Chapter 2 - Computer Organization

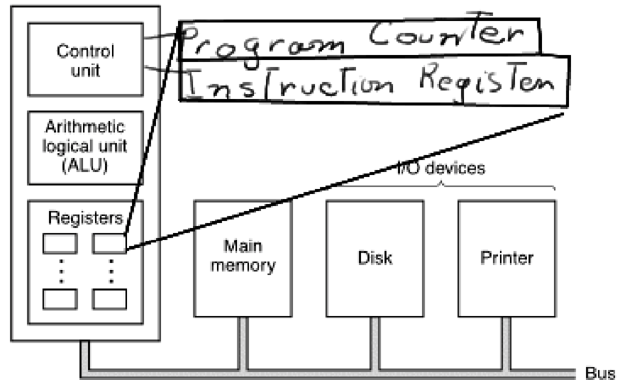
- **CPU organization**
  - **Basic Elements and Principles**
  - **Parallelism**
- **Memory**
  - **Storage Hierarchy**
- **I/O**
  - **Fast survey of devices**
- **Character Codes**
  - **Ascii, Unicode**
- **Homework:**
  - **Chapter 1 # 2, 3, 6; Chapter 2# 1, 2 4 (Due 4/8)**
  - **Chapter 2 #5, 9, 10, 12, 14, 21, 26, 36 (opt) (Due 4/15)**

Chapter 2 is a survey of the basics of computer systems: CPU architecture and overall systems architecture.

Homework: Here is the first homework, along with part of second.

Overall Architecture

- **CPU**
  - **Control**
  - **ALU**
  - **Registers**
  - **Data Paths**
- **Bus (es)**
- **Memory**
- **I/O**



Control unit sends commands to other units to execute instruction in instruction register

Control unit maintains Program Counter, which tells it which instruction to load next.

ALU performs basic data transforms: add, subtract, compare, etc.

Registers hold data - much faster access than main memory.

Memory, disk, I/O devices accessed off bus (sometimes lower speed IO off a subsidiary bus, ISA, PCI, etc.

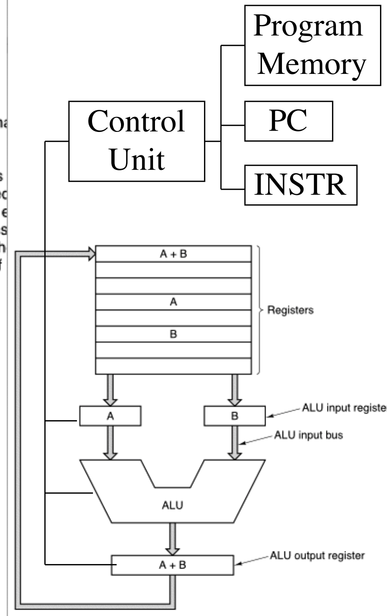
Basic Instruction Execute Cycle

```
public class Interp {
   static int PC;                   // program counter holds address of next instr
   static int AC;                   // the accumulator, a register for doing arithmetic
   static int instr;                // a holding register for the current instruction
   static int instr_type;           // the instruction type (opcode)
   static int data_loc;             // the address of the data, or -1 if none
   static int data;                 // holds the current operand
   static boolean run_bit = true;           // a bit that can be turned off to halt the ma

   public static void interpret(int memory[ ], int starting_address) {
      // This procedure interprets programs for a simple machine with instructions
      // one memory operand. The machine has a register AC (accumulator), usec
      // arithmetic. The ADD instruction adds an integer in memory to the AC, for e
      // The interpreter keeps running until the run bit is turned off by the HALT ins
      // The state of a process running on this machine consists of the memory, th
      // program counter, the run bit, and the AC. The input parameters consist of
      // of the memory image and the starting address.

      PC = starting_address;
      while (run_bit) {
         instr = memory[PC];   // fetch next instruction into instr
         PC = PC + 1;          // increment program counter
         instr_type = get_instr_type(instr); // determine instruction type
         data_loc = find_data(instr, instr_type);// locate data (-1 if none)
         if (data_loc >= 0)        // if data_loc is -1, there is no operand
            data = memory[data_loc];     // fetch the data
         execute(instr_type, data);       //execute instruction
      }
   }

   private static int get_instr_type(int addr) { ... }
   private static int find_data(int instr, int type) { ... }
   private static void execute(int type, int data){ ... }
}
```



Let's look at a simple instruction execution cycle. We've written it is pseudo-java, but most controllers are hardwired to execute a sequence somewhat like this.

(Note: You will be asked to interpret a similar, but not identical, microarchitecture on a simple program in the quiz on Friday).

So - step 1, set the program counter. What's that? It is a register (place that can store data) that holds the address in memory of the next instruction to execute.

Step 2. Check to see if we are running or not - if run bit is false, don't do anything!

Step 3 - get the instruction from the specified location in memory.

