CS 152
Computer Architecture and Engineering

Lecture 19 – Error Correcting Codes

2005-3-31

John Lazzaro

(www.cs.berkeley.edu/~lazzaro)

TAs: Ted Hong and David Marquardt

www-inst.eecs.berkeley.edu/~cs152/

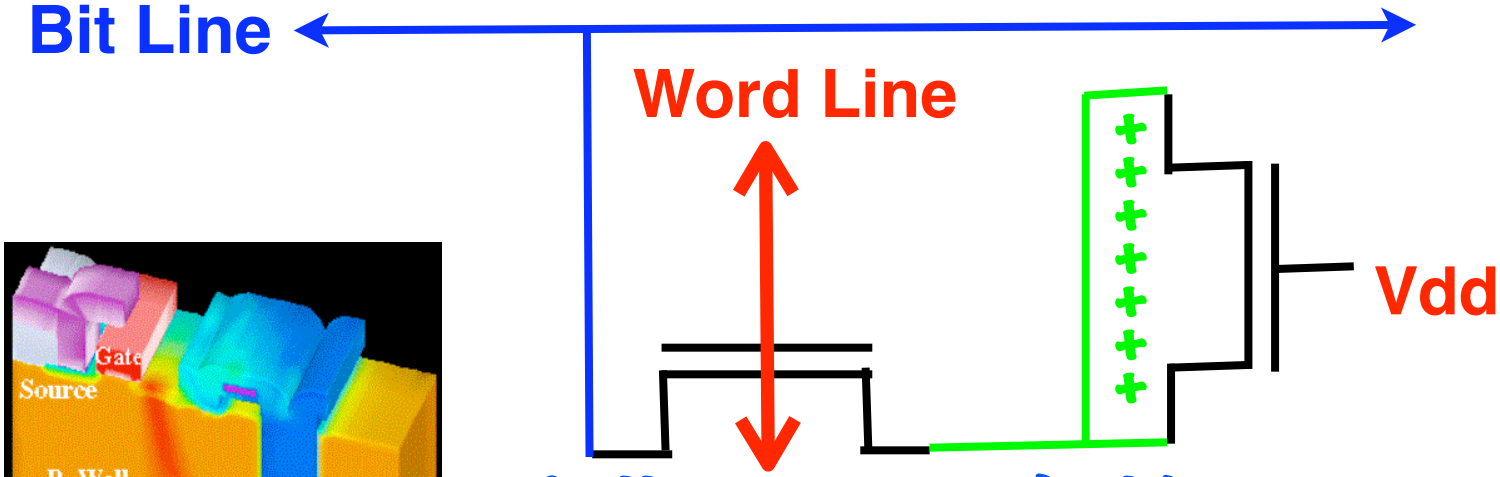
Today: Computing in an imperfect world

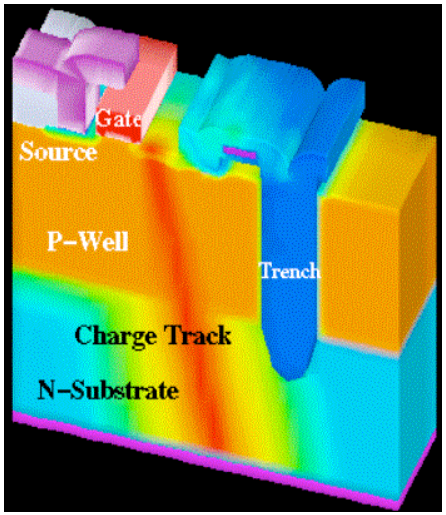
- Detecting and correcting RAM bit errors

