1 Byte = 8 bits.
Machine has 2^{32} bytes.
char: 1 byte float: 4 bytes
int: 4 bytes
pointer: 4 bytes.

Memory Leak – memory hasn't been freed.
1 instruction = 32 bits.

Array/Pointer Equivalence.
*(p+3) = p[3]

Return true if most significant byte is different from least significant

((n >> 8) ^ n) & 0xFF

Memory

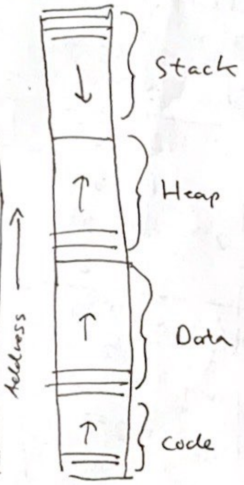
Code - actual code

Data- Static - statically declared data (global) global vars
- strings usually declared here

Stack - local variables
- memory is freed after function call.
- grows downwards
- new data is stored at lesser addresses.

char[] var stored here.

Heap - dynamically allocated data
i.e. malloc'd data
free()



Deadlock - state in which each member of a group of actions waits for some other member to release a lock

Livelock - state of processes involved constantly changes relative to one another. No progress

Concurrency - interweaving of processes to give appearance of simultaneous execution
sharing global resources safely - difficult

lock - method to prevent multiple threads from accessing a resource at the same time

$$\begin{tikzcd} \text{Address} \arrow[r, bend left, "*"] & \text{Variable} \arrow[l, bend left, "\&"] \end{tikzcd}$$

Pointer: {type} * {name}
Array: {type} arr [length]
↳ allocates space for array's contents

Struct: attributes called members
Access: (*struct).name
" struct->name

Malloc - use when you want data to exist after function call returns.

check for success.

cannot assign [] outside function.
int [] = {} ok
int * = {} not ok.

char[] var1 = "str"
↙ stack ↙ stack

char * var2 = "str"
↙ stack ↙ static.

Ptr Arithmetic

Operator	Return	Effect
*p++	*p	p=p+1
*--p	*(p-1)	p=p-1
++*p	(*p)+1	*p= *p+1
(*p)++	*p	*p = *p+1

Order of Precedence:
parenthesis → prefix → postfix

index into arrays: arr[index]
*(arr + index)

Initialize arrays: int arr[2]
int arr[] = {1, 2}

realloc (*old ptr, new-size)

calloc (#items, sizeof elem)

```
mat = (int**) calloc (n, sizeof (int*))
for (int i = 0; i < n; i++) {
        mat[i]= (int*) calloc (m, sizeof(int));
}
```

srl - 0s inserted
sra - sign extension

shamt = shift amount

Machine Instructions are in Code || matrix = int** - pointers of pointers

61C

Branch - conditional

beq, bne, blt, bge

<branch instr> <rs1> <rs2> <label>

Jump jr ra
↳ jump back to register ra } must restore: sp, gp, gp, s0-s11

jal ra, <label>
↳ saves next line of instruction's address at ra
↳ jumps to label, comes back to ra.
} Before using: save a0-a7, t0-t6, ra

j.

Loop

Loop: <branch> ... <label>
⋮

<label>

slli - shift left immediate
slli rd, rs1, imm
→ rd = rs1 << imm.

slli x11, x12, 2
x11 = x12<<2

insert zeros

lw/sw

lw <dest> <src> e.g. lw t0, 4(s0) ⇒ t0 = memory at s0 + 4

sw ~~[illegible]~~ ~~[illegible]~~ <from> <to> e.g. sw t0, 4(s0) ⇒ t0 = mem at s0 + 4

offset in bytes

sp points to where we are in stack.
↳ can rely on sp, gp, fp, "saved registers" s0-s11
↳ cannot rely on a0-a7, ra, t0-t6

- a0-a7 are args
- a0, a1 for ret.