Control Design I
Microprogramming

- **Control and Registers are fast but expensive.**
- **Microprogramming!**
  - **Allows a range of price-performance**
  - **Flexibility.**
  - **But - slower than direct execution.**

Control and register access are fast.

That means complex operations can be implemented faster in hardware than as programs.

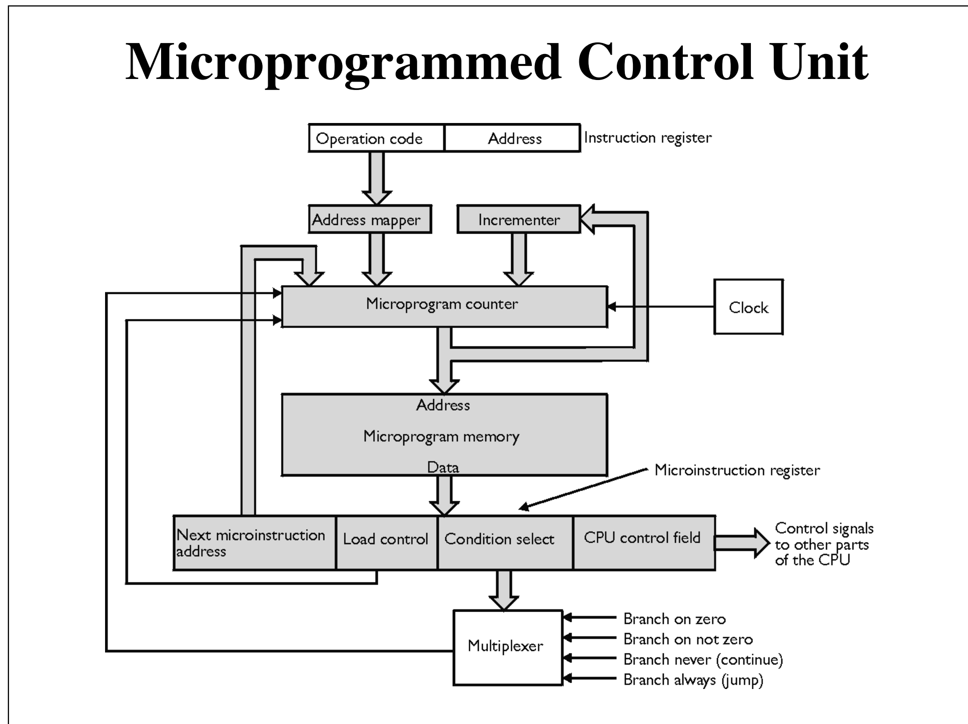
So bigger machines tended to have more complex instruction sets.

But that meant programs weren't transportable!

IBM-360: An ARCHITECTURE, or design for a family of machines all with same basic programmer-accessible instructions.

But how? Microprogramming: lower end machines used invisible "software" to *interpret complex instructions in terms of a simpler hardware*.

Microprogrammed Control Unit



Microprogramming is a way to implement a control unit:

-the control unit is implemented as a Von Neumann computer, with its own control, registers, data-paths, etc.

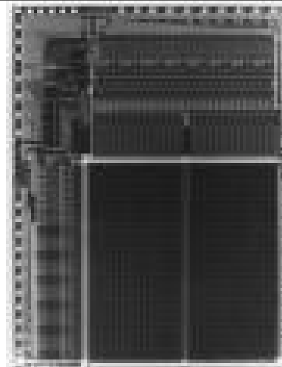
-Where is the "control" in this picture? It is the clock!

-Where are the registers? Instruction register, Microprogram counter, microInstruction register

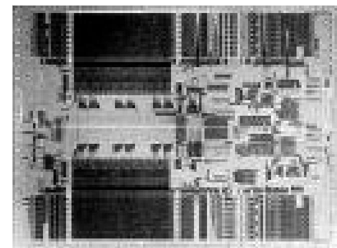
-Where are the operational units? The address mapper and incrementer, and multiplexer?

RISC vs CISC

- **Microprogramming**
  - **Inexpensive complex instructions**
  - **But slow?**
- **RISC!**
  - **Simple instructions**
  - **Direct execution**
  - **Large on-chip register arrays**
  - **Sparc, PowerPC**
- **Modern CPUs?**



RISC



SOAR

Microprogramming allows inexpensive Complex Instruction Set Computers.

But: slow

Lengthy development cycle

How to exploit massive amounts of real-estate?

Reduced Instruction Set Computers!

Simple instructions (all execute in 1 cycle!)

Huge register arrays for deep subroutine call stacks on chip (note large regular areas on Risc chip)

Sparc, PowerPC, …

The answer? Well, maybe not. Modern machines are a hybrid…

Design Guides

- **Direct execution of instructions**
- **Maximize instruction issue rate**
- **Design instructions for easy decode**
- **Only load and store ref memory**
  - **E.g., no add to or from a memory addr directly.**
- **Lots of registers**
  - **Accessing memory is time-consuming.**

Direct execution is faster, plain and simple. At least all simple instructions should be directly executed.

