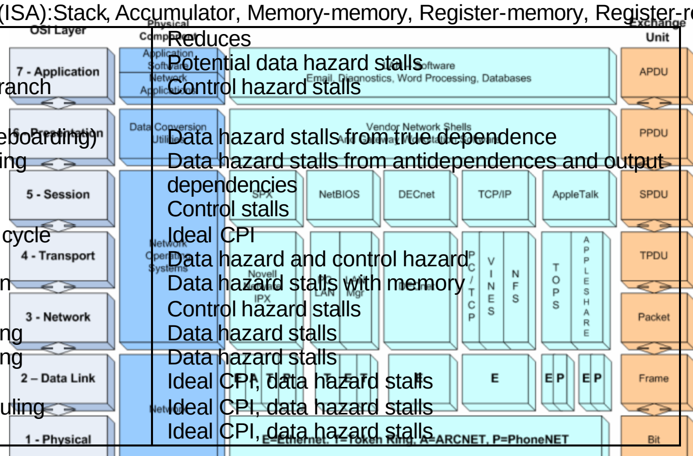
Architecture:Processor Architecture

Instruction Set Architectures (ISA):Stack, Accumulator, Memory-memory, Register-memory, Register-register **Instruction level parallelism**

Technique	Reduces
Forwarding and bypassing	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependence
Dynamic scheduling with renaming	Data hazard stalls from antidependences and output dependencies
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data hazard and control hazard
Dynamic memory disambiguation	Data hazard stalls with memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI, data hazard stalls
Software pipelining, trace scheduling	Ideal CPI, data hazard stalls
Compiler speculation	Ideal CPI, data hazard stalls

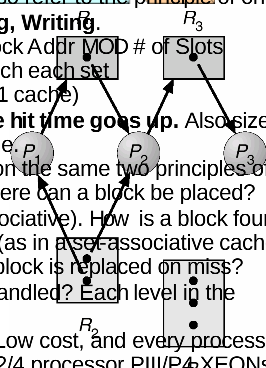


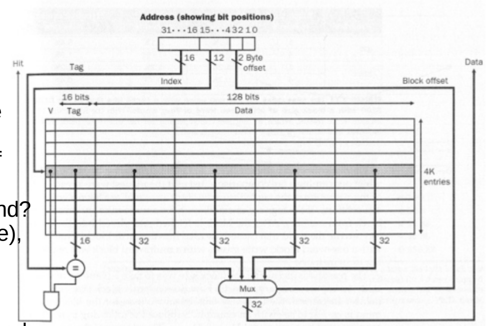
WAR - **name**/anti-dependence
WAW- **name**/output dependence does not exist in classic 5-stage pipeline
Name dependencies can be fixed using register renaming
RAW - true data dependence See table for resolution

Scalar pipeline – Single issue per clock cycle
Superscalar pipeline – Multiple instructions issued per clock cycle → Tomasulo pipeline
Superpipelining – Deeper pipeline that decomposes memory accesses from 5→8 stages
VLIW (Very Long Instruction Word 64-bit)
Performance enhancement techniques: Branch predictions, Prefetching, Out-of-order executions

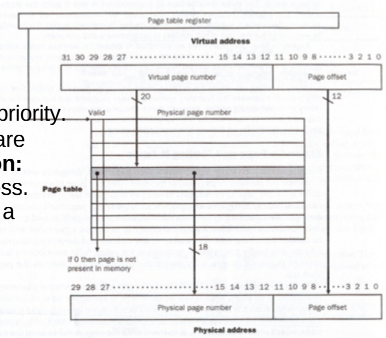
Memory Hierarchies

Locality: Temporal, Spatial
Principle of exclusion: (*Cache coherence protocol*) says that one processor has exclusive access to a block of memory in its cache (May also refer to the principle of one process having exclusive access to a block).

