hardware and more production cost (more wires/pins)

- Consecutive cachelines can be placed in *either* the same row for better row buffer hitrate or different banks for better parallelism
  - Address can break down as `row:rank:bank:channel:column:blkoffset` or you can move column in front of `rank` field
  - Depending on the volume of data that'll likely be accessed in a single shot, you can choose one or the other scheme…

2.2 Prefetching

- We prefetch data into cache before program needs it to lower program delays as a result of memory latency
  - Can also avoid compulsory cache misses
  - *Cannot* eliminate capacity msises, conflict misses, coherence misses (invalidation)
- Need to predict which address is needed in the future
- **Four Questions of Prefetching**
  - *What* address, *when*, *where* to put data, *how* to manage prefetches
  - **What**: needs to be selected carefully, since prefetching useless data wastes resources (bandwidth, cache space, energy, etc.). Based on past accesses/compiler's knowledge of DS, we use prefetching algorithm to choose what to prefetch.
  - **When**: prefetching too early can cause relevant data to be evicted before usage, too late will expose some memory latency to runtime.
    - Being too aggressive can cause premature prefetch eviction
  - **Where**: placing it in cache will simplify design but can evict useful data or pollute the cache. A separate prefetch buffer will protect demand data and keep the cache clean, but is a more complex system to design.
  - **How**: Does software prefetch w/ prefetch instructions, does *hardware* prefetch based on seen procsesor acesses, or is there an execution-based prefetcher, where a thread is executed to prefetch data for the main program?
    - Thread can be HW or SW generated
    - In software, can either prefetch into register w/ `ld` or into cache with a special instruction (but requires more processing/exec bandwidth, and adds more complexity for developer)
    - Hardware prefetchers can be tuned to system implementation and there are no code portability issues/instruction exec overhead, but you require more complex hardware to detect relevant patterns (in some cases software more efficient)
- *Next-Line Prefetcher*: After a demand access or demand miss, fetch the next N cachelines
  - Easy to implement, good for sequential access patterns (instructions, arrays)
  - Accuracy nosedives for *non-sequential* acccess patterns, so wastes bandwidth
  - Needs some detection of forward vs. backwards traversion to avoid prefetching next N lines on a backward traversion
- *Stride Prefetcher* - Can be based on PC or on cache block address
  - On each load, we compute `stride = ld_addr_2 - ld_addr_1`, prefetch `ld_addr_2 + stride`
  - For instructions, waiting 2 load instructions may be too much to initiate prefetch - we can use lookahead PC to index a prefetcher table
    - Similar concept with cache-block addresses
- **Benchmarking Prefetchers** - we evaluate *accuracy*, *coverage* (how many prefetches were used where there would normally be a miss), and *timeliness* (if we *do* use a prefetch, how often is it on-time)
  - Should also consider bandwidth consumption and cache pollution as performance evaluation points
  - *Prefetcher distance* is how far ahead of demand stream you are, *prefetch degree* is how many prefetches per demand access
    - Ex: A 4th degree NLP
- Prefetcher can be made more timely by making it more **aggressive** - staying far ahead of processors access stream

  - Very aggressive is well ahead of load access stream, hides memory latency better, more speculative (better coverage/timeliness, worse accuracy)
  - Conservative hardware prefetchers are more accurate and require fewer resource, but have lower coverage and are less timely (closer to load access stream, some memory latency may be exposed)

- *Execution-based Prefetcher*
  - Executes a partt of the pruned program ahead of time to prefetch data
  - The executed code is essentially a *speculative thread*, which can be executed on a separate core or the same core on a separate thread context, or during idle cycles caused by cache misses
  - **Reduction**: *where* to execute the precomputation thread, *when* to spawn the ethread, *when* to terminate the thread (and how?)
    * Insert spawn instructions ahead of the load, or when the main thread is stalled
    * Termination can be done with either pre-inserted `CANCEL` instructions or based on feedback