So how do we get speed range? Parallelism of various kinds.

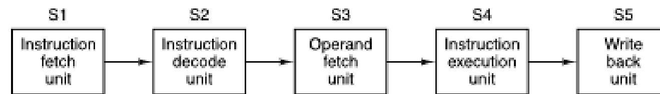
Let’s say it takes 3 ns to execute an instruction. If you can issue one instruction every ns, then you have a 1Ghz processor, even though each instruction takes 3ns to complete!

How is this possible? Well, remember our basic instruction cycle: fetch, decode, execute. Each of these three steps uses (largely) separate parts of the cpu!

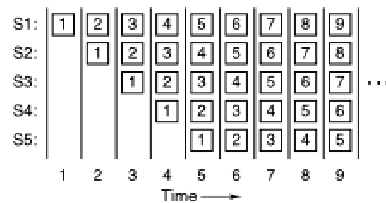
Easy decode helps optimize issue rate - critical step in issue is identifying resources needed (think of those traffic lights at freeway on-ramps, or a greeter at a restaurants - how many in your party?)

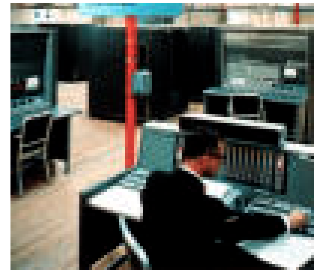
Pipelining

- **5 step instruction cycle**
- **Largely separate hardware for each step**
- **Assembly line!**



(a)





1. Fetching instructions from memory takes a long time, compared to reg-reg add.

2. Why not fetch next inst while executing current one?
   1. Well - what if current is a branch? - where to fetch from?

3. IBM Stretch - 1959, had a "prefetch" buffer to do just that.

4. Generally, can identify several stages in instruction execution, allocate separate hardware to each stage (except in microcoded cpus, most hardware for each stage is dedicated anyway!)

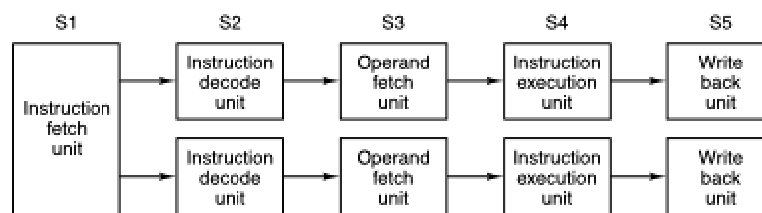
5. Diagram shows how this might work. So, if it takes 2 ns per stage, each instruction will take 10 ns to execute, but we will complete 1 new instruction every 2 ns - 500mips!

But notice a branch can take a long time - we won't discover branch untill step 4, and wil have to throw out all the work we had done on three subsequent instructions. Suppose 10% of instructions are branches that are taken - now what is mips rate?

Well, let's look at this:

SuperScalar Architectures

- **Why not two pipelines?**
  - **486 - one pipeline**
  - **Pentium - two pipelines, second limited**
  - **PII - one pipeline**



Why not just one longer or faster one?

Longer - we saw problems of deep pipelines

Faster - may not have the technology

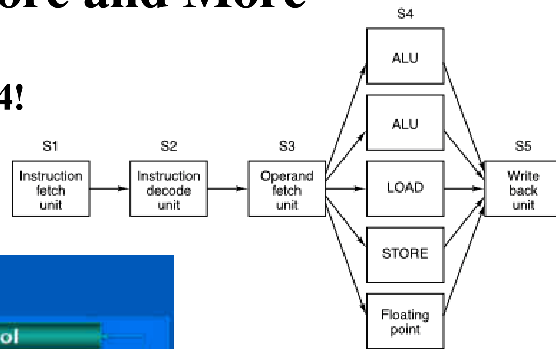
Anything else we can do? Well, if we have enough bandwidth to memory to sustain twice the instruction issue rate, why not two pipelines?

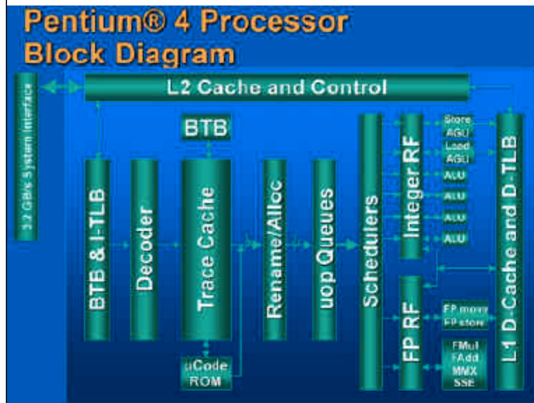
But how to coordinate? Dispatch instruction to second pipeline only if compatible

Special compiler hacks to optimize generation of code that is compatible (remember this!)