- Replacing lost network packets, recovering from disk drive failure

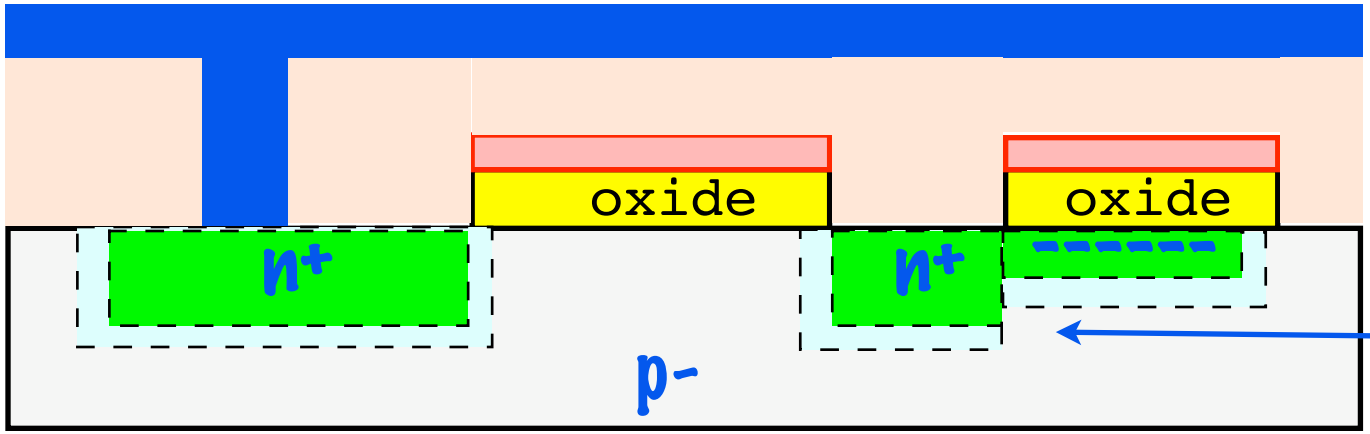
- Detecting arbitrary bit errors in network packets

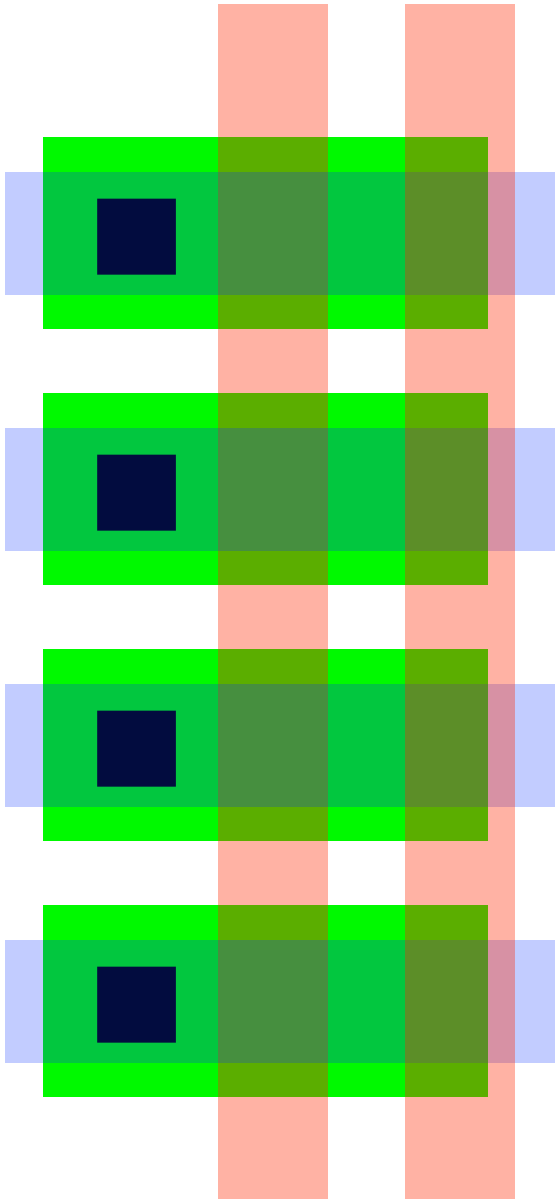
DRAM Challenge: Cosmic Rays ...





