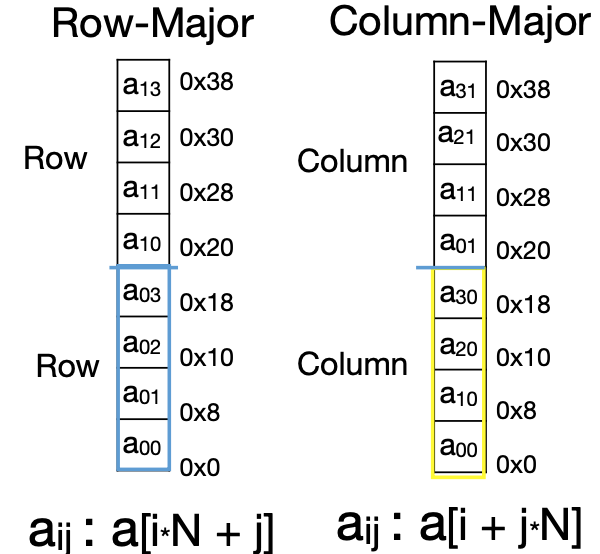
­LEC 18: Parallel Processing SIMD (Single Instruction/Multiple Data)

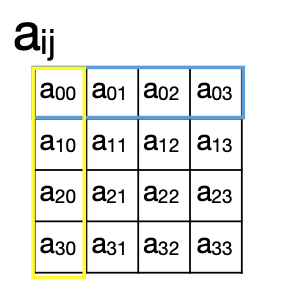
Reference Problem: Matrix Multiplication

* Basic operation in many engineering, data, and imaging processing task, core operation in Neural Nets and Deep Learning
* **Dgemm:**
  + double-precision floating-point general matrix-multiply

****

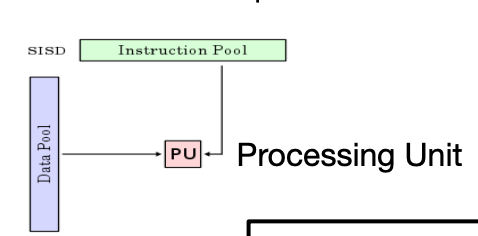
2D matrix Memory Layout:

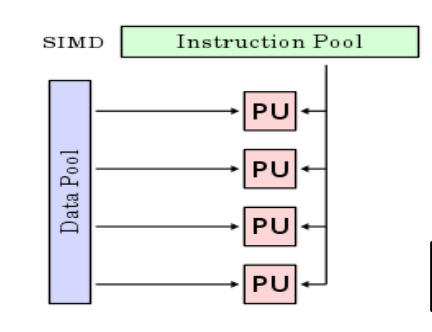
* **We’re using Column-Major**



Why Parallel Processing:

* CPU Clock Rates are no longer increasing
* Parallel processing is the only path to higher speed
* 2 basic approaches to parallelism:
  + **Multiprogramming**
    - Run multiple independent programs in parallel (easy)
  + **Parallel computing**
    - Run one program faster (hard)

Single-Instruction/Single-Data Stream (SISD) Single-Instruction/Multiple-Data (SIMD)

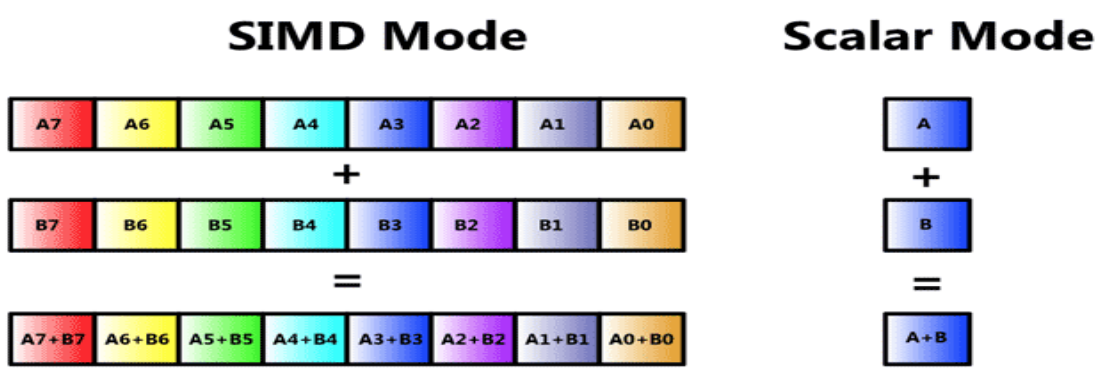


Flynn Taxonomy:



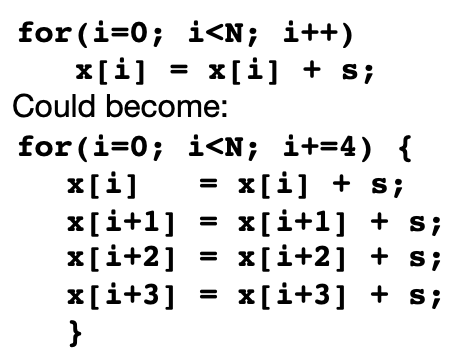
* SIMD and MIMD most common parallelism in architectures—usually both in same system

SIMD (Vector) Mode:



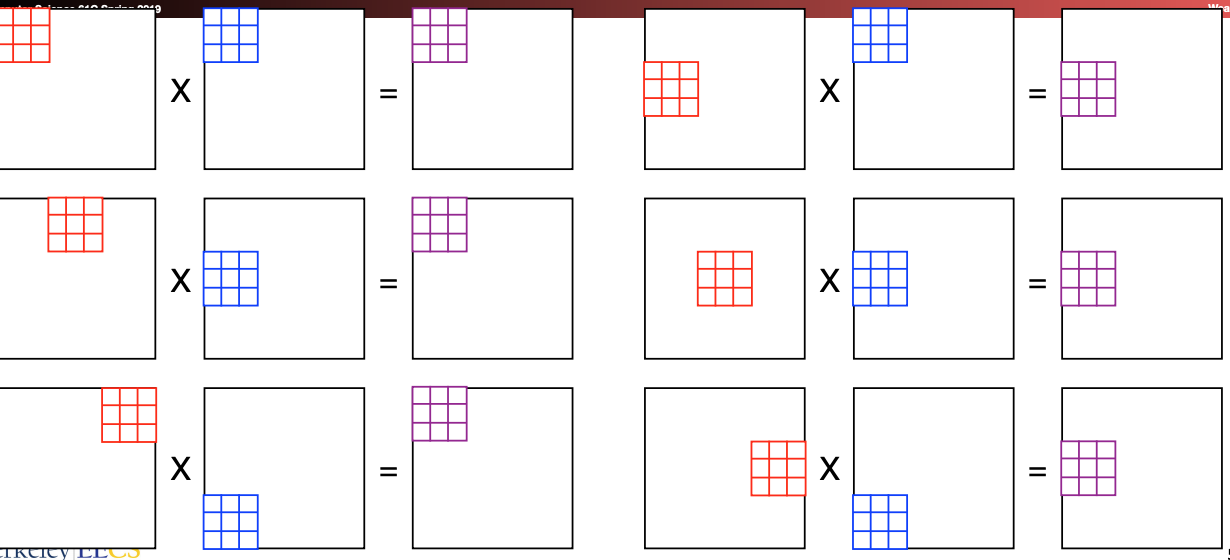
Intrinsics: Direct access to assembly from C

* Example:
  + \_m256 (256-bit as eight single-precision floating-point values)
  + \_m256d (256-bit as four double-precision floating-point values), etc.
* **Nomenclature**
  + **s/d** = single-or double-precision floating point
  + **[i/u]nnn** = signed/unsigned of bit size nnn (128, 64, 32, 16, 8)
  + **[ps/pd/sd]** = packed single, packed double, scalar double
  + **epi32** = extended pacekd 32-bit signed integer
  + **si256** = scalar 256-bit integer

Loop Unrolling:

* On high performance processors, optimizing compilers performs “loop unrolling” operation to expose more parallelism and improve performance:

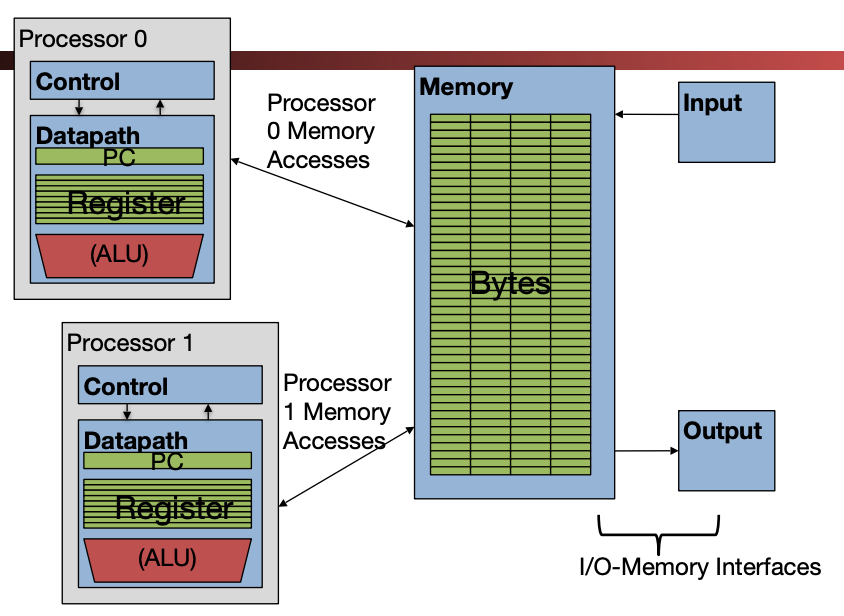
Blocking:

* **Idea:**
  + ****Rearrange code to use values loaded in cache many times
  + Only “few” accesses to slow main memory (DRAM) per floating pt operation
    - Throughput limited by FP hardware and cache, not slow DRAM