More and More

- **Most time in S4!**





Actually, most of the time was now seen to be spent in S4 - so why not just parallelize that?

As you can see from the picture below, the PIV has a single pipeline - but multiple execute units

Spend some time talking about what pipeline stages do.

Spend some time talking about different execute units.

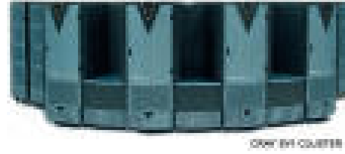
Spend a tiny bit of time talking about cache - kind of like registers - a small, fast memory. But, unlike registers, cache is invisible to ISA programmers!

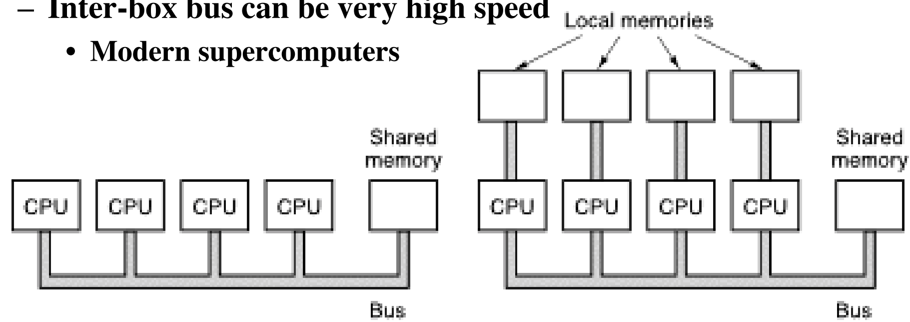
Again, complex coordination needed so instructions don’t step on each other.

Well, so basic idea is to have multiple things going on at once. Can we move this idea up a level? Much of the coordination among multiple instructions is application specific. What if we moved it up to blocks of instructions instead of individual ones, let programmers do it, and used general purpose CPUs as the elements of our parallel structure?

Multiprocessors

- **Several CPUs in a box**
  - **Shared memory and I/O**
  - **Intel, Apple, Sun all support this**
- **Several boxes networked together**
  - **Swarm**
  - **Inter-box bus can be very high speed**
    - **Modern supercomputers**





Two basic kinds of multiprocessors:

Shared memory

Private memory.

Ahead

- **Main Memory**
- **Secondary Memory**
  - **Disks, CDs, etc**
- **I/O**
  - **Buses**
  - **Monitors**
- **Character Codes**

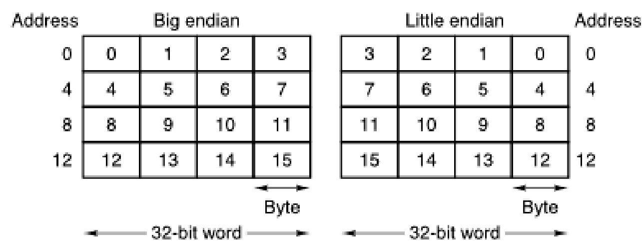
Memory

- **Bits, bytes, words**
- **ECC**
- **Cache**

Memory

- **Bit: 0/1**
- **Which bit?**
  - **Addressing**
- **Word organization**
  - **Big vs little endian**

ADDRESS				
0	0	1	0	0
1	1	0	1	0
2	0	0	0	1
3	0	1	0	1
4	1	1	1	1



Why not three or four or n levels?

The fewer levels the better the noise immunity and error tolerance. Can’t have one level (a memory that can only store “0” doesn’t hold much. So, 2 levels is the next choice.

Why not three levels? Not our problem, go ask the hardware people.

A one bit memory can’t hold much either, so we need to group. But then how do we select one? By ADDRESS. But usually we need more than one bit at a time, so rather than provide addresses at the bit level, they are provided at the “cell” (byte or word) level.

How many bits needed to hold an address to this memory? (3)

Big vs little endian: if a word in memory is bigger than a byte, what order are bytes stored?

Why are “bytes” important anyway? - unit of character encoding and interchange.

Note both store a 4-byte integer as 0001 left to right.

<Do example here>

It’s character and other byte or 1/2 word level data that matters.

Error Correcting Codes

- **Logic level is an abstraction**
  - **Errors happen**
- **Strategy: redundancy.**
  - **Prevent, Detect, Correct**
- **Parity - a 1 bit error detect scheme**
  - **00110011_0 - valid - total number of 1 bits is even.**
  - **00110011_1 - invalid - total number of 1 bits is odd!**
- **Hamming distance**
  - **Number of bit positions that differ**
  - **00110011 vs 01110000 - ?**

Remember the logic level is an abstraction.

Errors happen: noise, power supply fluctuations, timing jitter, ….