2.3 Out-of-Order and ILP

- *ILP* - instruction level parallelism, measure of inter-instruction dependency in an application
  - Assumes infinite resources, unit-cycle operation, and a perfect front-end
  - Upper bound of IPC, bounded by dependencies (data/control)
  - Larger ILP window means more instructions can be parallelized at once
- *Memory Dependency* - we need dynamic memory disambiguation mechanisms to increase ILP, such that parallelism is done in either a safe or recoverable way
- **Speculation** - use branch prediction to temporarily waive the control dependency, flush the pipeline on branch mispredict when that branch is actually evaluated
- ILP is best exploited if instructions are executed *as soon* as all source operands/destination locations are ready, and the execution unit is available
- **Dynamic Scheduling** - execute instructions out of order to exploit ILP
  - HW will track/maintain dependencies
  - Allows for scalable performance and handles cases where compiler does not know dependencies, but increases hardware complexity
  - In OoO, instructions *still* need to be committed (marked as complete) in order to track dependencies correctly

2.3.1 Tomasulo's Algorithm

- *Tomasulo's Algorithm* - ubiquitous OOO implementation, very widely used in modern processors both in base/modified forms
  - Has two functional units - FP Add, FP Mul/Div
  - Reservation stations (3 entries in FP Add, 2 entries in FP mul/div)
  - Tags (4-bit tag for each of 11 possible sources - one of the 5 RS entries or an FP load buffer [FLB])
    - Tag assignments/renaming eliminate false dependencies
    - *Common Data Bus* or CDB is driven by the 5 RS and 6 FLBs, is broadcasted to each operand in each location (2 operands per RS, 3 SDB, 4 FP regs [FLR])
    - Avoids centralized register file, instead RS will poach operands from CDB
      - Pending instructions will designate RS w/ relevant input
    - Register status table keeps latest write
- *Keys in Tomasulo's Algorithm* - `op`, `v = value`, `q = tag`, `a = address`
  - RS entries are enumerated 1-5, so 1-3 in add and 4-5 in mul/div
  - If tag is 0, the value is *already in the RS entry*, if tag is 1-5, it is coming from another RS entry's command

2.3.2 Enhanced Tomasulo

- When executing out-of-order, you don't know at which point an interrupt or exception occurred, don't know where to return
- There is also the issue that we need many reservation stations for a large instruction window, since cache miss can take many cycles (and we might want to service more instructions in meantime)
- **Exception Types**
  - Page Fault, Misaligned Memory Access, Memory Protection Violation, Undefined/Illegal opcode, Arithmetic Exception

- With imprecise interrupt, in current Tomasulo's model we potentially "corrupt" register value with out-of-order execution (later instruction overwrites register state)
  - When faulted instruction re-runs, the register state has changed
- *Precise Interrupts* - Keep precise state of execution, all instructions before interrupted instruction shoudl be completed, and state should not appear as if anything occurred after interrupted instruction
  - Interrupted PC should be given to interrupt handler
  - Similar to how a branch misprediction is handled, but OoO makes ordering hard
  - We need precise interrupts to know where to resume execution after interrupt handler
  - *How?* - buffer results (ROB), reconstruct scenario as sequential execution (commit stage), restart from saved PC when interrupt ends
- *Speculative Tomasulo's*
  - Add a ROB - stores instr. type, destination, value
  - Instead of using reservation station as tag, use ROB entry
  - It is now easier to "reset state" on a branch mispredict or exception
  - Four Steps: Issue, Execution, Write-Back (CDB write + free the RS), Commit (update register if ROB has this instr at head)
  - *Caveat* - size of reservation stations/reorder buffer grows with number of instructions in flight
  - Processors can only look at a small portion of the potentially larger ILP window to pick issued instructions