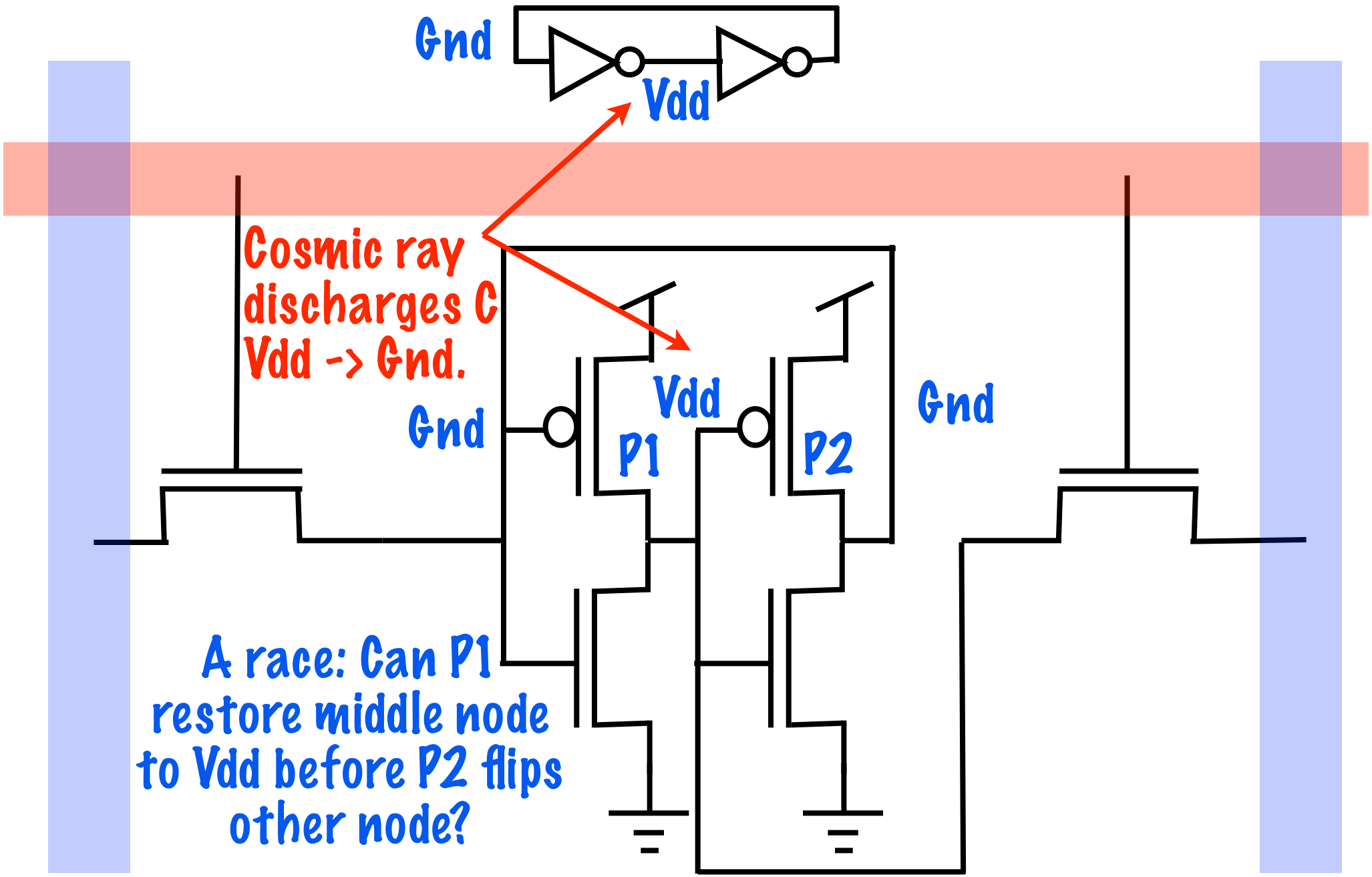
Cell capacitor holds 25,000 electrons (or less). Cosmic rays that constantly bombard us can release the charge!





Cosmic ray hit.

Can this happen in SRAM?