- Subtract from sp to create more space and add to free space.
- stack used to save registers values that may be overwritten.

1 word = 4 bytes.
Each register = 32 bits

slli a0, a0, 2
Multiplies int in a0 by 4.

FUNCTION CALL

```
fn(): addi sp, sp, -x  // for x/4 words       ~ allocate space on stack.
                          (how many vars) do
                          you need to keep track
                          of.
      sw  ra, 0(sp)
      sw  s0, 4(sp)
          ⋮
      # actual function
        execution

end:  add a0, _, x0  # set return value
      lw  ra, 0(sp)  # restore ra
      lw  s0, 4(sp)
          ⋮
      addi sp, sp, x  # free space on stack
      jr ra.  # return to caller
```

In RISC-V, can represent 32 registers because we have 5 bits for each rs/rd field → 2^5 possibilities
∴ 2 bits for rd/rs field = 2^2 = 4 registers.

1 word = 4 bytes = 32 bits

"Write a function in RISC-V": Always end with jr ra

Kibi	2^{10}
Mebi	2^{20}
Gibi	2^{30}
Tebi	2^{40}
Pebi	2^{50}
Exbi	2^{60}
Zebi	2^{70}
Yobi	2^{80}

$2^{16} = 2^{10} \cdot 2^6$
$= \text{Ki } 64$

$2^{61} = 2^{60} \cdot 2^1$
$= \text{Ei} \cdot 2$

$2\,\text{Ki} = 2^{10} \cdot 2^1$
$= 2^{11}$

$16\,\text{Mi} = 2^{20} \cdot 2^4$
$= 2^{24}$

Unsigned, N bits Base B
$[0, 2^N - 1]$ $[0, B^N - 1]$

Biased, N bits, bias B
$[0 + B, 2^N - 1 + B]$

Twos Complement
- to negate: flip all bits and add 1

$[-2^{N-1}, 2^{N-1} - 1]$

Binary good for computers because less garbled than higher radix signals - more distance between valid signals.

Hex is shorthand for binary.

To flip bits of n:

n XOR 0xFFF

Bitwise Operations:

and	&
or	|
xor	^
not	~
left	<<
right	>>

Before jal, save a0-a7, t0-t6, ra
Before jr, restore sp, gp, s0-s11

LUI x10, 0xDEADC # x10 = 0xDEADC000 ← need to increment last value
Addi x10, x10, 0xEEF # x10 = 0xDEADBEEF

when added, sign extended ← 0x11111EEF ⇒ subtract 1 from UI

R: Arithmetic

31 25	24 20	19 15	14 12	11 7	0 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7

I: Immediate

31 20	19 15	14 12	11 7	6 0
im[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset	base	width (lw lb)	dest	LOAD

S:

31 25	24 20	19 15	14 12	11 7	6 0
inn[11:5]	rs2	rs1	funct 3	inn[4:0]	opcode
7	5	5	3	5	7
offset	src (from)	base (to)	width	offset	STORE

B:

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	inn[10:5]	rs2	rs1	funct3	inn[4:1]	inn[11]	opco
1	6	5	5	3	4	1	7

offset = # instructions skipped × 4 byte instruction
= x bytes.
↳ into binary → imm

U:

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7

{ LUI AUIPC

J:

31	30 21	20	19 12	11 7	6 0
inn[20]	inn[10:1]	imm[11]	inn[19:12]	rd	opcode
1	10	1	8	5	7

imm = exact offset in bytes.

Branch - scale immediate by 2 instead of 4.
- imm = [-4096, 4094] in 2 byte increments

Instruction Format:
Store: Imm used as is
Branch: Imm multiplied by 2 ∴ gives you branch distance in bytes to next instruction.