Three levels of security:

(0) Prevent problems from occurringl

(1) Detect that a problem has occurred;

(2) Correct it.

Prevention by good design (e.g., only two levels rather than three or four improves noise immunity).

Detection via redundancy (e.g., keep two copies of everything and compare them).

Correction via more sophisticated redundancy - but only with respect to "expected" errors.

Perfection is unachievable.

Let's look at a simple detection scheme: even parity. The idea is we will add one extra bit to each word. If the number of "1" bits in the word is even, we'll add a "0". If the number of one bits is odd, we'll add a one. The idea is the result, in total, will always have an even number of ones (hence the name

Error Correction

- **0000000000, 0000011111, 1111100000, 1111111111**
  - **Hamming distance: ?**
- **0000100000 -> ?**
- **1110011111-> ?**

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Fig. 2-13. Number of check bits for a code that can correct a single error.

Hamming figured out something amazing -

To Detect d single-bit errors, you need a d+1 distance code.

To Detect AND correct d single bit errors, you need a 2d+1 distance code.

Note this doesn't depend on the size of the codeword! Hmm

Example above: Hamming distance 5 (takes 5 single bit changes to get from one codeword to another). That means this can correct ANY one or two bit error!

Huh? Well - suppose you see 0000100000 - what word was it? Only way to get there with two or fewer changes is from 0000000000, so that must have been the actual value.

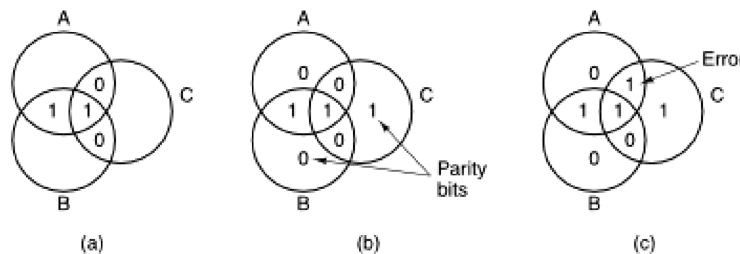
But notice it can DETECT, but not correct, a 3 bit error: it will guess 111001111 1was 1111111111

(of course, it doesn't know how many bits are bad, but it assumes at most two).

In fact, it can DETECT up to 4 wrong bits (d+1=5, d=4)

Single bit error correction

- **4 bit data, 3 bits of error correction (2d+1)**
- **`0011001`**
- **`AB0C123`**



So what does Hamming code for error correction actually look like?

Power of two bits are parity bits., so 1, 2, 4, 8, 16, etc are parity bits

In a seven bit code, 1,2,4 are parity, 3, 5,6,7 are data.

1 is parity for 1,3,5,7,9… (every other one)

2 is parity for 2,3,6,7,10, 11, 14, 15, 20,21… (every other pair, starring with itselt)

4 is parity for 4,5,6,7,12,13,14, 15, 20,21,22,23…(every other quartuple, starting with itself)

(1) Every data bit is covered by at least 1 parity bit

Hence, we can detect every single bit error.

But how do we correct. Let’s look at right set of circles.

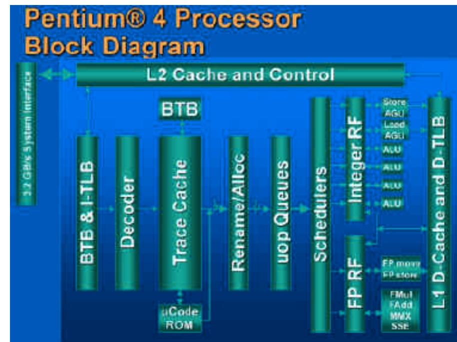
(Explain circles - bits in center are original data bits, bits at outside are parity bits.)

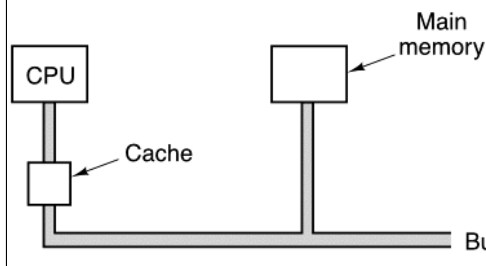
Note in right hand set of circles, error in databit shows up in wrong parity for both A and C. But the only single bit that contributes to both A and C, and not B, is the bit that is in fact the error, so that must be the wrong bit!

Comments: Error correction gets cheaper as the word size increases. Interesting

Cache

- **Locality + invisibility = Cache**
  - **(cacher - to hide, fr.)**
  - **Size?**
  - **Line size?**
  - **Organization?**
  - **Unified?**
  - **Levels?**





What's a register? Really just a small, temporary scratchpad that is visible at the ISA level.

But registers are a pain. You have to worry about how to use them

And they only work for data.

Ah - but we have prefetch buffers for instructions.

So we have temporary fast memory for both data and instructions

What ever happened to stored program concept? Program and data sharing space? (well, maybe keep separate?)