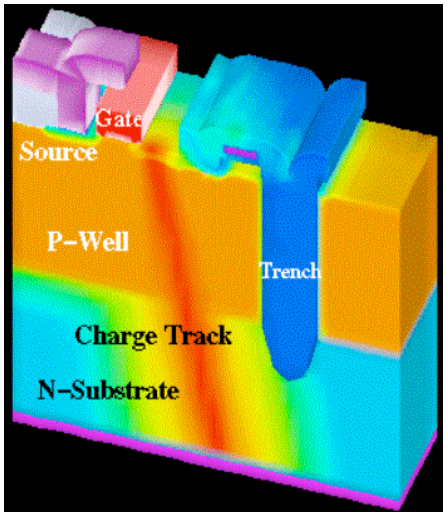


Practical effect of a cosmic ray ...

```
ADDIU R1, R0, 7
SW R1, 100(R0)
```

Address 100: 0b00...0111

Cosmic ray hit.



Address 100: 0b00...0011

```
LW R1, 100(R0)
```

After LW, R1 holds 3 but it should hold 7.

Bit flips on memory holding instructions are bad too!

To "detect" errors -- add 'P', a parity bit

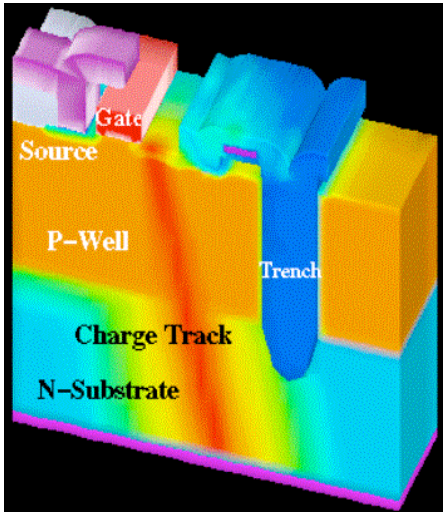
Extra "parity" bit for every word. Not seen by software. Hardware sets it on every write, so that the number of 1's in every 33 bit word is even (even parity).

P

Address 100: 0b00...0111 1

Does this work if two bits flip? If three?

Cosmic ray hit.



Address 100: 0b00...0011 1

On a read, count the number of 1s. If odd, a bit flipped.

So, halt the program and reboot? Application may know if this bit matters, but there's no API to ask it ...

Error Correction: Hamming Codes ...

Richard Hamming.
Computing pioneer.

Famous quote:

"Computers are
not for numbers.

Computers are for
understanding."

Trick: compute parity of subsets of bits

Consider 4 bit words. Add 3 parity bits.

$D_3D_2D_1D_0$ $P_2P_1P_0$

0 1 1 0 ???

Each parity bit computed on a subset of bits

$P_2 = D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 1 \text{ xor } 1 = 0$

$P_1 = D_3 \text{ xor } D_2 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$

$P_0 = D_3 \text{ xor } D_1 \text{ xor } D_0 = 0 \text{ xor } 1 \text{ xor } 0 = 1$

Use this word bit arrangement

$D_3D_2D_1P_2D_0P_1P_0$

0 1 1 0 0 1 1

"Just believe" for now, we will justify later ...

Case #1: No cosmic ray hits

We write: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

Later, we read: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

No errors ... but how do we know that?

On readout we compute:

P_2 xor D_3 xor D_2 xor D_1 = 0 xor 0 xor 1 xor 1 = 0
P_1 xor D_3 xor D_2 xor D_0 = 1 xor 0 xor 1 xor 0 = 0
P_0 xor D_3 xor D_1 xor D_0 = 1 xor 0 xor 1 xor 0 = 0

If $P_2P_1P_0 = 0$ no errors

These equations come from how we computed $P_2P_1P_0$

P_2 = D_3 xor D_2 xor D_1 = 0 xor 1 xor 1 = 0
P_1 = D_3 xor D_2 xor D_0 = 0 xor 1 xor 0 = 1
P_0 = D_3 xor D_1 xor D_0 = 0 xor 1 xor 0 = 1

Case #2: A cosmic ray hits ...

