|  |
| --- |
|  |
| Dennard Scaling:  A measure of how chip capability increases at constant power draw due to a decrease in transistor feature size.   * Originally, *Voltage* and *Capacitance* were both 0.7x leading to no increase in power draw * 2x Transistors and 1.4x freq. led to **2.8x** increase in performance. * Eventually voltage could no longer be scaled down and freq. could not be increased. This led to a performance increase of only **1.4x.** | | | | | Types of ISA:  **Stack Machine –** uses only pop, push, and add instructions   * “Push” loads memory into 1st register (“top of stack”), moves other regs down * “Pop” does the reverse * “Add” combines contents of first two registers, moves rest up   **Accumulator Machine**   * Only 1 register (called the “accumulator”) * Instruction include “store” and “acc ← acc + mem”   **Register-Memory Machine**   * Arithmetic instructions can use data in registers and/or memory   **Load-Store Machine**   * Arithmetic instructions can only use data in registers | |
| ISA Intro:  Instruction format: Length, Registers, and Operands  **Length**:   * Variable * requires multi-step, complex fetch and decode * smaller binary programs which leads to less DRAM access and better cache efficiency * Fixed * larger binary programs which leads to more DRAM access and lower cache efficiency * easy fetch and decode * simplifies pipelining and parallelism   **Register Count**:   * Small number of registers * requires fewer bits to specify which one within the instruction * less hardware (less power and energy) * faster access (shorter wires and less gate latency) * Faster context switch (fewer registers to save) * Large number of registers * Fewer memory accesses (load and stores) * Easier pipelining since you don’t need to wait for memory   **Operands**:   * Types – memory locations, registers and immediate values * Location – keep consistent operand bit fields (decreases wires) | | | | |
| Performance:   * **Response time (latency)** refers to how long a job takes to execute * **Throughput** – how many jobs can the machine complete in a minute * ; * Cycle time – seconds per cycle | clock rate – cycles per second | **CPI** – cycles per instruction * **Single Cycle CPU** * Cycle time is determined by the critical path of the longest instruction * One instruction per cycle is executed * **Multi-Cycle CPU** * Cycle time is determined by the longest * Instructions are broke up in to smaller parts or **State**. * MIPS – Millions of instructions per second * Higher for program using simple instructions | |
| Memory Hierarchy   * **Temporal Locality –** reference to same memory location many times (close together in time) * **Spatial locality –** reference to near neighbors around the same time (Makes use of DRAM burst) * + | | | | | Four Central Questions About Caches   * P-I-R-W * **Placement** – Where can a block of memory go? * **Indexing** – How do I find a block of memory? * **Replacement** – How do I make space for new blocks? * **Write** **Policy** – How do I propagate changes? * Capacity of Cache – the total amount of data that cache can hold 🡪 This does not include the tag, dirty, valid, and LRU bits | |
| General Facts of DM, FA, and SA   * Tag bits are used for both direct mapped and fully associative to determine hits and offset bits are used to index into a block (cache line)   Direct Mapped Caches   * 1-way set associative cache * set # = (Physical Address) % (# of sets) * **Pathological programs that target the same set will cause a lot of misses 🡪 Bad**   Fully Associative Caches   * 1 set, N way set associative cache * No index bits * Theoretically ideal * **However, too much delay, area on chip, power usage, and energy consumption.**   N-Way Set Associative Caches   * More logic when compared to direct mapped cache 🡪 more area and more power. Less than fully associative cache 🡪 compromise * Set associative caches faster than fully associative due to less delay. * Overall has less utilization than a fully associative, but more than a direct mapped cache. | |  | | | | |
|  | | | | | | |
| 4-way set associative cache with 256 sets | | | | | | General Cache Facts   * **K-bit** physical address, **M** cache lines per way, **B** bytes per line * # offset bits **=** log2­(**B**) * # index bits **=** log2(**M**) * # tag bits **= K** – (# offset + # index) * Types of Cache Misses * **Cold/Compulsory Miss**: cache is empty the first time you access * Fewer cold misses will occur with larger cache lines. * **Conflict Miss**: occurs when you index into a set that is full and you must evict a way to make space for your data. * Fewer conflict misses with more associativity cache * **Capacity Miss**: cache too small to hold all of the blocks you would like to access at any given time. Causes conflict misses. * Fewer with larger cache sizes. * All capacity misses are conflict misses but not the other way around. |
| Replacement Strategies   * **Least Recently Used (LRU)**: Replace least recently used. Best strategy besides pseudo LRU. * **Not Most Recently Used (NMRU)**: Mark the one that is most recently used, randomly replace any other. * **Round Robin**: Pick the next way to replace based on a round robin approach * **Random:** only good if random memory accesses are occurring * **FIFO:** Only good if you access memory locations only once because the first thing placed is the first thing replaced. |
| True LRU   * How many bits per set? **Klog2(K)** bits per set where **K** is the number of ways/associativity * **Log2(K)** bits per block (cache line) in the cache. * **Implementation:** LRU stack/queue 🡪 remember the adder method and only add to valid ways in a set. | | | Pseudo LRU   * How many bits per set? **K-1** bits per set where **K** is the number of ways/associativity * Given a tree with **K** leaf nodes, there are **K – 1** non-leaf nodes. * Example: Assume that leaf nodes are A, B, C, D in that order. * Assume sequence of accesses C, D, A, B, A, C, B, D * AB/CD bit = LRU0, A/B bit = LRU1, C/D bit = LRU2 * The configuration of the tree after each access would be [1, -, 0], [1, -, 1], [0, 0, 1], [0, 1, 1], [0, 0, 1], [1, 0, 0], [0, 1, 0], [1, 1, 1] | | | |
| Write Policy:   * Write-through: for every write to cache, write that cache line to main memory. Write-back buffer makes it faster. * Write-back: Only write the cache line back to main memory on replacement of a dirty line. * Read/Write Allocate: the block is loaded into the cache on a miss and is then followed by a hit action. * Read/Write No-Allocate: if there is a miss, the processor interacts with main memory directly and does not allocate space in the cache. * Write-through is typically used for L1 cache and Write-back is typically used for L2, L3, etc. caches(?) | | | | | | |
|  | | | |  | | |
| Cache + VM:   * VIVT: fastest, but has synonym and homonym issues * PIPT: most secure with no synonym and homonym issues, but slowest * VIPT: Faster than PIPT but slower than VIVT. No homonym problem but there could be a synonym problem. (BEST ☺) * PIVT: Slow and has both problems (WORST ☹) | | Synonym and Homonym:   * Synonym: when two virtual addresses index to the same physical address/sets unintentionally. * Homonym: when one virtual address indexes to two different physical addresses/sets unintentionally. 🡪 Only happens if it is virtually tagged * In order to ensure that the same physical address wouldn't be loaded twice into the cache, you must have your line offset + index bits fit into the page offset bits of your VA, that way the VPN won’t be a part of your index bits 🡪 Gets rid of aliasing. * Side Note: **Size of PA VA** | | | | |
|  | | | | | | |