Let's add one more concept: locality.

Locality: the next data/instruction we will touch (read/write) is probably near the ones we are currently reading/writing.

Locality + invisibility => cache memory

Issues in cache memory:

Size: bigger is better, right?

Secondary Storage

- **Memory Hierachies**
- **Magnetic storage**
  - **Disk**
  - **IDE/SCSI/RAID**
- **Optical**
  - **CD/CD-R/CD-RW/DVD**

Memory Hierarchy

- **Registers - Cache - Main Memory**

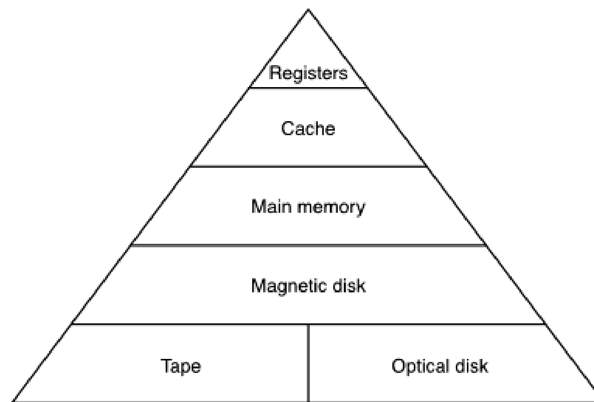


Fig. 2-18. A five-level memory hierarchy.

Registers - 1 ns 128 Bytes

Cache - 10 ns.1 - 10 MB

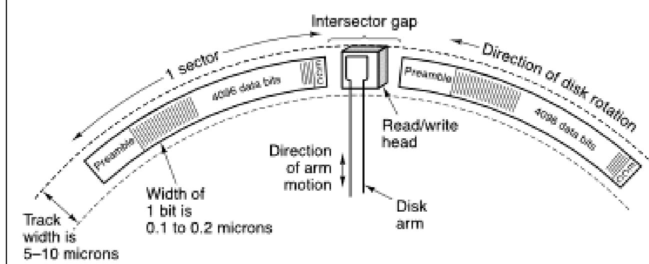
MM - 50 ns 100s-1000s MB

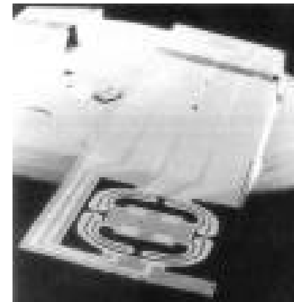
Disk - 10 ms 10s -100s GB

Optical/Tape sec-min 100s - ? MB - removable!

Magnetic Disk

- **Platters coated with magnetic medi**
  - **Magnetism <-> electricity**
- **"head" positioned radially**
- **Data recorded in "tracks"**
  - **Separated into "sectors"**





Fyi - head "fly" over disk surface - they are suspended in air. Huh? Head isn't moving? Right, but it is so close to disk that it is in the airflow of air dragged along with disk.

A head "crash" occurs when the head touches the disk surface - bad.

Why so close? Because mag fields: (1) are weak, decay as d^2; (2) spread out - closer means smaller bits, more density.

Why does mechanics matter? Because it determines access speed:

Slow disks rotate at 5400 rpm, fast at 10,000 or more.

6000 rpm is 100 rps. That means the same spot on the disk passes under the head 100 times a second. So, if the data you want is in the cylinder you are currently positioned over, still can take up to 10 ms to get to it.

If data is in a different cylinder, can take even longer - "head" has to be moved. This can take 5-50 ms, as low as 1ms to an adjacent track.

Note space allocated for error correction - errors happen

Density Growth

- **Bottom line: no slowdown in near future.**

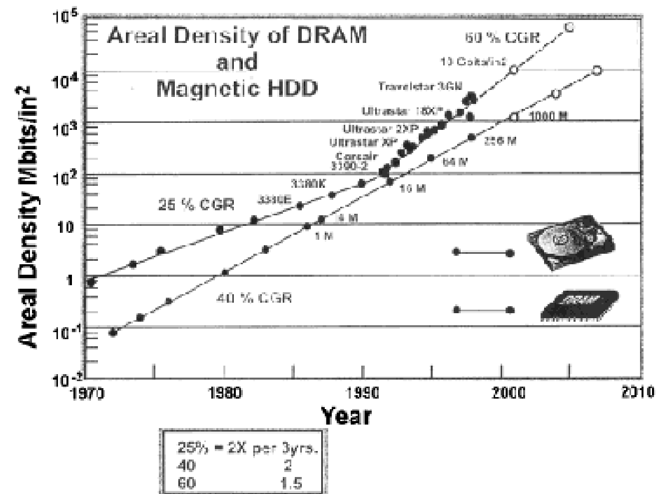


Chart shows “areal” density - how much can be packed into 1 sq inch.