We write: $D_3D_2D_1P_2D_0P_1P_0$
0 1 1 0 0 1 1

Later, we read: $D_3D_2D_1P_2D_0P_1P_0$
0 1 0 0 0 1 1

Cosmic ray hit D1. But how do we know that?

On readout we compute:

P_0 xor D_3 xor D_1 xor D_0 = 1 xor 0 xor 0 xor 0 = 1

P_1 xor D_3 xor D_2 xor D_0 = 1 xor 0 xor 1 xor 0 = 0

P_2 xor D_3 xor D_2 xor D_1 = 0 xor 0 xor 1 xor 0 = 1

$P_2P_1P_0$ = b101 = 5

What does "5" mean?

7 6 5 4 3 2 1
$D_3D_2D_1P_2D_0P_1P_0$
0 1 0 0 0 1 1

Note: we number the least significant bit with 1, not 0! 0 is reserved for "no errors".

The position of the flipped bit! To repair, just flip it back ...

Why did we choose "3" parity bits?

Consider 4 bit words. Add 3 parity bits.

$D_3D_2D_1D_0$ $P_2P_1P_0$

0 1 1 0 ???

Observation: The parity bits need to encode the "no error" condition, plus a number for each bit (both data and parity bits)

For "p" parity bits and "d" data bits:

$$d + p + 1 <= 2^p$$

For mid-term II: be able to construct a Hamming code for an arbitrary "d" ...

Why did we arrange bits as we did?

Consider 4 bit words. Add 3 parity bits.

$D_3D_2D_1D_0$ $P_2P_1P_0$

Why do we arrange bits?

With this order, an odd parity means an error in 1,3,5, or 7. So, P0 is the right parity bit to use:

$P_2P_1P_0$

7	6	5	4	3	2	1
D_3		D_1		D_0		P_0
D_3	D_2			D_0	P_1	
D_3	D_2	D_1	P_2			
D_3	D_2	D_1	P_2	D_0	P_1	P_0

Start by numbering, 1 to 7.

Etc ... each bit narrows down the suspect bits, until it is certain.

An odd parity means a mistake must be in 2, 3, 6, or 7 -- the four numbers possible if P1 = 1!

Why did we arrange bits as we did?

Consider 4 bit words. **Add 3 parity bits.**

$D_3D_2D_1D_0$ $P_2P_1P_0$

$P_2 = D_3 \text{ xor } D_2 \text{ xor } D_1$ $P_1 = D_3 \text{ xor } D_2 \text{ xor } D_0$ $P_0 = D_3 \text{ xor } D_1 \text{ xor } D_0$

0	0000000
1	0000111
2	0011001
3	0011110
4	0101010
5	0101101
6	0110011
7	0110100
8	1001011
9	1001100
A	1010010
B	1010101
C	1100001
D	1100110
E	1111000
F	1111111

7 bits can code 128 numbers, but only 16 of these numbers are legal.

It takes 3 bit flips to move from one legal number to another (for all 16 numbers)

If only one bit flips, we can always figure out the "closest" legal number, and correct.

What if 2 cosmic rays hit?

We write: $D_3 D_2 D_1 P_2 D_0 P_1 P_0$

0 1 1 0 0 1 1

Later, we read: $D_3 D_2 D_1 P_2 D_0 P_1 P_0$

0 1 0 0 0 0 1

Cosmic ray hit D1 and P1.

On readout we compute:

P_0 xor D_3 xor D_1 xor D_0 = 1 xor 0 xor 0 xor 0 = 1

P_1 xor D_3 xor D_2 xor D_0 = 0 xor 0 xor 1 xor 0 = 1

P_2 xor D_3 xor D_2 xor D_1 = 0 xor 0 xor 1 xor 0 = 1

$P_2 P_1 P_0$ = b111 = 7

What does "7" mean?