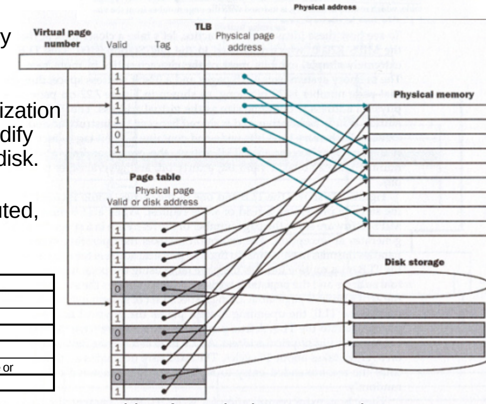




The four fundamental issues for data in cache blocks: **Placing, Replacing, Finding, Writing**.
Direct Mapped: Block Address = (Byte Address) DIV (Bytes Per Block) , slot# = Block Addr MOD # of Slots
Set-Associative: Set = (Block Address) modulo (Number of sets in the cache), search each set
Fully Associative: Place anywhere, LRU usually, search whole cache (to slow for L1 cache)
KEY: as you increase set associatively the miss rate goes down, however the hit time goes up. Also size of the cache and *n* are not independent when determining performance of the cache.
While caches, TLBs, and Virtual memory may initially look very different, they rely on the same two principles of locality and can be understood by looking at how they deal with four questions: Where can a block be placed? One place (**Direct Mapped**), a few places (set associative), or any place (fully associative). How is a block found? There are four methods: **Indexing** (as in a direct mapped cache). **limited search** (as in a set-associative cache), **Full Search** (as in a fully associative cache), and a **separate lookup table.** What block is replaced on miss? Typically, either the LRU (least recently used) or a random block. How are writes handled? Each level in the hierarchy can use either WriteThrough or WriteBack.
Cache coherence: Multiple caches and coherence protocols. Snooping Protocol: Low cost, and every processor has a mechanism to monitor a common bus – problem: it doesn't scale well – used on 2/4 processor PIII/P4 XEONs. Directory protocols have a centralized repository of information about what blocks are in use, and whether they are shared, or exclusive. This information is sometimes distributed to different processors, but blocks are assigned to repositories much like they are assigned to sets in a cache.



I/O: Basic Bus protocols: arbitration: Daisy chain: requests are sent down a chain from highest priority to lowest priority. Starvation is possibly for the lowest priority, but it is simple! **Centralized:** You have a central arbitrator that requests are queued to and it determines which is first. (used in CPU – memory busses) **Distributed arbitration by self selection:** Each device places a code identifying itself on the bus. All devices must be able to participate in the arbitration process. **Distributed arbitration by collision detection**: Ethernet uses this. Buss protocols are made up of a specification of a sequence of events and **timing** requirements in transferring information.
I/O methods: **Programmed I/O** (The program busy waits for the device to respond be ready for a request), **Polling** (The CPU or OS occasionally polls the devices to see if they have responded or are ready for a request), **Interrupts** (The devices themselves interrupt the processor thus relieving the CPU from wasting time checking on the status of the devices), **DMA** (A separate controller moves data into specific memory locations and interrupts the processor when the requested block or blocks have been placed in the memory location requested). The I/O bus is usually connected to the memory bus so usually we communicate with peripherals using memory mapped I/O. **IOPs**: Input/output operations per second.
RAID 0: striping blocks **1:** Disk Mirroring **2:** Memory style ECC organization **3:** bit-interleaved parity organization Down side is every disk must be involved in a read/write. **4:** Block-interleaved parity organization: read modify Write problem exists for parity disk. **5:** Block-interleaved distributed parity solves the overuse of the parity disk.



Operating Systems

Types of systems: Batch, Multi-programmed, Time-sharing, Personal-Computer Systems, Parallel, Distributed, Real-Time systems.
System Calls: require context switch, used for I/O operations and other protected operations.

Process Management Process Control Block→

Pointer	Process state (New, running, waiting, ready terminated)
Process number	
Program counter: address of next instruction	
CPU Registers	
CPU-scheduling information: **priority**, pointers to queues, parameters	
Memory-management information: **base** and **limit registers**, **page table** or	
List of open files	