2.4 Multicore Processors

- Hard to make single-core clock frequencies higher, and ILP can only be extracted so far (expensive, slow, power-hungry logic)
  - Highly pipelined circuits have heat problems, are hard to design/verify, need large design teams
  - Most modern operating systems are scheduled, multithreaded pieces of software - why not just execute them on separate cores?
  - Two 1GHz cores are more power efficient than one 2GHz core, while maintaining throughput
    - Need to find parallelism in programs to occupy cores
- As uniprocessor advancements slow down, multicore architecture is the way forward
- **Thread-Level Parallelism (TLP)** - coarse-scale parallelism, where independent tasks are separated into different threads
  - Differs from *ILP*, which is fine-grain parallelism between independent instructions
  - Single-core superscalar processors cannot fully exploit TLP - while they present the illusion of concurrency, their throughput is still the same. TLP can augment throughput on multicore parallel processors
- *Chip Multiprocessor* - multiple cores on a single processor socket (multi-core CPU chip)
- OS views each core as a separate processor, thus the scheduler must dispatch threads/processes to them individually
- **Random Terminology**
  - *Multiprocessor* - any computer with multiple processors
    - Multiple processors on motherboard can be connected by a special short-distance interconnect
    - Can have *shared* or *distributed* memory scheme
  - *SIMD* - single-instruction, multiple data
  - *MIMD* - multiple instruction, multiple data
- Multi-core processors are a *subset* of multiprocessors, where all processors are on a single chip
  - Classified as MIMD, since different cores execute *multiple instructions* at the same time on *different parts of memory*
  - Is a *shared memory* multiprocessor, since all cores share the same memory (think about RAM DIMMs on mobo)
- *Simultaneous Multi-Threading (SMT)* - multiple independent threads run simultaneously on a single core
  - Instructions interleaved between multiple threads (hyper-threading)
  - ex: one thread waiting on mul, another thread can use int ops
  - Does not magically erase structural hazards or constraints, but can enable better threading
  - OS and applications consider each simultaneous thread a separate virtual processor (4 core/8 thread)

    > Multi-core has multiple weaker cores, and optimizes TLP. SMT has a single large/fast *superscalar* core and optimizes for single-thread performance and ILP.
    >
    > * LeBajji Rao
- Multi-Core processors will have private L1 Caches per-core, and L2 cache privatization is a design decision
  - *Private Caches* are faster since they are closer to core, and there is less contention

  - *Shared Caches* allow multiple threads on different cores to have access to coherent cache data, and offer more cache space in the event a small number of high-performance threads are running (not all cores fully utilized)

- *Symmetric Multiprocessor (SMP)* - multiprocessor with shared memory
  - Also called shared-memory multiprocesing

2.5 Cache Coherence

- Idea - data should be consistent between the memory accessed by each core - this becomes an issue with *private* caches
  - General multiprocessor issue, not just multi-core (arises whenever you have a private or distributed memory schema)
- **Defining Memory Coherency**
  - Program order is preserved - if the memory location is not shared, the core acts as a uniprocessor
  - If an address is written to, it must eventually be seen by *all processors*
  - Commands issued to an address should be ordered uniformly across all cores - we want to *preserve causality*
- We now have *four* Cs for cache misses - compulsory, capacity, conflict, and *coherence* in the event of invalidation
- Coherence Miss comes from two sources
  - *True Sharing* - same data accessed by multiple processors
  - *False Sharing* - different data in same block accessed by multiple processors

2.5.1 Snooping

- **Snooping** - monitor memory commands issued on memory bus, use that to update state of private storage
  - Typically used on shared-bus multiprocessors (SMP)
  - Separates into *write-update* and *write-invalidate* flavors of snooping
    - *Write-Update* requires that the new memory value is *also* broadcast to all other cores so they can update copies
    - *Write-Invalidate* invalidates all other cached copies, forces a re-fetch of the relevant memory
    - "Better" depends on application, but write-invalidate is simpler to implement and is more common in multiprocessors
- Coherency protocol state is stored as part of the per-cacheline state in a cache
  - each block maintains an instance of the protocols discussed below
- **MSI Protocol**
  - *Invalid* - not cached in C, need to make a request on bus before RW
    - Enters this state when *a bus write is snooped*
  - *Modified* - B is dirty in C, and no other cache has the block - can freely RW
    - Need to write-back on eviction or if another core has read miss (snooped bus read)
  - *Shared* - B is clean in C, but other caches have the block - can read without going to bus, but need to send a bus request to write
- What to do if memory is in S state and another core requests it?
  - Let memory supply it?
  - All cores in S state try to write to bus, whoever wins arbitration posts the value
  - Separate state *like* S that *maybe* others have the same data we do, but we are designated "providers" of the data if someone else asks
  - Note: much clearer if data is in M state - whoever has it needs to supply it, that is the only accurate copy

- **MESI Protocol (Illinois Protocol)**
  - We add the *Exclusive* state, where data is clean but only one cache has a copy
  - Reduces bandwidth overhead, since we don't need to broadcast any memory writes to this block until transitioning into the S state
- **MOESI Protocol**
  - Adds the *Owned* state, where multiple caches have the latest copy, but this cache will source data for cache-to-cache transfers

1. An Aside Regarding Write-Through Caches When operating the above protocols, *write-through* vs *write-back* will affect how you transition states. Focus on consistency with the underlying memory rather than whether a write was made at all. Especially for a write-through cache, you can ***never enter the M state***. As a result, it makes no sense to have an *exclusive* state, since we are always required to broadcast memory writes anyway. Thus it is most efficient, and possible, for write-through caches to support an *SI* scheme when snooping.