61C

$x \cdot \bar{x} = 0$ $x + \bar{x} = 1$ $\boxed{x + \bar{x}\bar{y} = x + \bar{y}}$

$x \cdot 0 = 0$ $x + 1 = 1$

$x \cdot 1 = x$ $x + 0 = x$ $x\bar{y} + \bar{x}y$ XOR

$x \cdot x = x$ $x + x = x$

$x \cdot y = y \cdot x$ $x + y = y + x$

$(xy)z = x(yz)$ $(x+y)+z = x+(y+z)$

$x(y+z) = xy + xz$ $x + yz = (x+y)(x+z)$

$xy + x = x$ $(x+y)x = x$ $\bar{x} + xy$

$\overline{x \cdot y} = \bar{x} + \bar{y}$ $\overline{(x+y)} = \bar{x} \cdot \bar{y}$ $= y + \bar{x}$

1 GHz = 1 ns

1 MHz = 1000 ns

1 GHz = 1000 ps

1 MHz = 1,000,000 ps

~~$P \quad \frac{1}{Hz} \quad \frac{Hz}{S}$~~

$\boxed{S = \frac{1}{Hz}}$

pico: 10^{-12}

nano: 10^{-9}

Hz = Instructions per second.

2's complement.

- 2's ⟷ +
- invert
- add one

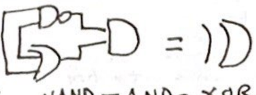
Do not

D and

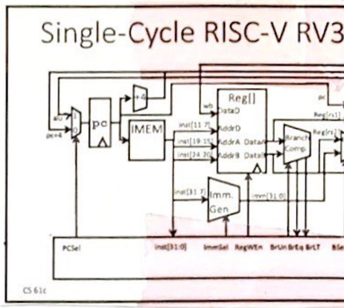
D or same = 0 diff = 1

D xor

2^{2^n} n-input logic gates



NAND — AND = XOR
OR —



Single-Cycle RISC-V RV32I Datapath

Load instruction takes longest in single cycle CPU.

Memory stage takes longest in pipelined CPU.

jal uses RegWEn.

	BrEq	PCSel	ImmSel	BrUn	ASel	BSel	MemRW	RegWEn	WBSel
add	x	+4	x	x	Reg	Reg	Read	1	ALU
ori	x	+4	I	x	Reg	Imm	Read	1	ALU
lw	x	+4	I	x	Reg	Imm	Read	1	Mem
sw	x	+4	S	x	Reg	Imm	Write	0	x
beq	1	ALU	SB	x	PC	Imm	Read	0	x
jal	x	ALU	UJ	x	PC	Imm	Read	1	PC+4

Dataparth forwards to ALU

Comparison is in execute stage.

branch - half words (immediate).

1 byte = 8 bits.

31	30 … 23	22 … 0
S	Exponent	Significand
1	8 bits	23 bits

$\cdot (-1)^S \cdot (1.significand) \cdot 2^{Exp-127}$

Denormalized - $(-1)^S \cdot 0.significand \cdot 2^{Exp-126}$

Exponent	Significand Mantissa	Meaning
0x00	0	0
0x00	non-zero	± Denorm num
0x01 - 0xFE	anything	± Norm Num
0xFF	0	± ∞
0xFF	non-zero	NaN.