LEC 19: Amdahl’s Law, Thread Level parallelism

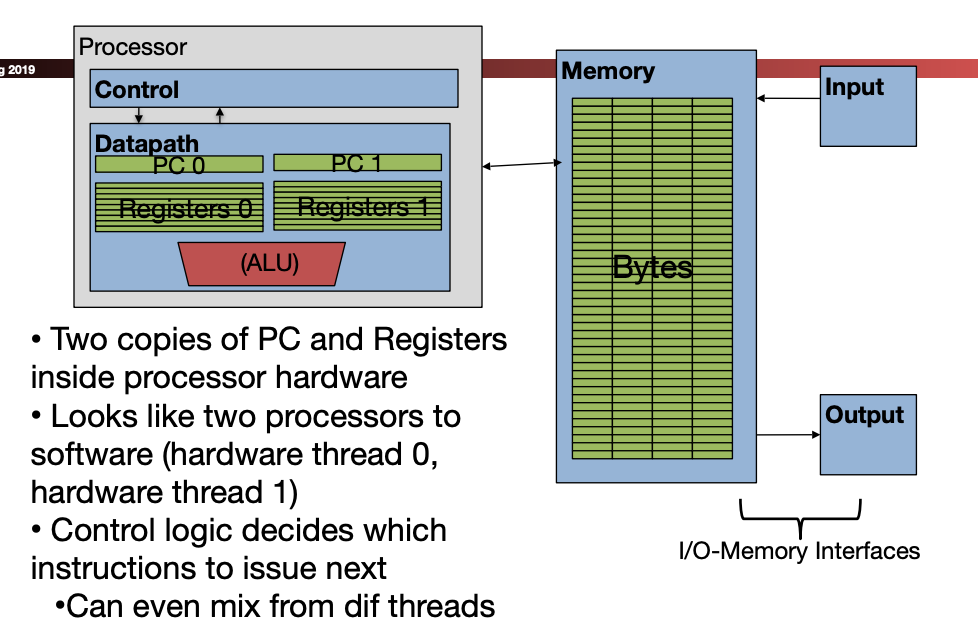
To get good speedup on a parallel processor while keeping the problem size fixed is harder than getting good speedup by increasing the size of the problem

* **Strong Scaling:** 
  + when speedup can be achieved on a parallel processor without increasing the size of the problem
* **Weak Scaling:** 
  + when speedup s achieved on a parallel processor by increasing the size of the problem proportionally to the increase in the number of processors
* **Load Balancing** is another important factor,
  + every processor doing the same amount of work

Multiprocessor (Multicore) Execution model:

* Each processor has its own PC and executes an independent stream of instructions (MIMD)
* Different processors can access the same memory space
  + Processors can communicate via shared memory by storing/loading to/from common locations
* Two ways to use a multiprocessor:
  + Deliver high throughput for independent jobs via job-level parallelism
    - E.g. your operating system & different programs
  + Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel-processing program

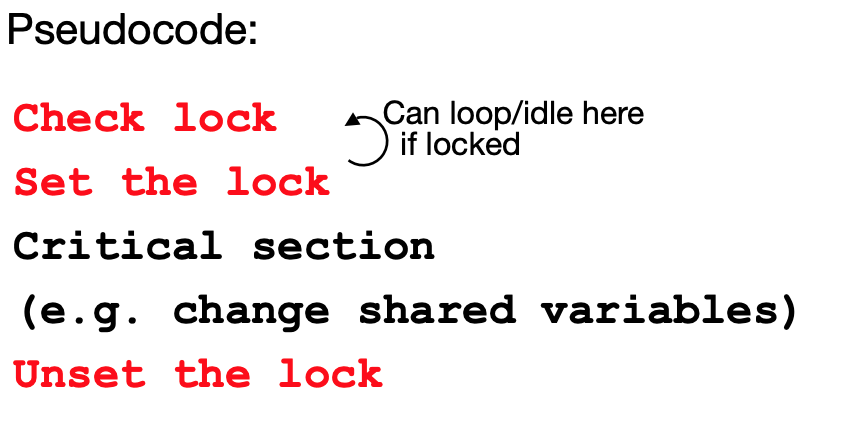
Thread:

* **Thread**:
  + a sequential flow of instructions that performs some task
* Each thread has a PC + processor registers and accesses the shared memory of the process
* Each core provides one or more hardware threads that actively execute instructions
* **Operating system Thread:**
  + Give the illusion of many active threads by time-**multiplexing multiple software** threads onto the available **hardware threads**
  + **Remove** a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  + Make a different software thread **active** by loading its registers into a hardware thread’s registers and jumping to its saved PC
* **Hardware Multithread:**
  + **Basic idea:**
    - Processor resources are expensive and should not be left idle
    - Long memory latency to memory on cache miss is the biggest one
  + Hardware switches threads to bring in other useful work while waiting for cache miss
  + Cost of thread context switch must be much less than cache miss latency
  + Put in redundant hardware so don’t have to save context on every thread switch:
    - PC, Registers

Data Races and Synchronization:

* Two memory accesses form a **data race** if different threads attempts to access the same location, and at least one is a write, and they occur one after another
* Avoid data races by **synchronizing writing** and **reading** to get deterministic behavior
* **Synchronization** done by user-level routines that rely on **hardware synchronization instructions**

Lock Synchronization:

* Use a “Lock” to grant access to a region (**critical section**) so that only one thread can operate at a time
  + Need all processors to be able to access the lock, so use a location in shared memory as the lock
* Processors read lock and either wait (if locked) or set lock and go into critical section
  + 0 means lock is free / open / unlocked / lock off
  + 1 means lock is set / closed / locked / lock on

Hardware Synchronization:

* Hardware support required to prevent an interloper (another thread) from changing the value
  + **Atomic** read/write memory operation
    - No other access to the location allowed between the read and write
    - Take the value **pointed to** by rsi
      * Load it into rd
      * Apply the operation to that value with rs2
      * Store the result back to where rs1 is **pointed to**

LEC 20: Synchronization and Threading

Instruction-Level Parallelism:

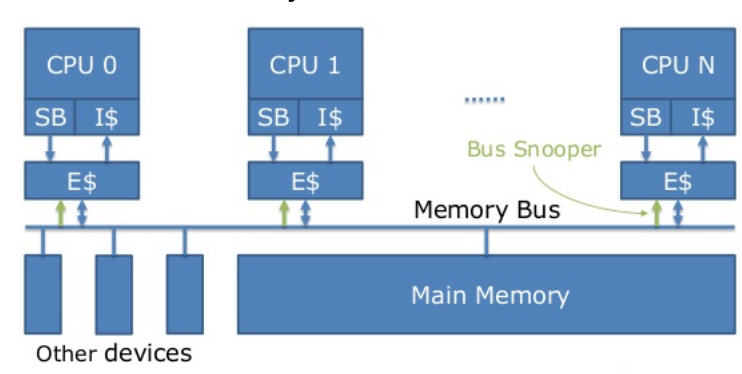
* Multiple instructions in execution at the same time, e.g., instruction pipelining
* **Superscalar**: launch more than one instruction at a time, typically from one instruction stream
* ILP limited because of pipeline hazards

Thread Level Parallelism:

* **Thread**: sequence of instructions, with own program counter and processor state (e.g., register file) but common memory within a process
* **Process**: consists of at least one but could be arbitrary number of threads
  + Each process has its own memory space

Deadlock:

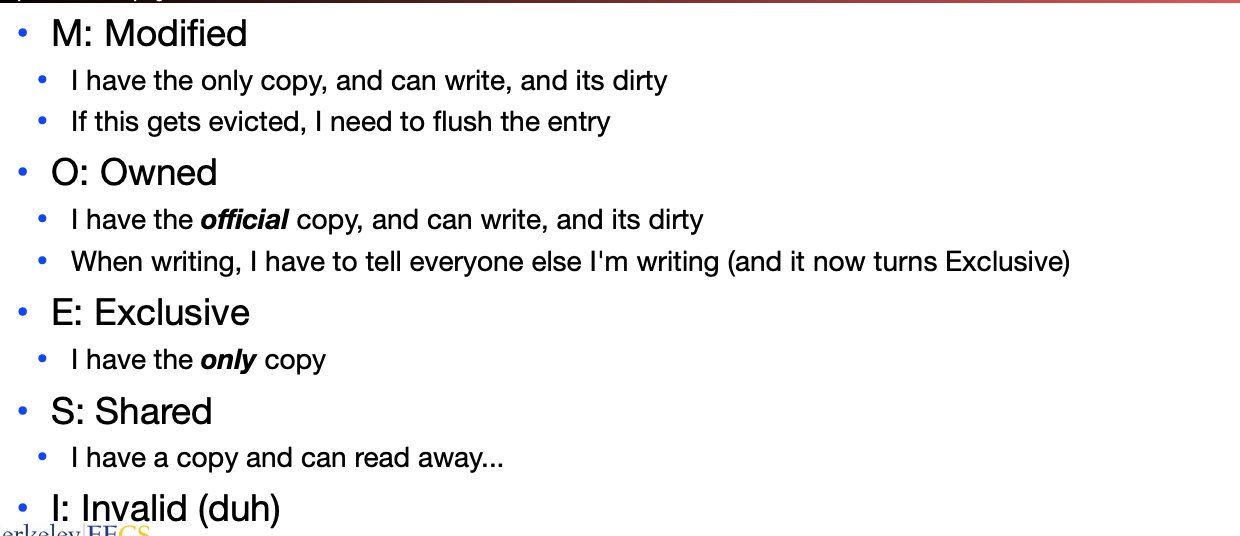
* System state in which no progress is possible because everything is locked waiting for something else
* Locks act to inhibit parallelism (innately sequential regions 🡪 Amdahl’s law problem …)

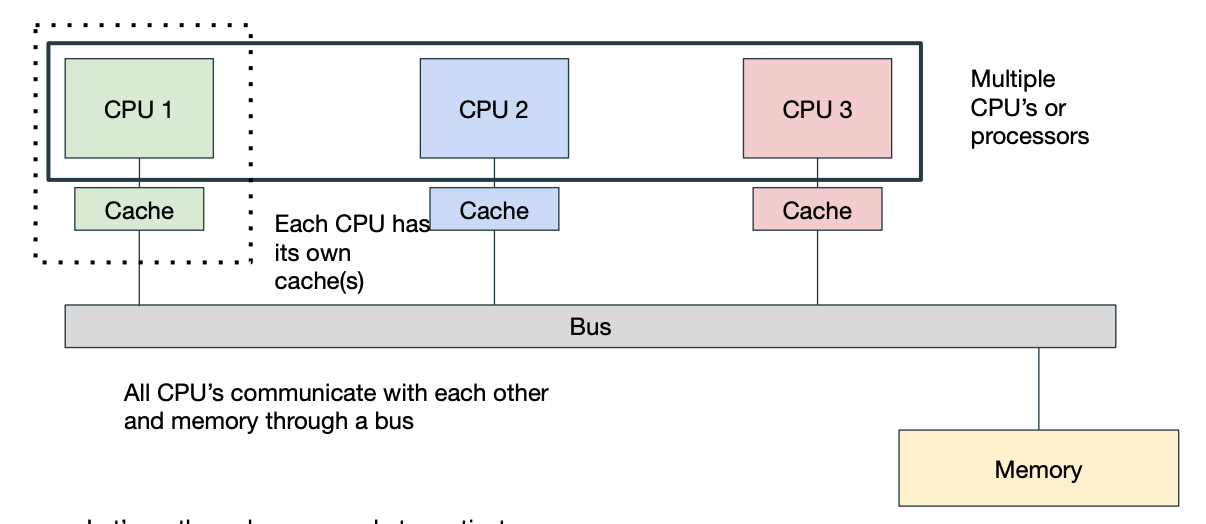
Multicore Multiprocessor

* SMP (Symmetric Multiprocessor)
  + 2 or more identical CPUs/Cores
  + Single shared coherent memory
* Single address space shared by all processors/cores
* Processor coordinate/communicate through shared variables in memory (via loads and stores)