2.5.2 Directory Based Coherence

- Typically in distributed shared memory (physically separate, single address space)
- For every local memory block, local directory has an entry indicating who has cached copies, and what state they have the block in
  - Dirty bit for entire block, presence bit per processor P
  - *Not a file directory*, separate hardware controller/construct parallel to memory
  - Misses are all routed to block's "home node", and directory will perform necessary coherence actions

```
Ex: Read Miss
On read miss, directory controller will find entry for the block.

If dirty, block is requested from the processor with presence bit, memory is written,
D bit is cleared, block is sent to requestor and presence bits updated.

If not dirty, then you just read memory and send the data.
```

- Note that each cache will store its own coherence state, but the Directory helps avoid broadcasts and order accesses to each location in a larger memory system (becomes overwhelming in super-multiprocessors)

2.6 SIMD + GPU

- *Flynn's Taxonomy of Computers*
  - SISD - Typical Uniprocessor
  - SIMD - Vector Processor, GPU, etc.
  - MIMD - Multiprocessor (e.g multiple instruction streams)
    - Separates into tightly coupled *shared memory* and loosely coupled *distributed memory* topologies
  - Technically MISD *could* exist, but it's pointless since it reduces down to SISD
  - *SIMD Execution* - a single instruction triggers operation upon many data elements
    - For example, a vector add instruction adding two vectors across many memory locations
    - Many applications for this class of execution, basically anything data-heavy or data-driven
      - Multimedia applications tend to have contiguous data items with short data types
- *Advantages of SIMD*
  - In addition to ILP, we can now take advantage of data parallelism
  - Simple design, since you just replicate functional units
  - More energy efficient, since you avoid redundant fetch/decodes
  - Since you only require one instruction scheduler for multiple processing units, you reduce die area
  - *Caveat* - Hardware cannot identify SIMD opportunities, this must be pointed out by compiler or software programmer, and hardware needs to expose instructions to do this

3 Final Exam

3.1 IO Subsystem

- Computer System comprises of Memory, Datapath, Control and *I/O*
- Want to make it so that the communication between processor and peripherals is *modular* - we should be able to swap out devices without having to swap out the entire bridge.
- Nowadays we have a hierarchical interconnect
  - High-performance interconnect that all cores are directly connected to - this interconnect is also connected to memory, display, and bridge to legacy/low-speed interconnect
  - Low-speed interconnect connected to Disk, Keyboard, Network, etc.
- Buses vary in *width* (no. of wires), *transfer size* (length of transaction data), and *synchronicity* (is bus clocked or self-clocked)
- Processor-Memory Bus is *wide*, *fast*, and *short*
  - CPU + Cache + Interconnect + Memory Controller -fixed topology
- I/O and Peripheral busses are *slower longer* and *narrower*
  - Flexible topology w/ multiple and varied connections
  - Standards for how to interface with these buses like USB, PCI, or SATA
  - Connected to fast bus over bridge
- Term "interconnect" and "bus" used somewhat interchangeably in this context - parallel set of wires for data transfer and control
  - Multiple senders/receivers, all bus transactions are "public"
  - Bus protocol - defines how to use the bus wires
- *Point-to-Point Interconnects*: instead of unified bus/bridge, which can create bottleneck, we create dedicated point-to-point channels
- Typically, a device will have a set of interface registers
  - Command registers - send command to device, writing to this will trigger something
  - Status registers - read status of device (doy)
  - Data registers - write will send data, read will receive data (R/W access for these ones)
- Communication Interface
  - Special commands for special I/O buses - port-based I/O
    - Downside: only possible in kernel mode, and kernel-user transitions are expensive
  - Memory-Mapped I/O: Bus will map the I/O registers into specific memory addresses
    - If paged correctly (protection bits set/unset), can be accessed using normal LD/ST commands, even in user space
    - Data/Commands/Status are all transmitted over the high speed memory bus
    - OS, MMU, and devices configure the actual memory mapping
    - Faster than port-based I/O since there is no user/kernel space switching