This claims in 2000,:

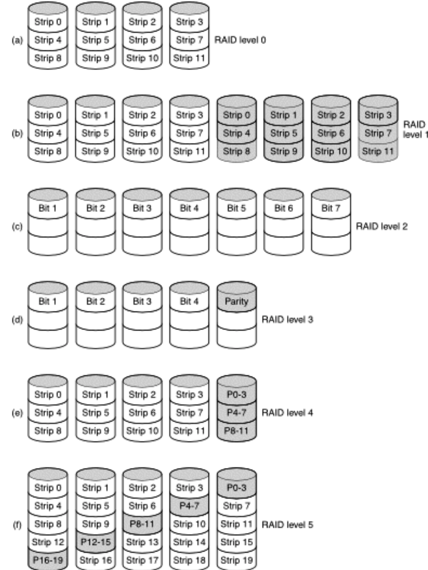
- could put roughly 800 mbits in 1 sq inch of dram chip
- could pack 5000mbits in 1 sq inch of disk drive surface.

This also predicts increase in disk drive capacity by factor of 12 by 2005 - 1 terabyte drive on your desktop!

Also predicts 4Gbits in 1 sq inch of dram - factor of 5 increase over 2000 - 1-2Gb ram on your desktop or laptop!

RAID

- **Redundant Array of Inexpensive Disks**
  - **High performance**
  - **Low cost**
- **Levels**
  - **0 - sector striping**
  - **1 - 0+duplicates**
  - **2 - ECC+bit-level stripping**
  - **3 - bit-stripping+parity**



Raid 0 - High transfer rate for multi-sector requests, but bad reliability (if you lose any one disk you have lost your data!

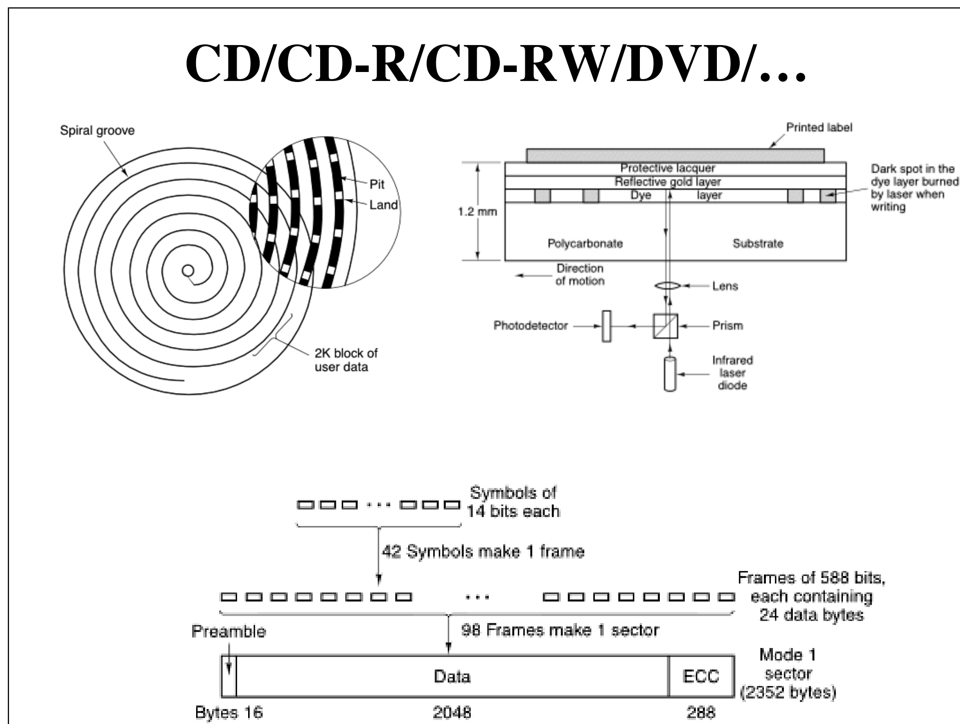
Raid 1 - 0+duplication - now if a disk is lost, have a copy to restore from. Also can distribute read requests over two copies, potentially doubling read transfer rate over RAID 0.

RAID 2 - Complex - take 4 bits, add hamming code, spread over 7 drives. Now can recover from any one disk crash without keeping a complete backup (but only saved 1 disk drive, and made life rather complicated…)

RAID-3 - bit stripping + parity. Note that since we know location of error bit (crashed disk) we CAN do error correction with just a parity bit. So now we have full protection from any single disk crash with only one extra drive!

RADI-4 - Strip level parity on extra disk - why? Because it doesn't require rotational synchronization that 2-3 need

RAID 5 - distribute parity info over all drives via round robin scheme. Why? Because parity is needed on every request, parity disk in RAID4 can be a bottleneck.. Data recovery in RAID 5 is complex.