Multiprocessor Caches:

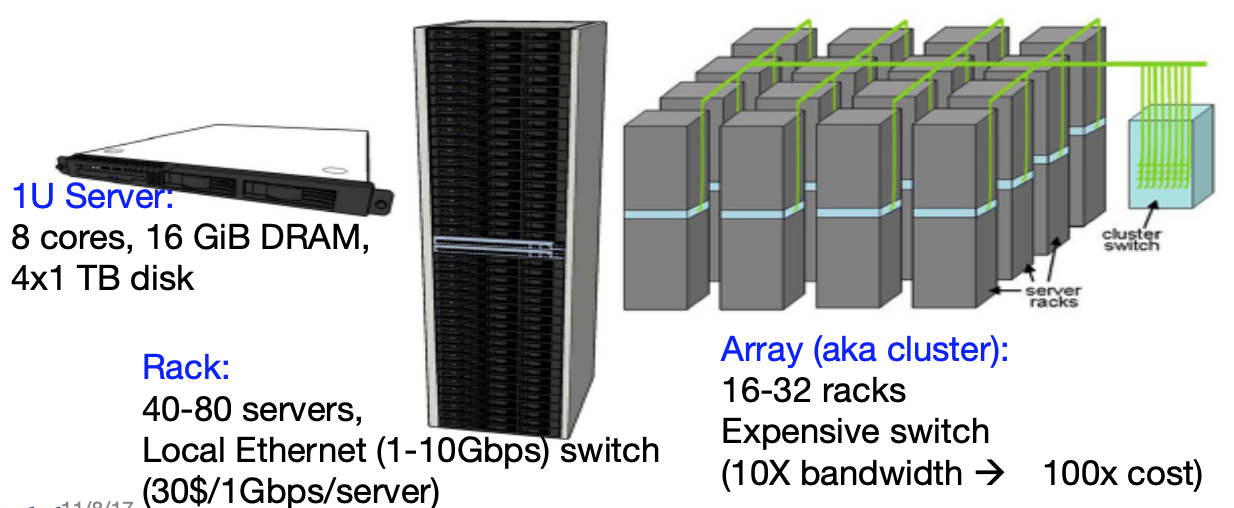
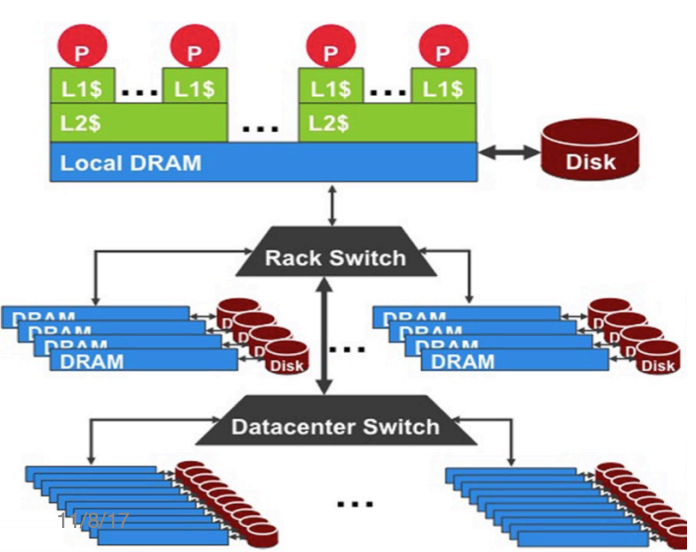
* Memory is a performance bottleneck even with one processor
  + Use caches to reduce bandwidth demands on main memory
  + Each core has a local private cache holding data it has accessed recently
  + Only cache misses have to access the shared common memory
* **Keeping Multiple Caches Coherent (shared memory)**
  + When any processor has a cache miss or writes, notify other processors via an interconnection network (**BUS)**:
  + Valid and Dirty, now **MOESI:**
    1. **Shared** 
       - up-to-date data
    2. **Modified** 
       - up-to-data data, changed/dirty, no other cache has a copy, OK to write, memory out-of-date
    3. **Exclusive**
       - Up-to-date data, no other cache has a copy, OK to write, memory up-to-date
    4. **Owner**
       - Up-to-date data, other caches may have a copy (they must be in **shared** state)
    5. **Invalid** (not in cache)
  + If only reading, many processors can have copies
  + If a processor writes, invalidate any other copies





* Want to use write-**back** caches
* Want to minimize writes overall
* We can communicate by broadcasting requests

LEC 21: Warehouse Scale Computing

WSC Architecture: Storage Hierarchy:

Cloud Computing Distinguished by:

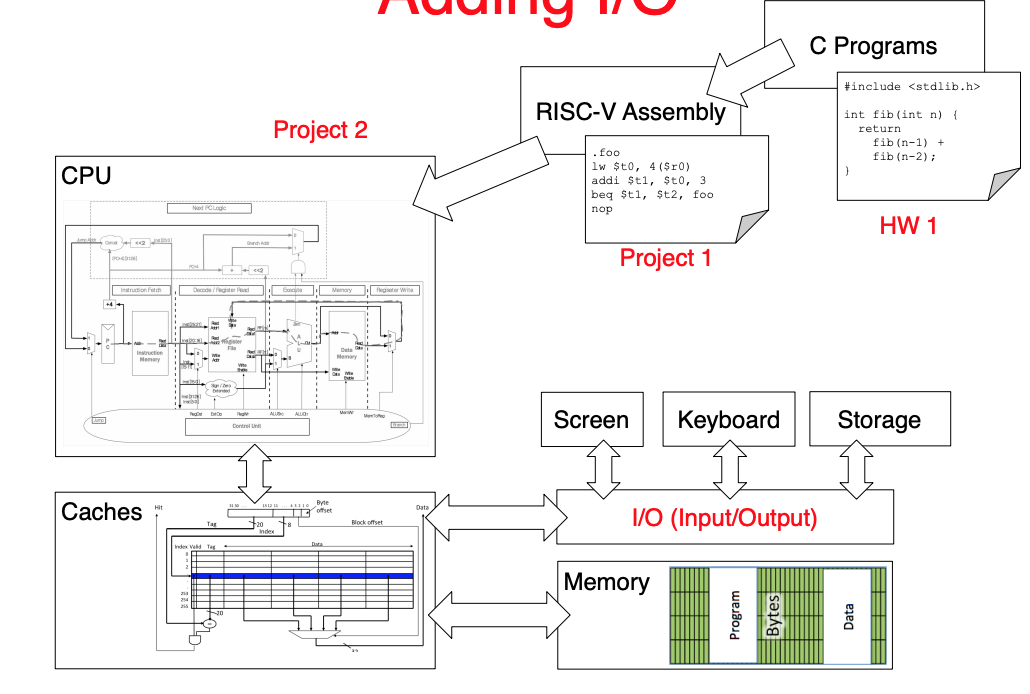
* Shared platform with illusion of isolation
* Attraction of low-cost cycles
* Elastic service
  + Pay for what you need, get more when you need it
* Services:
  + **SaaS:** deliver apps over Internet (Google Docs)
  + **PaaS:** deliver computing stack as a service, using cloud infrastructure to implement apps (Apache Spark on GCP)
  + **IaaS:** Rather than purchasing servers, software, data center space or net equipment, clients buy resources as an outsourced service (Amazon Elastic Compute Cloud)

Request Level Parallelism:

* Hundreds of thousands of requests per second (google search, social networks)
  + Often involve read-mostly databases
  + Rarely involve read-write sharing or synchronization across requests
* **Challenge:** Parallelize computation, distribute data, tolerate faults without obscuring simple computation with complex code to deal with issues
* **Solution: MapReduce**
  + Simple data-parallel **programming model** and **implementation** for processing large datasets
  + Users specify the computation in terms of a **map function** and a **reduce function**
  + **Map**
    - Divide large data set into pieces for independent parallel processing
  + **Reduce**
    - Combine and process intermediate results to obtain final result

LEC 22/23: Operating System and Virtual Memory

Adding I/O



Operating Systems:

* Just software
  + First thing that runs when computer starts (and intermittently runs as long as computer is on)
* Finds and controls all I/O devices in the machine in a general way
  + Relying on hardware specific “device drivers”
* Starts services (100+)
  + File system
  + Network stack (Ethernet, WiFi, Bluetooth, …),
* Loads, runs and manages programs:
  + Multiple programs at the same time (time-sharing)
  + Isolate programs from each other (isolation)
  + Multiplex resources between applications (e.g., devices)

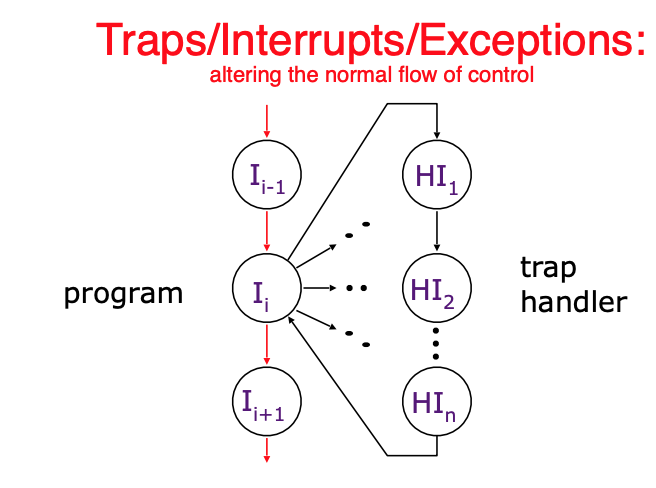
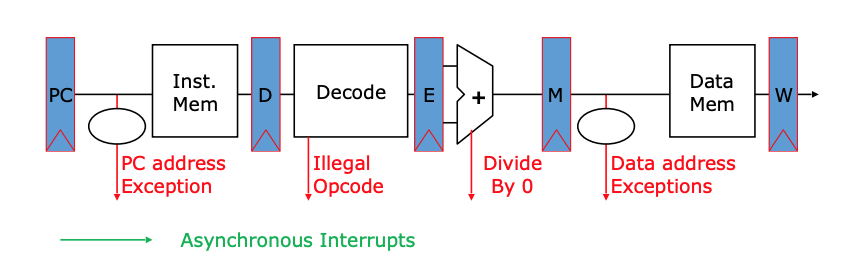
I/O Interface for Keyboards, Network, Mouse, Screen, etc

* **Input**: read a sequence of bytes
* **Output**: write a sequence of bytes
* **Memory mapped I/O**
  + Portion of address space dedicated to I/O
  + I/O device registers there (no memory)
  + Use normal load/store instructions (e.g. lw/sw)
  + Very common, used by RISC-V