Kibi - 2^{10}

Mebi - 2^{20}

Gibi - 2^{30}

Tebi - 2^{40}

Pebi - 2^{50}

Exbi - 2^{60}

Zebi - 2^{70}

Yobi - 2^{80}

Unsigned, N bits, Base B

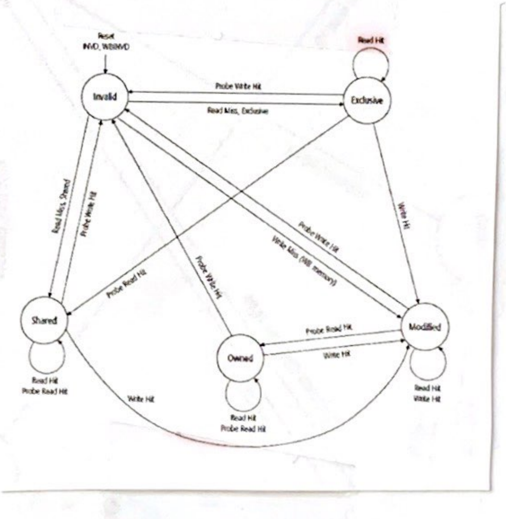
$[0, B^N - 1]$

Biased, N bits, base 2

$[0 + B, 2^N - 1 + B]$

Twos Complement

$[-2^{N-1}, 2^{N-1} - 1]$



Combinational Logic circuits
- perform function on inputs
  e.g. add them and output result
  -ALU

Sequential Logic
- Store information, input and output updated on flip-flop
- registers

Setup Time - when input must be stable before CLK edge

Hold Time - when input must be stable after CLK edge.
-violation when new data arrives at register before hold time has passed

CLK-to-Q - how long it takes output to change, measured from edge of CLK.

Critical Path - path data could flow thru in a circuit that causes longest delay.

Max Delay - Setup Time + CLK-to-Q + CL Delay

Min Period = Max Delay

$\text{Max Freq} = \frac{1}{\text{Min Period}}$

↑ registers, ↓ critical path

↑ registers, ↑ latency

Execution Steps
- instruction fetch, increment PC
- decode/register read
- execute: calculate address, ALU ops
- memory: read data (load), write data (store)
- register write: write data back

Latency - time for one instruction to go through pipeline to completion
= #stages · clock period

Throughput - rate at which we can complete instructions per unit of time
= #stages/latency #instr/second

Hazard -

Structural - required resource is busy
→ ≥ 2 instr want 1 resource

Data - instr depends on result from prev inst
↳ stall
↳ forwarding

Control - branch instr
↳ instr (2) fetched before we decode
↳ kill 2 if branch taken
↳ branch prediction

$$\text{Time} = \frac{\text{Instr}}{\text{Program}} \cdot \underbrace{\frac{\text{Clock Cycles}}{\text{Instr}}}_{\text{CPI}} \cdot \frac{\text{Seconds}}{\text{clock cycle}} = \frac{\text{Sec}}{\text{Program}}$$

$10^{-12} = \text{ps}$, $10^{-9} = \text{ns}$.

Write Through - update cache, update memory
No WA

Write Back - update cache until cache block gets tossed. → update memory.
Allocate

Dirty - cache different from memory

AMAT = Time for a hit + Miss rate · miss penalty
(cycles)

Cache Miss

Compulsory - first time you bring in a block

Conflict - data block was in cache, but different block needed and replaced block
- conflict between blocks
-can fix by increasing associativity

Capacity - all cache slots were full, had to replace data I was looking for.

Write Allocate - put block inside cache and perform write hit action.

No write allocate - write to memory and don't put block inside cache.

$\#\text{index bits} = \log_2(\#\text{blocks} / N)$

$\#\text{blocks} = \text{cache size} / \text{block size}$

N = N-way set associative

$\#\text{offset bits} = \log_2(\text{block size})$

$\#\text{tag bits} = \text{total} - \text{index} - \text{offset}$

$\log_2(\text{Address Space}) = \text{bits in address}$

$\text{Cache Data per row} = \text{block size (bytes)} \cdot \frac{8\ \text{bits}}{\text{byte}}$.

Cache

Fully Associative - Block placed ~~aby~~ anywhere in cache
- no index field, one comparator/block

DM - Block goes one place in cache.
- #sets = #blocks -prone to more conflict misses
- 1 comparator.