Process contains the code, PC (program counter), Processor's registers, process stack (temporary data such as methods parameters, return addresses and local variables as well as a data section which contains global variables.
Process scheduling: Ready Queue holds all processes waiting to execute. **Device or I/O Queue** holds processes waiting for a device to complete an operation like a disk access. **Long-Term Scheduler**: selects which jobs to load into memory for execution. It controls the amount of multiprogramming. **Short-Term or CPU Scheduler**: selects from among process that are ready to execute, and allocated the CPU to one of them.
Cooperating processes: Bounded/Unbounded Shared Buffer between **Producer and Consumer**. **IPC** using message passing. *Direct* (duh) *Indirect*: processes communicate via ports/mailboxes that they *share*. Who receives if multiple receivers? Depends on the scheme used. A mailbox may have an owner or it may be owned by the OS. If the mailbox has an owner only the owner can receive messages and user processes can only send messages. When the owner terminates the mailbox disappears too. **Synchronization:** blocking → synchronous, non-blocking → asynchronous. When both receiver and sender are blocking, we have a **rendezvous**. Messages are queued. If the queue is *zero length, the sender must block*, this is often called a no-buffering system. The same happens with a bounded queue that is full. Bounded or infinite buffering systems are called **automatic buffering**.
RPC: in contrast to IPC facility, the messages exchanged for RPC are well structured and are thus no longer just packets of data. A **Stub** is provided on the client side and when invoked the RPC system invokes the appropriate remote procedure. Parameters are **marshalled** by packaging the parameters into a form which may be transmitted over a network. A similar stub on the server side receives this message and invokes the procedure on the server. If necessary return values are passed back to the client using the same technique. Binding to a port can be either static or dynamic. Dynamic port binding provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port. **RMI** works similarly except the server side stub is called skeleton.
Threads: A thread is a flow of control within a process. Benefits include (Responsiveness, resource sharing, economy over proc ess creation, utilization of multiprocessor architectures). **Types user** (implemented in a library above the kernel and the kernel is unaware of the scheduling issues etc. The can be created fast and are easy to manage, but if one thread makes a blocking system call, all the threads will be blocked.) **Kernel** (The kernel creates, schedules and manages

the threads. Since they are created by system calls they are a little slower, but the kernel can schedule other threads even if one thread performs a blocking system call, and the kernel can schedule different threads on different processors in a multiprocessor system.) **Mapping User threads to Kernel threads: Many-to-one** (this is essentially user threads, and you have the blocking problem – precludes the use of multiple processors – *true concurrency is not achieved*) **One-to-one** (Much higher overhead, but true concurrency is achieved, the user must be careful not to create to many threads and in some cases the number of threads is restricted – used by NT/2000/OS/2) **Many-to-many** (Multiplexes many user threads with many kernel threads – it does not suffer from either of the problems of the other two types.) **Fork** calls may either duplicate the thread that called it or all the threads (depending on the fork used), but exec works the same, replacing all threads with the processed called. **Signal Handling:** When a <ctrl+c> signal is sent to a processes which thread handles it? The kernel has a default handler, but this may be overridden by a user handler. Synchronous signals need to be sent to the thread that generated it, but asynchronous signals are less clear: 1. Deliver it to the tread to which it applies (e.g. I/O signal) 2. Deliver the signal to every thread in the process (e.g. ctrl+c) 3. Deliver the signal to certain threads in the process. 4. Assign a specific thread to receive all signals for a process. **Thread Pools**: Usually faster to service a request, limits the number of threads that exist at any one point. Solaris 2 Threads: Two types bound and unbounded – bound thread have a 1-to-1 relationship with a light-weight thread (LWP). Unbounded threads is not permanently attached to a LWP and more than one one user-level thread can be attached to an LWP. Many-to-many model. LWPs also have a many-to-many model with processes.

CPU SCHEDULING: scheduling criteria(**CPU utilization:** busy as possible, ideally 40-90%, **throughput**: Number of processes completed in unit time, **turnaround time:** the interval from the time of submission of a process to the time of completion including wait time and executing. **Wait time**: waiting in the ready queue and i/o queue. **Response time:** the time from the submission of a request until the first response is produced – generally limited by the speed of the output device.