Programmed/Port IO	MMIO
Special Instructions	Load/Store
Dedicated HW Interface	Memory Bus
Protections enforced w/ Kernel Mode	Paging enforces permissions
Hard to Virtualize	Normal memory Virtualization

Finer Comparisons

- *Polling vs. Interrupt based communications*
  - *Polling* - Predictable timing, inexpensive, but can waste CPU cycles
    - More efficient if there is always something to do (fast network cards, for example)
  - *Interrupts* - Device sends signal, CPU is interrupted
    - "Cause" register identifies the device, handler determines what to do by examining device
    - Can now establish priority on which devices to handle first, which devices can interrupt other device handlers
    - Improves efficiency of device interaction, but actually running the handler is more expensive because of the save context and kernel space switch
- *Direct Memory Access (DMA)*
  - OS will provide an address for where the data is and how long it is, the controller or device will autonomously copy it over/write to it
  - Interrupt triggered when completed, or if there is an error
  - Much more efficient, since you save many unnecessary instruction cycles in the CPU (would normally have an LD-ST loop)
- Since RAM needs to interact with DMA, but RAM does not know virtual addresses, DMA requests are sent from OS using physical addresses
  - OS allocates contiguous physical pages for DMA
  - miniTLB for mappings of DMA to virtual address
  - Page needs to be pinned prior to DMA so that there are no issues with dest. page getting bounced to swapdisk
  - *Bounce Buffer* - DMA is done to a pinned kernel page, then memcpy to a destination page
  - OS will flush cache (partially or fully) before DMA, and won't touch pages during DMA. This keeps caches coherent
    - OS can also mark pages as uncached - this is needed for MMIO regs

3.2 Storage

- CPU performance is getting better every year, but I/O performance sees sub-10% improvements due to mechanical limitations
- The storage bottleneck reduces the effectiveness of higher RAM/CPU speeds
- HDD and SSD have the same form factors, and similar interfaces (SATA, PCIe, SCSI)
  - The "e" stands for express!
- I/O subsystem performance is based on *response time* and *throughput*
  - Latency increases from software paths, hardware controller, and I/O device service time
- A single *platter* in an HDD is comprised of multiple concentric *tracks*. Those tracks are divided into *sectors*
  - OS transfers groups of sectors called *blocks* together
  - A disk can access any block directly
- Disk usually has 500-20,000+ tracks (rings) per surface, and 35-800 sectors per track
- Bit density is constant - more sectors on outermost rings since they are larger
  - Speed may vary with track location
- *Cylinder* - all tracks under the head at a given point on all surfaces
  - Head has multiple pins, one per platter it seems

- HDD R/W process
  - *Seek Time* - head needs to move over correct track (~5-10ms)
  - *Rotational Latency* - wait for the track to rotate to the correct sector(s) (~8ms)
  - *Transfer Time* - transfer a sector under R/W head (~50/MB/s)
  - `Disk Latency = queueing time + controller time + HDD_R/W_time`
    - Ignore seek time if on correct track, ignore rot. delay if near correct sector on the same track
  - Queue is in software (device driver), controller refers to piece in between software queue and HDD
  - HDD is *cheap* - $<\0.01/GB$
- Disk also needs a scheduling algorithm
  - FIFO, Shortest Seek Time First, SCAN, C-SCAN
    - Fair among requesters, but may run into very long seek times
  - *SSTF* - pick request closest to the head (minimize seek time)
    - Reduce seek times, but can cause starvation
  - *SCAN* - like an elevator, you take the closest request in the same direction you were already moving. Head moves towards center, then outwards
    - No starvation and low seek times, but you favor middle tracks
- *Logical Block Addressing (LBA)* - a way of linearly addressing blocks w/o specifying physical characteristics
  - Allows us to abstract memory without needing to know details about the internal memory architecture
- Interface Speeds
  - SATA-III ~= 500 MB/s
  - PCIe ~= 1GB/s per lane
    - Up to 16 lanes w/ low power and broad hardware support