N-way - N places for block
- #sets = #blocks / N
- N comparators

AMAT = L1 Hit ~~rate~~ time + L1 Miss rate (L2 hit time + L2 local MR · L2 Miss penalty)

L2 global miss rate = #L2 misses / total # accesses
= (# L2 misses / # L1 accesses) · (# L1 misses / # L1 misses)
= L1 MR · L2 local MR

---

tag - which block is currently in cache slot.
index - the cache slot/set
offset - where in block desired data is

$$\text{L1 MR} = 1 - \frac{\text{L1 size (bytes)}}{\text{memory in chip}}$$

$$\text{L2 global MR} = \frac{1 - \text{L2 (size in bytes)}}{\text{memory in chip}}$$

False sharing - only applicable when data being accessed by threads are distinct.

L2 local HR · L1 local MR = L2 global HR

L2 local MR = 1 - L2 local HR

Compiler - Input: High Level language code, e.g. .c
Output: assembly language, contains pseudo instructions

pseudo-instr expanded

Assembler - Input: Assembly - computes offset of branch instr
Output: Object code, info tables
.o

Header -uses directives
Text -replaces pseudo instructions
Data -produce machine language
Relocation -creates object file.
Symbol
Debugging -2 passes over the program to solve "Forward Reference"
1.) remember positions of labels
2) use label positions to generate code.

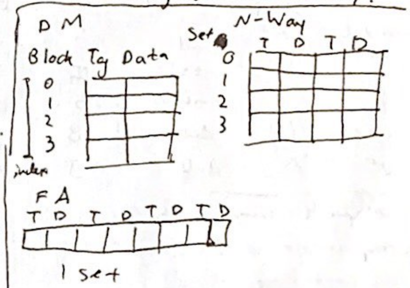
References to static data not determined...

Symbol - list of "items" that are used by other files

Relocation - list of items this file needs.

Linker - Input - Object code files, info tables
Output - Executable code .out
resolves references. machine code

Loader - Input: executable code.
Output: program run.
load program into memory.



Adding a register changes exact sequence of outputs. | Strong Scaling - increase number of processors

Floating Point - Bias = 127

Sign	Exp	Sig.
1	8	23

Normalized

move decimal by

$(-1)^{Sign} \cdot 2^{[Exp-Bias]} \cdot 1.sig_2$

Denorm

$(-1)^{Sign} \cdot 2^{Exp-Bias+1} \cdot 0.sig_2$

Exp	Sig	Meaning
0	–	Denorm
1-254	–	Normal
255	0	∞
255	!0	NaN

AMAT - average time for mem access

= hit time + miss rate · miss penalty

Amdahl's Law

$$S = \frac{T_o}{T_E} = \frac{1}{(1-F)+\frac{F}{S_E}},$$

F = Fraction of program that uses enhanced task.

data level/parallelism
- vectorized calculation
- e.g. Intel intrinsics

thread level parallelism
- OpenMP
  parallel
  - doesn't automatically split threads
  parallel for
  - splits into threads, own private variables.

Cache Coherence

MSI

modified - info is changed only in one cache

shared - info is not modified in ≥ 1 caches memory up to date

invalid - info not in cache

Valid	Dirty	State
0	0	invalid
0	1	invalid
1	0	shared
1	1	modified

- cannot share information that is dirty until memory up to date
- cannot avoid checking if other caches have information in shared/invalid state

MESI

New: Exclusive
- data not modified
- data only in one cache

⇒ shared: ≥ 2 caches have it (1 other)

Valid	Dirty	Shared	State
0	0	0	invalid
0	0	1	invalid
0	1	0	invalid
0	1	1	invalid
1	0	0	exclusive
1	0	1	shared
1	1	0	modified
1	1	1	error

Availability: $\frac{MTTF}{MTTF+MTTR}$

Mean time between failures: MTTF + MTTR

MTTF = mean time it takes for one disk to fail / Reliability

MTTR - mean time it takes for one disk to repair

Mean time to repair Service interruption

MOESI

Problem: cannot share data that isn't up to date in main memory

New: Owned

→ data is modified and cache responsible for updating main memory

→ data also in another cache

modified - data differs from memory, in 1 cache

owned - data differs from memory, in 1 cache

exclusive - data is same as memory, in 1 cache

shared - data in > 1 cache. data may (not) be same as in memory

invalid - data not in current cache

Truth table: same as MESI, last state owned.

map(func) - returns new DD by passing each element of src thru func

flatmap(func) - each input can be mapped to 0 or more output

reduceByKey(func) - each key has values aggregated according to func

reduce(func) - elements aggregated regardless of key

Write permission necessary in write through cache.