Scheduling algorithms: **FCFS -** Simple but often times *bad average wait time which we call the convoy effect* where one process dominates the CPU and other line up behind it. **Shortest Job First (SJF)** should be called shortest next CPU burst first is provably optimal in that it gives the minimum average waiting time for a given set of processes. However, it can't be implemented at the level of short-term CPU scheduling because we don't know how long the next CPU burst will be! Used for long-term scheduling. It can be approximated using a exponential average: p_{n+1}=at+(1-a)p_n where t is the last value and p_{n+1} is the new prediction based on the last prediction. May be either *preemptive (sometimes called shortest-remaining time first) or non-preemptive*. SJF is a specific type of **Priority Scheduling:** Can use an internal priority like SJF does or some external priority assigned to a process. It can be either *preemptive or non-preemptive*. A major problem with priority scheduling is indefinite blocking or *starvation*. A solution is aging where over time a process gets a higher and higher priority. **Round-Robin Scheduling:** is similar to FCFS, but preemption is added to switch between processes. Still has long average wait times. Time slice should be large with respect to the context switch time, performance depend on time slice size as well small=processor sharing – large = FCFS queue type. **Multi-level Queues:** Have two ready queues and a scheduling scheme for the queues (e.g. interactive is RR and background is FCFS) and then a priority scheduling algorithm to determine which queue to take jobs from. Subtype: Multilevel feedback queues are characterized by: 1) The number of queues 2) The scheduling algorithm for each queue 3) The method used to determine when to upgrade a process to a higher priority queue 4) Method used to determine when to demote a process 5) Method used to determine which queue a process will enter when that process needs service. **Little's formula** n=?*W where ? is the average arrival rate of new process and W is the time we expect a process to wait.

PROCESS SYNCHRONIZATION The critical-section (CS) problem(Sections: Entry, critical, exit and remainder): Mutual Exclusion (If P_i is executing in its critical section, then no other process can be executing in their critical section problem.) **Progress** (If no process Pi is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.) Bounded Waiting(There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.) Hardware testandtest(sets to true and returns value before set. Testandset(lock) if the CS is not locked go in and lock it, then unlock it in the exit code. The swap works very similar we have key=true, swap(lock,key) if key is false, then we "no longer have our key, it's in the door and we have exclusive access to the CS, when we leave we take our key and set the lock=false (not occupied). **Semaphores**: typedef struct{int value; struct process *L;} semaphore; semaphore S; NOTE L has a pointer to another process. So that we can use L as a queue of some sort. The following is your typical counting semaphore. Void wait(semaphore S) { S.value–; if(S.value<0) { add this process to S.L; block();}} Void signal(semaphore S) {S.value++; if(S.value<=0) {remove a process from S.L; wakeup(P);}} **Deadlocks**: happen when two process are waiting (via semaphore) to access a resource that another process is holding. **Necessary conditions**: **Mutual Exclusion, Hold and wait, No preemption, Circular wait. Prevention** by devising a protocol that breaks one of the necessary conditions: ME: Breaking ME is not reasonable because some resources can not be shared. H&W: a process must request all need resources before it executes or must release all resources before requesting additional ones. This may cause starvation! NP: Allowing preemption works well for some resources like CPU that can have their state saved and restored quickly, but won't work at all for things like printers. CW: We could define a 1-to1 mapping of resources to natural numbers and allow processes to request only resources with higher numbers than those it holds. This hierarchy will not allow deadlock, because it breaks Circular wait. NOTE This may cause low utilization of resources. **Avoidance**: Deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist. The system is in a safe state if there exists an ordering of the processes such that the requests for resources can be satisfied. We deny requests that lead to unsafe states. **Bankers/Safty Algorithm: 1.** Let *Work* and *Finish* be vectors of length m and n respectively (m- number of resource types, n- is the number of processes). Initialize *Work:=Available* and *Finish[i]=false*. **2.** Find an *i* such that both *finish[i]=false* and *Need[i]<=work*. If no such I exists, go to step 4. **3.** *Work:=Work + Allocation[i]* and *Finish[i]:=true; goto step 2.* **4.** If Finish[i]=true for all I, then the system is in a safe state. **Resource Request Alg.:** Let Requesti be the request vector for process Pi. If Requesti[j]=k, then process Pi wants k instances of resource type Rj. To grant the request: **1.** If Requesti<=Needi, go to step 2. else, raise an error because we exceeded **max claim**. **2.** If Requesti<=Available go to 3, else Pi must wait since the resources are not available. **3.** Pretend to allocate the resources and check to see if it results in a safe state. **DEADLOCK Detection: 1.** [bankers +] If Allocationi<>0, Finish[i]=false, else Finish[i]=true. **2.** Replace need[i] with request. **3. [bankers] 4.** If There exists finish[i]=false, process i is deadlocked.

```
//Correct Cr Region
Do {
//Entry section
  Flag[i]=true;
  Turn = j;
  While (flag[j] &&
         turn== j);
  //Critical section:code
//Exit Section
  Flag[i]=false
//remainder section
```

```
Bakery Algorithm
do {
  choosing[i] = true;
  number[i] = max(number[0],
number[1],...,number[n-1]) + 1
  choosing[i] = false;
  for (j=0; j<n; j++) {
    while(choosing[j]);
    while((number[j]!=0) &&
((number[j],j)<(number[i],i)));
  }
  //critical section
  number[i] = 0;
   //remainder section
} while(1);
```