3.2.1 SSDs

- Since 2009, have been using multi-level cell NAND flash memory
  - Stores 4-64 pages per memory block, with 4KB pages
  - No moving parts, so no rotational/seek latency
  - Very lightweight, low power
- SSD comprised of many NAND devices in parallel connected to different channels
  - Anywhere from 10 to 60+
  - Very high storage, with higher bandwidth than an HDD
- SSD controller will make storage act like an HDD from the CPU perspective
  - Each channel gets its own controller
  - Contains microcontroller, DRAM buffer, error correction, flash interface modules
  - *Flash Interface Modules (FIM)* - physically/logically connects controller to NAND flash devices, more FIMs will increase performance
  - *Microcontroller* - performs error correction on data, converts LBA to physical address, will move data to flash or from flash
    - Usually some kind of ARM core, but the IP fees are going to stack up

  - *DRAM Cache* good to cache the information you retrieve from flash to increase performance, and can buffer address translation/mapping table
  - *NAND Flash* - cells arranged in multiple planes, which means you have parallel access to NAND, and can interleave memory accesses
  - NAND flash is cheap, but the controller is very expensive
- SLC vs. MLC vs. TLC vs. QLC - more bits per cell (single level vs multi, triple, quadruple), at the cost of speed and reliability
- Host will talk to some kind of buffer manager/software queue over SATA, as usual (this queue is in the hardware)
  - Queue will decide whether to dispatch the request to the DRAM cache buffer, or to the flash memory controller
  - Reading data similar to memory read (25 micros), no seek or rotational latency (only transfer latency)
  - Latency is bottlenecked by the controller and the disk interface
  - `latency = queuing time + controller time + transfer time`
  - Bandwidth maximized on sequential random reads
- Flash memory chips will not let you granularly overwrite data, you have to erase the entire block before you can write to a page
  - Erasing block takes about 1.5ms, write ranges from 200 micros to 1.7ms
  - Out-of-Place update means that you write your data to a new block instead of overwriting same block
    * Controller will maintain a list of obsolete/empty blocks, and will garbage collect obsolete pages by copying copying valid pages within block w/ obsolete pages to a new erased block
    * Mapping table must be updated for LBA translation
- Write and erase cycles are performed on high voltage (~20V), which damages memory cells and reduces SSD lifespan
  - Wear leveling tries to move around and arrange data so that data writes and erases are distributed across the entire available NAND flash
- SSD has a flash translation layer between the filesystem and the flash
  - Flash translation layer will maintain address translation and wear leveling
    * Translation is needed since SSDs can update their blocks out-of-place, meaning that memory contents move around the NAND array
- SSD costs about 10x as much as HDD per GB, and is about half the size in the upper end
  - SSD has up to 10x the bandwidth compared to HDDs, and DRAM will have 10x the speed of SSD

TL;DR: SSDs are low-latency high-throughput devices (since they eliminate seek/rotational delays), and have no mechanical moving parts. This makes them very lightweight, lower power, quiet, and shock insensitive. They also have reads on par with memory speeds.

However, SSDs are significantly smaller and more expensive than a disk, have asymmetric block writ performance, and a limited drive lifetime (failure rate is usually about 6 years, with a 9-11 year lifetime)