Atomic read/write
- read/write in single instruction

Disk Access Time

= seek time + rotation time + transfer time + controller overhead

seek time → positioning activator

rotation time → thing to r/w must rotate to activator

transfer time → data transfer

controller overhead → handshake/machine overhead

Seek Time: # tracks/3 · time

Rotation: time for 1/2 rotation ; seconds / halfrotation

transfer: Size/ transfer rate

OS

Hardware } Kernel } Shells } Applications

- OS is first to start/run
- finds controls all machine devices
- starts services
- loads, runs, manages programs

I/O interfaces for keyboard, network, etc.
- connect/control using PCI bus

Processor Action:
- Special I/O instructions + hardware
- Memory Mapped I/O
  → memory contains registers for I/O

Polling

Device Registers:

Control: OK to read/write?

Data: contains data

Processor reads from control

waits for device to set ready bit 0 → 1

loops reading control register until ready

Cost: $poll/_s \cdot instr/_{poll} = instr/_s$

Processor throughput: instr/s

Ratio: P/T

Contiguous, large blocks per process makes memory fragmented

Interrupt
- occurs when I/O is ready, needs attention
- interrupt current program
- transfer control to interrupt handler
- must act soon

CPU Vector Interrupt Table - stores locations of different interrupts
- where to go when receiving specific interrupt

Interrupt comes - trap handler

↳ contains epc, special register to contain return address, etc of interrupt

Trap - action of servicing interrupt or exception by hardware jump to "interrupt" or "trap handler" code

Exception - must act NOW
- can have at any/all pipeline stages
- handled like pipeline hazards, but by exception handler

Boot
1) Bios: find storage device and load first sector
2) load kernel
3) OS boot: initialize services, drivers
4) Init: launch application that waits for input in loop

Sys Calls: OS routine call
- create interrupts, OS handles

Context Switch: process switching/change application

DMA
- allows CPU to do other things
- controls data transfer from controller's memory to processor's memory
- DMA engine contains register written by CPU

Incoming Data
- CPU gets interrupt, calls DMA

Outgoing
- CPU initiates transfer
- calls DMA to copy data from register to device

Where to put DMA:
1) between L1 and CPU
   ✓ cache coherency fine
   X lose locality benefits
2) between last level cache and main memory
   ✓ no data cached
   X need to manage coherency

61C

VM

$2^{\text{page offset bits}}$ = page size

page offset

$\log_2$(# pages) = virt page bits

page - chunk of memory/disk with set size

VA → PA

page table - determines mapping
stored in memory
each process gets own

∃ register telling hardware the address of first entry of page table

VPN	Page Valid	Permission Bits (r, w, ...)	Physical Page Number (PPN)

VM much bigger than PM

cache → memory
page → disk ← slower

ps uses more memory ⇒ more valid bits in page table

Dependability via Redundancy
- confirmation by duplicating
  1) data centers - internet service
  2) routes - internet
  3) disks - data
  4) memory bits - data

Fault - failure of component

failure - entire system unusable

Spatial Redundancy - replicated data or check info "copies" in different places

Temporal Redundancy - sending multiple times to ensure correctness

Protection Fault - Page table entry for virtual page has permission bits prohibiting requested operation

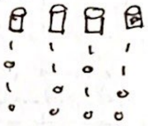
Page Fault - page table entry for virtual page has valid bit set to false - entry not in memory

check TLB before page table
If miss, look at page table row corresponding to page bits
- Replace in TLB ⇒ invalidate in page table
- TLB is invalidated every process switch