Networking

Connection oriented (ATM) and Connectionless Services (TCP) TCP/IP Model = App/TCP/IP/Host-to-Network(Datalink+Physical)

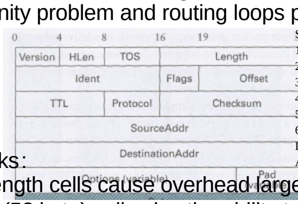
Network Layer 0[Net7][Host24]A,10[Net14][Host16]B,110[21Net][8Host]C,1110[Multicast]D,1111[Reserved]E

Network Layer Design issues: Works, Simple, Clear Standardized choices, Exploit Modularity through layers,Expect heterogeneity, Avoid dynamic parameters, Think about scalability, Performance/Cost. **Routing Algorithms OSPF** is a breadth first search using reliable flooding and Dijkstra's Algorithm. **RIP** uses the link state algorithm in where global information is shared with neighbors. Count to infinity problem and routing loops possible.

CRC: Divide generator polynomial into data using XOR instead of subtraction. Remainder → error. **Bit Stuffing:** remove 0 after 5 1's because 01111110 is a special tag. **Congestion Control Algorithms**

IPv6 128 Bit address space, Minimum Header is only twice as long as IPv4

Provides Stateless Autoconfiguration (prefix + Ethernet Address)

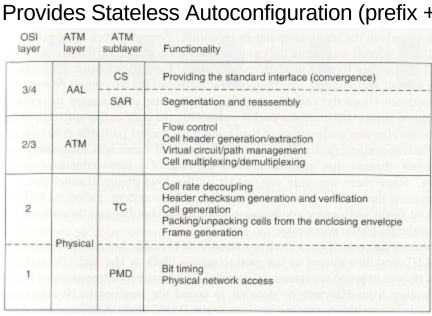


Step	Confirmed	Tentative
1	(A,0,-)	(C,2,C),(B,3,B),(D,5,D)
2	(A,0,-),(C,2,C)	(B,3,B),(D,4,C),(F,3,C)
3	(A,0,-),(C,2,C),(B,3,B)	(D,4,C),(F,3,C),(E,7,B)
4	(A,0,-),(C,2,C),(B,3,B),(F,3,C)	(D,4,C),(E,5,C)
5	(A,0,-),(C,2,C),(B,3,B),(F,3,C),(D,4,C)	(E,5,C)
6	(A,0,-),(C,2,C),(B,3,B),(F,3,C),(D,4,C),(E,5,C)	None

Destination	Next Hop	Cost
A	-	0

Destination	Cost	Next Hop
B	1	B
C	1	C

Stored at Node A	Distance to reach Node A	B	C	D	E	F	G
A	0	1	1	inf	1	1	inf
B	1	0	1	inf	inf	inf	inf
C	1	1	0	1	inf	inf	inf



OSI layer	ATM layer	ATM sublayer	Functionality
3/4	AAL	CS	Providing the standard interface (convergence)
		SAR	Segmentation and reassembly
2/3	ATM		Flow control Cell header generation/extraction Virtual circuit/path management Cell multiplexing/demultiplexing
2	Physical	TC	Cell rate decoupling Header checksum generation and verification Cell generation Packing/unpacking cells from the enclosing envelope Frame generation
1		PMD	Bit timing Physical network access

The Network Layer in ATM networks:

ATM Performance issues: Fixed length cells cause overhead large Overhead for short packets. Small (53 byte) cells give the ability to control delay and especially to control its variation with time(jitter), which can be an important factor for some applications. Since most queues have to wait until a complete packet arrives before you send out another one, small packets also have the advantage of utilizing the link better – less time is spent waiting for that big packet, and subsequent small packets are pipelined.

Speedup = ExecTimeEnanced/ExecTimeWithoutEnhanced

$$Speedup_{overall} = \frac{ExecTime_{old}}{ExecTime_{new}} = \frac{1}{\left((1 - Frac_{enhanced}) + \frac{Frac_{enhanced}}{Speedup_{enhanced}}\right)}$$

Miss penalty = Hit time + retrieval time + [write time]