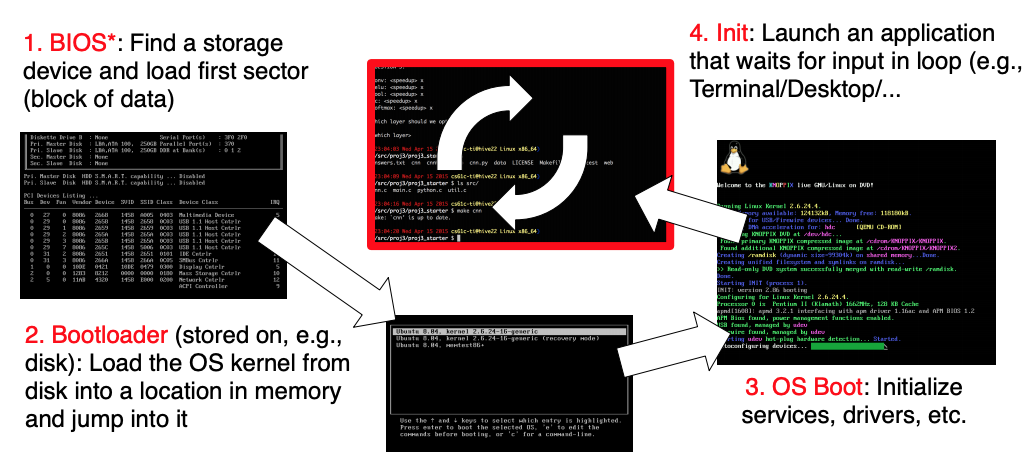
Polling:

* Device Registers generally serve 2 functions:
  + **Control Register** says its OK to read/write (I/O ready)
  + **Data Register** contains data
* Processor reads from Control Register in loop
  + Waiting for device to set **Ready** bit in Control reg (0 ! 1)
  + Indicates “data available” or “ready to accept data”
* Processor then loads from (input) or writes to (output) data register
  + I/O device resets control register bit (1 ! 0)

Alternative to Polling: **Interrupt Handler**

* Polling wastes processor resources (Akin to waiting at the door for guests to show up, what about a bell?)
  + Occurs when I/O is ready and needs attention (interrupt current program)
* **Interrupt**
  + caused by an event external to current running program (E.g., key press, disk I/O)
  + **Asynchronous** to current program
  + Can handle interrupt on any convenient instruction
* **Exception**
  + caused by some event during execution of one instruction of current running program (E.g., divide by zero, bus error, illegal instruction)
  + **Synchronous**
    - Must handle exception precisely on instruction that causes exception
* **Trap**
  + action of servicing interrupt or exception by hardware jump to “interrupt or trap handler” code
  + Trap Handling in 5-Stage Pipeline

What happens at Boot?



Launching Applications:

* Application are called “processes in most OS”
  + **Thread:** shared memory
  + **Process:** separate memory
  + Both threads and proceses run (pseudo) simultaneously
* Apps are started by another process (e.g. shell) calling an OS routine (using a syscall)
* To help protect the OS from the application, CPUs have a **supervisor mode** (set by a status bit in a special register)
  + Process can only access a subset of instructions and (physical) memory when in **user mode**
  + Process can change out of supervisor mode using a special instruction, but not into it directly – only using an interrupt

Syscalls:

* Calling an OS routine (read a file, launch a new process, ask for more memory, send data)
* Perform a **syscall**
  + Set up function arguments in registers
  + Raise **software interrupt (with special assembly instruction)**
* OS will perform the operation and return to user mode (OS mediates access to all resources and devices)

Multiprogramming:

* The way the OS runs “multiple applications at the same time” is by switching between process very quickly (this is called a **context switch**)
* When jumping into process, set timer interrupt
  + when it expires, store PC, registers, etc. (process state)
  + Pick a different process to run and load its state
  + Set timer change to user mode, jump to new PC
* Deciding what processes to run is called **scheduling**

Supervisor Mode:

1. **Read & write to all physical memory**
   1. nables supervisor mode to conduct I/O and change any state controlled by magic memory addresses
2. **Manage interrupts** 
   1. Prevents supervisor mode from being interrupted by anything else
3. **Adjust virtual memory mappings** 
   1. Manages the isolation between programs

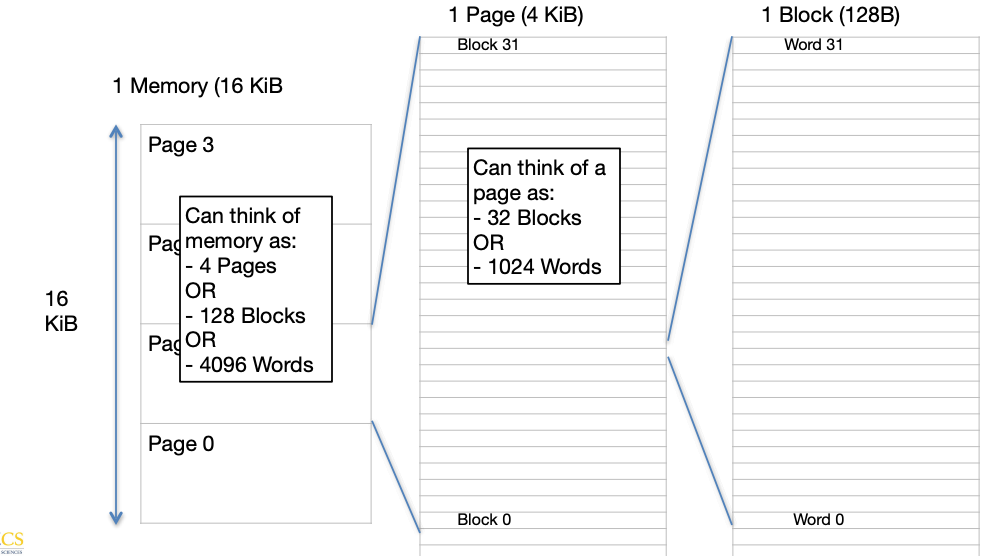
Virtual Memory: Why do we need it?