3.2.2 NVM

- Core count doubles every 2 years, but DRAM DIMM capacity only doubles every 3 years - memory is becoming a bottleneck
- DRAM-based main memory is hitting power and scaling bottlenecks
  - DRAM does not scale well to small feature sizes, and majority of power consumption large servers is already memory
  - We need alternatives to achieve better scaling
- *Storage-Class Memory* - non-volatile memory systems that have performance very close to DRAM, but with very high capacity
  - Also called *non-volatile memory architecture*
- Usually memory hierarchy has DRAM, then significantly slower Flash and HDD memory
  - The main memory system needs to increase performance, in order to reduce performance losses from memory misses from Flash/HDD accesses
- Examples of emerging scalable NVM technologies
  - *FeRAM* - 50ns R/W, destructive read, and low density. *Good for high endurance*
  - *(STT)-MRAM* - 30-50ns R/W, and good scalability (high capacity), but incurs large write current and low on-off ratio (limited endurance)
  - *RRAM* - Great density, but performance is much worse than DRAM (100 micros), and high write currents
- *Phase Change Memory (PCM)* - PC material exists in two states, amorphous (high resistance) and crystalline (low resistance)
  - In PCM, the resistivity determines bit storage. Cell can switch between states fast and reliably
  - Phases are switched by heating the material w/ electrical pulses
  - On SET, sustained current pushes cell above T_{cryst}. On RESET, cell is heated even higher than before, and "quenched"
- PCM scales better than DRAM, has small cell sizes. Can also store multiple bits/cell, meaning higher memory density in same area (dual bit for now). It is also NVM, with retention lifetime of 10 years.
  - *Caveats* - Higher read latencies than DRAM, limited write endurance, slow/power-hungry writes
- With PCM comes new challenges - higher read/write times mean we cannot just directly use PCM
  - Hybrid memory system architecture suggests using DRAM as a cache of sorts for larger capacity PCM memory
  - Fast read/write times with reduced miss penalties due to (much) higher PCM capacities
  - Performance will suffer in applications with poor locality due to higher eviction rates from the smaller DRAM buffer
  - DRAM caching can reduce PCM write traffic (coalesced writes over time)
    - Can reduce traffic by only writing dirty words, lines, or bits (some bookkeeping necessary) when evicting pages
- While PCM solves the issue of insufficient DRAM sizing, we now need to be able to bridge the gap between DRAM/main memory and storage
  - Optane DC persistent memory - fast, high-density, non-volatile memory that sits between the DRAM and lower level storage

- Intel DC Persistent Memory Controller interfaces with optane media channels on one end, talks to Host CPU over a DDR4 slot
  - The memory-end of the memory controller is rather standard - scheduler, read/write queues, and error handling logic
- Do we organize persistent memory access with filesystem or virtual memory?
  - Originally, Intel used a filesystem, since the filesystem has persistency with the rest of the NVM, but filesystem overhead/software latency is very high
  - Now, filesystem is used to organize data on-device, but application will interact with the persistent memory using memory mapping
- There is still volatile memory in parts of the CPU (like cache, or DRAM) - need to be able to restore that data on power failures
  - Industry uses battery-backed technologies to make sure cache is also somewhat persistent
  - Asynchronous DRAM refresh (ADR)
- Intel offers two memory modes for Optane
  - *Memory Mode* - use Optane as a very large memory, data persistency is not guaranteed (data not regularly refreshed from caches) and this improves performance
    - In memory mode, DRAM used as a write-back cache managed by the host memory controller
    - DRAM cache hits will have the same latency as always, but now cache *misses* are more likely to have the DC persistent memory latency than full storage latency
  - *Persistent Memory Mode* - requires persistent-memory-aware software or application, but this adds a new tier between the DRAM and storage.
    - No paging, context switching, interrupts - just in place persistency
    - Byte addressable load-store access, has DMA capabilities, and is cache-coherent
- Write-reordering is okay with DRAM, but we need to be more careful with NVM. Write-back cache destroys the sequentiality of writes.
  - If there is a crash, then write reordering may make it seem like all the pending stores were committed, when in reality on some of them were. OS is expecting data consistency for all stores before the last one it committed, but doesn't get it (i.e cache may have data that persistent memory doesn't have)
  - *Simple solutions* - don't cache data before persistent memory, use write-through cache, flush entire cache at commit
  - Can also define a set of memory stores as a *transaction*, meaning they should be atomically committed to the persistent memory

3.3 Energy-Efficient Computing

- Terminology
  - *Work* - movement of electrons, we use it to refer to amount of computation needed
  - *Energy* - capacity to do work (joules)
  - *Power* - work per unit time
  - *Power density* - power per unit area
- Power density is growing at a staggering rate
- $P_{tot} = P_{dyn} + P_{static} = C_L V_{dd}^2 f + V_{dd} I_{leak}$
  - dynamic is switching power, static is leakage power
  - Dynamic power dominates, but static/leakage power is increasing in newer process nodes

  - Higher power density increases CPU cost, since you need more hardware to keep CPU temps low, and beefier power delivery circuitry