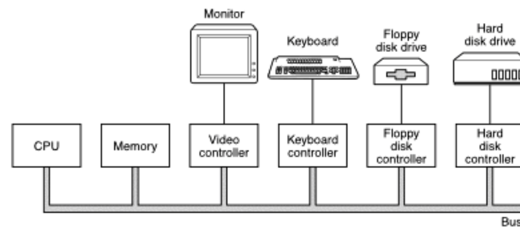


Rotating storage, Optical. Logically organized much like magnetic disks. Major differences:

1. Laser eliminates problem of weak field dispersing over distance, so head can be further from media
2. -> removable media!
3. Access and read/write times are very different. - Write tends to be much harder than with magnetic, and hard to make reversible.
4. Note some music disks now have intentionally incorrect ECC! Why? To prevent copying.

Buses

- ISA, EISA, PCI, AGP, SCSI, Firewire, USB, ….
  - Physical spec
  - Electrical spec
  - Protocol



USB Goals

Easy to use for end user

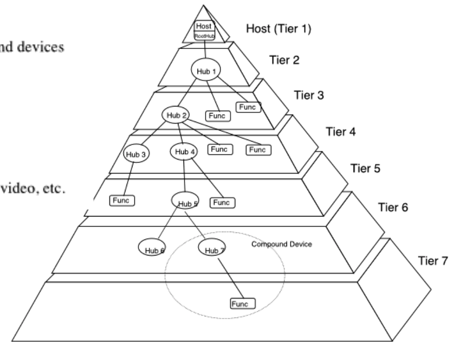
- Single model for cabling and connectors
- Electrical details isolated from end user (e.g., bus terminations)
- Self-identifying peripherals, automatic mapping of function to driver and configuration
- Dynamically attachable and reconfigurable peripherals

Wide range of workloads and applications

- Suitable for device bandwidths ranging from a few kb/s to several hundred Mb/s
- Supports isochronous as well as asynchronous transfer types over the same set of wires
- Supports concurrent operation of many devices (multiple connections)
- Supports up to 127 physical devices
- Supports transfer of multiple data and message streams between the host and devices
- Allows compound devices (i.e., peripherals composed of many functions)
- Lower protocol overhead, resulting in high bus utilization

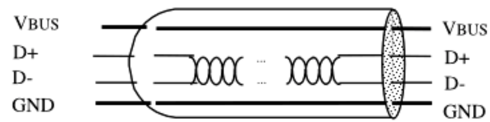
Isochronous bandwidth

- Guaranteed bandwidth and low latencies appropriate for telephony, audio, video, etc.



USB

- **Electrical**
  - **12MB/sec**
  - **480MB/sec**
  - **1.5 MB sec**



Single and double bit error correction

Character Codes

- **ASCII**
  - **7 bit code - 128 characters**

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex
20	(Space)	30	0	40	@	50	P	60	`	70
21	!	31	1	41	A	51	Q	61	a	71
22	"	32	2	42	B	52	R	62	b	72
23	#	33	3	43	C	53	S	63	c	73
24	$	34	4	44	D	54	T	64	d	74
25	%	35	5	45	E	55	U	65	e	75
26	&	36	6	46	F	56	V	66	f	76
27	'	37	7	47	G	57	W	67	g	77
28	(	38	8	48	H	58	X	68	h	78
29	)	39	9	49	I	59	Y	69	i	79
2A	*	3A	:	4A	J	5A	Z	6A	j	7A
2B	+	3B	;	4B	K	5B	[	6B	k	7B
2C	,	3C	<	4C	L	5C	\	6C	l	7C
2D	-	3D	=	4D	M	5D	]	6D	m	7D
2E	.	3E	>	4E	N	5E	^	6E	n	7E
2F	/	3F	?	4F	O	5F	_	6F	o	7F

Ascii - 7 bit code (128 characters)

Summary

- **You should know**
  - **Basic instruction execution cycle**
  - **Basic computer architecture**
  - **Storage hierarchy**
    - **Speed at each level**
    - **Design tradeoffs and challenges at each level**
  - **Map from text <-> ASCII**

What about ε?

- "Escape" characters to shift font
  - 57 68 61 74 20 61 62 6F 75 74 1B 01 45
- UNICODE
  - 16 bit code.
  - Each char or diacritical mark has own "code point"
    - Latin (336 code points)
    - Greek (144)
    - Telugu (128)
    - Katakana (1024)
    - Han (20,992)
    - Hangul (11,156)
- But a decent Japanese dictionary has 50k kanji, only 20,992 Han ideographs allocated. Which ones?
  - Probably not a final answer

What about more than 128 different printable characgters?

Well, could use an "escape" character - a special character reserved for signaling a change in character set. So for example the "1B" hex is the "escape" character in ascii. - when it occurs, the next byte is interpreted as the character set to use for remaining characters until the next escape. That would give us 256 character sets of 127 characters each (can't use escape in any char set!)