RAID 1
- each disk fully duplicated onto mirror
  - high availability can be achieved
- writes limited to single-disk speed
- reads may be optimized

Expensive (most) - 100% capacity overhead

RAID 3
- P contains sum of other disks per stripe mod 2
- if disk fails, subtract P from sum of other disks to find missing information



RAID 4
- high I/O rate parity
- works well for small reads

small writes (one disk):
- 1) read other data disks, create new sum and write to parity disk
  2) since P has old sum, compare old data to new data, add difference to P

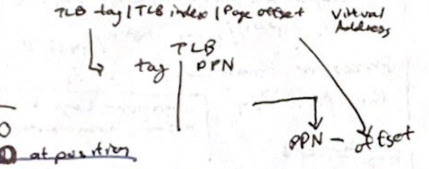
RAID 5
- high I/O rate interleaved parity
  ↳ makes possible independent writes
- Check information is distributed across disks, not a single disk is queried for all check
- load balance
- can execute independent writes in parallel
- XOR

4,5 use block striping

3,4,5 - a single write can require 2 reads and 2 writes.

Fill in TLB
- write, not in TLB
  1) map VPN to PPN
  2) set valid and dirty

TLB



Hamming ECC

1 0

- ~~even number of 's = at position~~
- ~~odd number of 1's = 1 at position~~
- want parity bit to get those assigned to add up to an even number. Either 0 or 1.

Page offset = $\log_2$(Page size) / same for virtual + physical

Virtual Address bits = $\log_2$(VA space)

Physical address bits = $\log_2$(PA space)

Virtual, physical pages are same size

VPN = VA - offset

PPN = PA - offset

- replace large disk drive with cabinet of small disk drives

Problem: replacement makes reliability worse.

Raid 3
- dedicated disk drive that did parity calculation across all bit-by-bit level
- failure - will tell you (disk drives)
- parity allows you to figure out what happened

Raid 4
- larger block size
  - more efficient parity
  - dedicated disk drive for parity
- limited by writes to parity
- no concurrent independent writes

Raid 5
- parity not on just one drive
- Problem! every write, have to update parity disk drive → bottleneck
- interleave parity across drives
- concurrent independent writes
- reading not an issue

Networks

Internet:
1) internet protocol suite
2) world wide web - HTTP

Shared Interconnect: 1 at a time
vs
Switched: pairs communicate
- can use more bandwidth, more ways to get from A to B

to send and receive

Send:
- copy data to OS buffer
- calculate checksum, start timer
- send data to network interface

Receive
- copy data from network interface to OS buffer
- calculate checksum ACK
  1) OK - send ACK, copy data to user address, signal application to continue
  2) Not OK - delete message, timer expires, sender resends

6 Great Ideas
- Design for Moore's law
  - multicore, parallelism, openMP
- Abstraction to Simplify Design
- Make common case fast
- Dependability via Redundancy
- Memory hierarchy
  - locality, consistency, false sharing
- Performance via Parallelism / Pipelining / Prediction

5 Kinds of Parallelism
- Request level (WSC)
- Instruction (Pipelining)
- Data Level (SIMD)
- Data / Task Level (MapReduce)
- Thread Level (Multicore, OpenMP)

P1: 1, 3, 5, 7, 9

P2: 2, 3, 6, 7, 10, 11

P4: 4, 5, 6, 7, 12, 13, 14, 15

P8: 8, 9, 10, 11, 12, 13, 14, 15 ⟼8 , 24, 25, ...

Layers
- application - what does app expect to be delivered
- transport (TCP/ P) - protocols to ensure data reliably delivered
- network (IP) - how to get from A to B
- data link - can I communicate on thing I'm connected to
- physical link - shape of waveform

Loop Unrolling: less time checking, increment; more time in block