|  |
| --- |
|  |
| Dennard Scaling: power vs performance increase   * 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 – complex fetch/decode, smaller binary programs (efficient DRAM and cache usage) * Fixed – larger binary programs (inefficient DRAM and cache efficiency) simple pipeline, and easy fetch/decode   **Register Count**:   * Small number of registers – Few bits to specify which, less hardware, faster access, faster context switch * Large number of registers – Fewer memory accesses, faster pipelining   **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) * + | | | | | | |
| 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. | | Cache Design Considerations: 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? | | | | | | | | | |
| 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. | | | | | | | | | |
| 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. | | | | | | | | | | 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. | |
|  | | | | | | |  | | | | |
| 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** | | | | | | | | | |
| **Tomasulo**  • Op: Opcode  • Qj, Qk: The reservation stations producing source operands  • Vj, Vk: The values of source operands  • Busy: Indicates if reservation station is occupied | | | | http://wiki.expertiza.ncsu.edu/images/thumb/6/6c/MESInew.jpg/400px-MESInew.jpg**Top:** perspective from our requests  **Bottom:** perspective of others requests | | | | | | | http://wiki.expertiza.ncsu.edu/images/thumb/d/d0/MSInew.jpg/600px-MSInew.jpg**Left:** perspective from our requests  **Right:** perspective of others requests |
|  | | | | | | | | | | | |
|  | | | | | | | | |
| https://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/MOSI_Processor_Transactions.png/287px-MOSI_Processor_Transactions.png | | | | | | | | | | | |
| BusRdx = ownGetx = otherGetx (own if it is current processor, other if someone else).  **Left:** perspective from our requests | **Right:** perspective of others requests | | | | | | | | | | | |
| https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MOESI-Zustandsdiagramm_f%C3%BCr_aktive_CPUs.png/500px-MOESI-Zustandsdiagramm_f%C3%BCr_aktive_CPUs.pnghttps://upload.wikimedia.org/wikipedia/commons/thumb/0/0a/MOESI-Zustandsdiagramm_f%C3%BCr_passive_CPUs.png/500px-MOESI-Zustandsdiagramm_f%C3%BCr_passive_CPUs.png  https://upload.wikimedia.org/wikipedia/commons/thumb/1/12/Legende_MOESI.png/220px-Legende_MOESI.png**SNOOPING = BUS OPS | Left:** perspective from our requests | **Right:** perspective of others requests | | | | | | | | | | | |
| **Cache Coherence:**   * MSI: Has a lot of bus access and write backs * MESI: Better for bus accesses, but has a lot of write backs * MOSI: Better for write backs, but has a lot of bus access * MOESI: Better for both, but has larger state machine overhead | | | | | |  | | | | | |
| **Data Dependencies:**   * True dependency: RAW (Read after Write) * Anti-Dependency: WAR (Write after Read) * Output dependency: WAW (Write after Write) * Control dependency: an instruction execution is dependent of a branch | | | **Branch Predictors**   * Correlating predictor: uses history of branches to index into a table, which corresponds to the branches address. This will have 1 or 2 bits saturating counter, which will predict. (Two images on left) * Tournament predictor: a predictor that chooses between two other predictors to use. One is usually global and one is usually local. (Image on right). | | | | | | | | |
| https://upload.wikimedia.org/wikipedia/commons/thumb/c/ce/Two-level_branch_prediction.svg/420px-Two-level_branch_prediction.svg.pnghttp://www-ee.eng.hawaii.edu/~tep/EE461/Notes/ILP/Figs/predict_corr.gifhttps://scontent.xx.fbcdn.net/v/t34.0-12/17793426_1486341448083588_1338754619_n.png?oh=d761fa3a88e087aedaa27e43d0130fdc&oe=58E59112 | | | | | | | | | | | |
| Software ILP:   * Instruction scheduling: reordering instructions to reduce dependencies * Loop unrolling: unrolling a loop, allows multi-issue processing. * Prologue and epilogue might be needed * Register renaming: removes anti-dependencies * Software pipelining: achieves similar effect of loop unrolling without all of the code expansion. | | | | | | | | Pipeline: How large  ts = stage delay | Ni = No of instructions | T = time of execution  to = latch delay | a = avg degree of superscalar processing  p = No pipeline stages | tp = time between pipe stages?  https://scontent.xx.fbcdn.net/v/t34.0-12/17793261_1319333451481199_1768174323_n.jpg?oh=9ecabf3bf6b0ac58177774b1dc5a6961&oe=58E6C4B7  T = Tbz + Tnbz  Ts = tp/p + to  Tbz(busy) = NiTs =Ni(tp/p + to) = (Ni/a)(tp/p + to)  Tnbz = NhTpipe = Nh(tp + pto) = Nh(tp + pto)(1/Nh ([summation of all Nh]Bh))  T/Ni (time per instruction) = (1/Ni)[Tbz + Tnbz] | | | |
| Power:     * Dynamic power (≈ 40 - 70% today and decreasing relatively) * Short-circuit power (≈ 10 % today and decreasing absolutely) * Leakage power (≈ 20 – 50 % today and increasing) * Static power grows proportionally to the transistor count * Dynamic power is proportional to the product of the number of switching transistors and the switching rate * Area, power = O(k\*n/k) * Performance = O(k\*sqrt(n /k)) | | | | | | | |
| Superscalar and Other Shit:   * **CMP**: Chip that provide multiprocessing, called Multi-core Architecture. There are many cores per processor.   • **SMT**: Simultaneous Multithreading. There are many threads per core. They run together within multiple issue slots. Context switch is not required. This is because there are multiple context regs/reg files  • **CMT**: Chip Multithreading has support to CMP Technology and SMT Technology. There are many cores with multiple threads per processor. | | | |
| GPU:   * CPU: Latency oriented, large caches * GPUs: Throughput oriented, small caches * Amdahl’s Law: If fraction X of a computation is serialized, the speedup cannot be more than 1/X * If X = 50%, then half the calculations are serialized. No more than 2X speedup, no matter how many computing cores are used | | | |
| Memory Models:   * A **consistency/memory model** is an “agreement” between the execution environment (H/W, OS, middleware) and the processes. Runtime guarantees to the application certain properties on the way values written to shared variables become visible to reads. This determines the memory model, what’s valid, what’s not. * **Coherence** is the memory model in which (the runtime guarantees to the program that) writes performed by the processes for every specific variable are viewed by all processes in the same full order. * **Program Order**: The order in which instructions appear in each process. This is a partial order on all the instructions in the program. * A **serialization**: A full order on all the instructions (reads/writes) of all the processes, which is consistent with the program order. * A **legal serialization**: A serialization in which each read X returns the value written by the latest write X in the full order.   Let P be a program; let PX be the “sub-program” of P which contains all the read X/write X operations on X only.   * **Coherence**: P is said to be coherent if for every variable X there exists a legal serialization of PX (Note: a process cannot distinguish one such serialization from another for a given execution) * **Sequential Consistency** is the memory model in which all reads/writes performed by the processes are viewed by all processes in the same full order. P is said to be sequentially consistent if there exists a legal serialization of all reads/writes in P. * **Release Consistency: (A)**Before a read or write access is allowed to perform, all preceding (program order) acquire accesses must be performed, and **(B)** Before a release access is allowed to perform, all preceding (program order) read or write accesses must be performed, and **(C)** acquire and release accesses are sequentially consistent. | | | | | | | | | | | |
| Image result for directory based cache coherence | | | | | | | | | | | |

s