- A little less than half of the power goes into the Clock signal, the vast majority of the rest goes into Datapath, Memory, and Control IO (in that order)
- Power Reduction Techniques
  - Voltage scaling, Clock scaling, Some circuit design techniques, Lower power logic synthesis, specific circuit technologies
  - *Challenges* - area complexity, potentially lower performance when reducing power consumption
- *Dynamic Voltage Scaling (DVS)* - OS controls processor speed, finds minimum voltage needed for the desired speed
  - *DVFS* - Intel's CPU throttling technology, SpeedStep
- *Dynamic Thermal Management (DTM)* - e.g thermal throttling. Software/Hardware techniques at runtime to limit CPU's operating temp
  - DTM tries to provide inexpensive HW/SW responses, reduce power, and limit performance hit
  - DTM can be triggered by temperature sensors, on-chip activity counters
  - When triggered, DTM can scale clock frequency, voltage, throttle the decode/speculation, disable ICache, migrate computation to a different unit, etc.
  - Migration of computation keeps a secondary regfile/computation unit, uses that one while main one is hot
- Smaller processes usually have greater leakage power, and this power/current increases with temperature
  - Can use *power gating* with a sleep signal to turn off supply voltage - this reduces both dynamic and leakage power
  - *Clock gating* reduces switching activity, thereby reducing dynamic power (but not static leakage power consumption)
- *Pipeline Gating* - if branch predictor is more inaccurate, then turn off speculative execution (gate the front end of the pipeline) to save power

3.4 Hardware Accelerators

- Accelerator appears as device on the bus. Coprocessor will excute instructions that are dispatched by CPU
  - Accelerators help increase cost/performance ratio - custom logic may perform operation faster than CPU of same cost, since it is more specialized
  - Accelerators also improve real-time performance, since time-critical functions have less contention, and are put on less-loaded processing elements
  - Good for real-time I/O processing, data streaming, complex algorithms like NNs, etc.
- Accelerators can be implemented on ASIC and on FPGA
  - FPGA is fast to market, has a simpler design flow, and is more flexible, at the cost of higher unit cost, higher area/power, and impacted performance
  - ASIC is slow-to-market and has a much harder design flow, but is cheaper per unit, has better performance, and reduces area/power
- 3 steps of Accelerated System Design
  - Does system actually need to be accelerated?
  - Make the accelerator

  - Design the CPU interface
- Accelerator Execution Time: `Taccel = Tin + Texec + Tout`
  - Tin and Tout are artifacts of synchronizing with master CPU
  - CPU can potentially have multithreaded or nonblocking execution - keep chugging along while accelerator cooks
- Accelerator implements pretty standard I/O interface for CPU via control registers, and some data registers for small data objects
  - Accelerator may have special R/W logic
  - Can use *remote register procedure calls (RPC)* to offload data/commands to the accelerator
- Hardware Accelerators have limitations - not all applications can be improved, tools need improvement, and design strategies often overly specialized (design may not be adaptable to future changes in algorithm or general architecture of accelerated task)
- Google TPU - hardware accelerator designed to speed up machine learning workloads
  - Optimized to perform highly parallelized matrix operations - is structured similarly to a normal CPU, albeit with a slower clockrate
  - Main competitor is NVIDIA GPUs, which have many small cores, and are also able to perform massively parallelized operations
  - Lower frequency and power than GPU or CPU, but 25x the multiply-accumulate units (MACS) compared to GPU, and 100x the MACs in a CPU
  - TPU/GPU uses HBM (high-bandwidth memory to keep up with the high throughput of computations
  - Has a matmul unit with 64K MACs, CISC instruction set + 4-stage pipeline, and 8GiB RAM to store the weights for active models
  - TPU can be used for inference and for training both

3.4.1 Near-Storage Computing

- Moving data between the memory hierarchy is expensive (storage → DRAM → cache → DRAM → storage)
- You can either reuse the low-power processor in the SSD controller or add another one to the controller to perform computations at the storage level
  - "ISC" (in-storage computing) has this core operate off of the DRAM cache in the SSD controller to perform computations while avoiding the memory hierarchy latency
  - SSDlet code chunk is compiled on the host system and then downloaded to the SSD, then host will request that the code be run
  - Examples of ISC Accelerators: data preprocessing for database workloads, NN computations very close to the flash chips