**Gives each process the illusion of a full memory address space that is has completely for itself**

1. Adding disks to memory hierarchy
2. Simplifying memory for apps
3. Pro­­tection between processes

Address spaces:

* Set of addresses for all available memory locations
  + **Virtual Address space**
    - Set of addresses user program knows about
    - In the form of **pages**
      * Facilitates virtual 🡪 physical address translation
      * Provides isolation & protection
      * Extends available memory to include disk
  + **Physical address space (DRAM)**
    - Set of addresses that map to actual physical locations in memory
    - Hidden from user applications

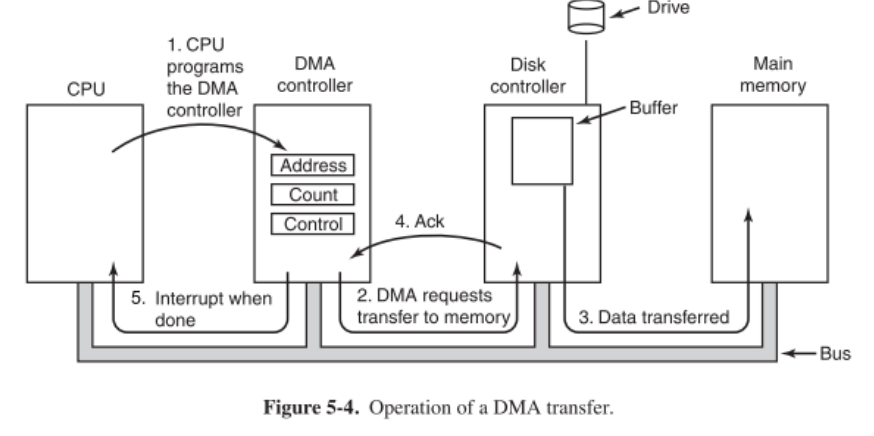


LEC 24: Input/Output

I/O is asynchronous

* **Option 1:** Trigger an interrupt
* **Option 2:** Wait for the OS to poll the device

Direct Memory Access (DMA)

* Allows I/O devices to directly read/write to main memory
* New Hardware: **DMA Engine**
  + Memory address to place data
  + # of bytes
  + I/O device #, direction of transfer

LEC 25: Dependency, Redundancy, and RAID

Dependability via Redundancy:

* Applies to everything from data centers to memory
  + Redundant data centers so that can lose 1 datacenter but Internet service stays online

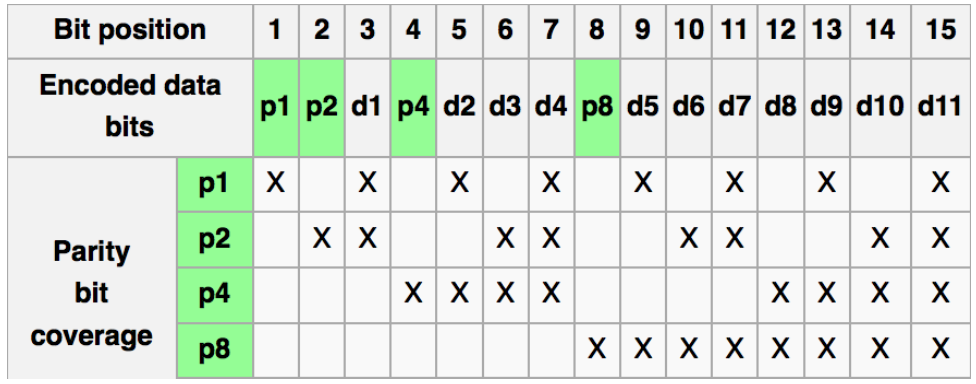
Redundancy:

* **Spatial Redundancy:**
  + replicated data or check information or hardware to handle hard and soft (transient) failures
* **Temporal Redundancy:**
  + redundancy in time (retry) to handle soft (transient) failures

Dependability Measures:

* **Reliability**: Mean Time To Failure
* **Service interruption:** Mean Time To Repair
* **Mean Time Between Failures:** MTBF = MTTF + MTTR
* **Availability =** MTTF / (MTTF + MTTR)
  + **Increase MTTF:** More reliable hardware/software + Fault Tolerance
  + **Reduce MTTR:** improved tools and processes for diagnosis and repair

Block Code Principles:

* **Hamming Distance =** difference in # of bits
* Graphic of Hamming Code

RAID: Redundant Arrays of (Inexpensive) Disks

* Files are striped across multiple disks
* Redundancy yields high data availability
* Disks will still fail
* Contents reconstructed from data redundantly stored in array
* **RAID 0: Striping**
  + Not actually RAID, just spreading the data across multiple disks
* **RAID 1: Disk Mirroring/Shadowing**
  + Each disk is fully duplicated into its “mirror”
  + Most expensive solution: 100% capacity overhead
* **RAID 5:** 
  + Interleave Parity across the disk (result of RAID 3 and 4 ideas)