7 6 5 4 3 2 1

$D_3 D_2 D_1 P_2 D_0 P_1 P_0$

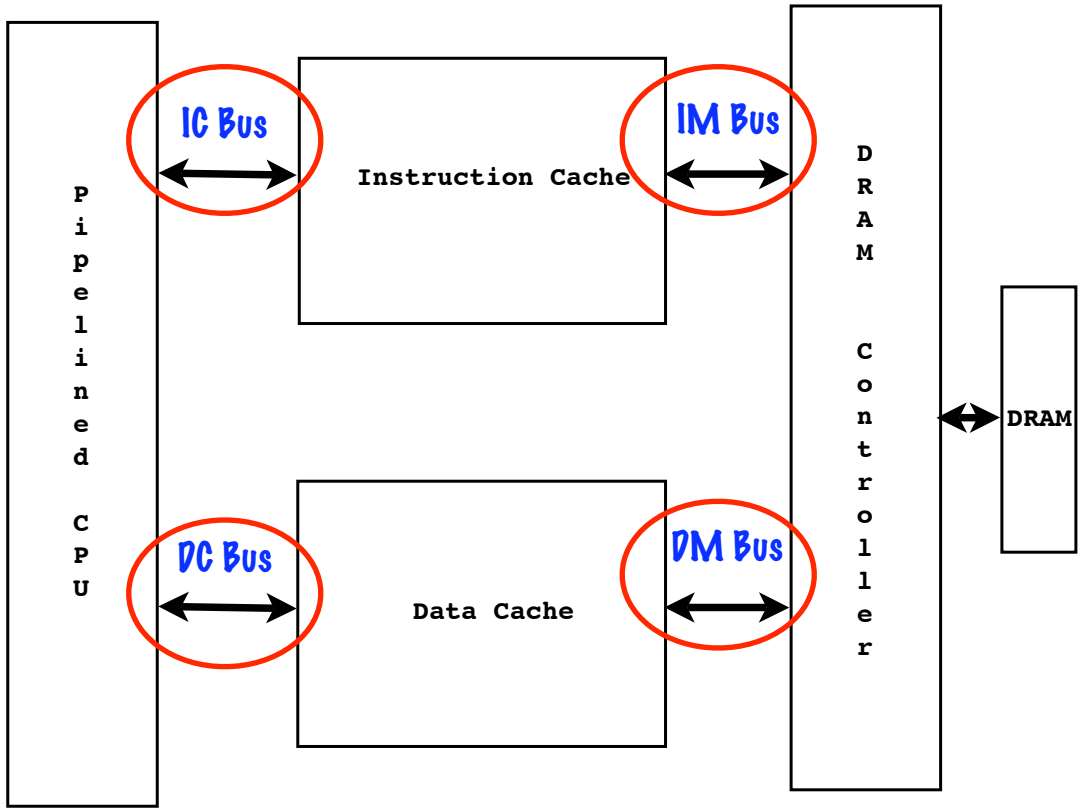
0 1 0 0 0 1 1

Note: it does do 2-bit "detect" (since P3 P2 P1 does not code 0), but it does not let us know that we can't correct ...

"Correcting" this bit makes things worse! Thus, this code corrects "single" bits only.

Tomorrow: Preliminary Review in Section

Th 3/31	Error Correcting Codes			Final Project: Preliminary Design Document due to TAs, 11:59PM **Also: Peer Evaluations ...**
F 4/1				Final Project: Preliminary Review



Define the timing diagrams and signal names for the IM, DM, IC, DC buses.

List the bugs you will target in test benches.

Other items ...

Block Data and Errors

Error correct/detect in blocks of data

Some devices deliver many bits at once

Programs request a "block" of data from a disk:

Block = 0.5K to 4K bytes

Networks deliver data in "packets" - blocks of data, max block size depends on network

In a perfect world, blocks always arrive, holding the exact data stored (disk) or sent (network).

What happens in an imperfect world?

Every so often, a block does not show up

A "block went bad" on the disk. The data you wrote you will never see again.

Or, maybe the disk died. All of those blocks you will never see again.

RAID: Was invented to solve this problem ...

Networks "lose" packets in transit. Maybe a router was overloaded, and "dropped" the packet. Other reasons too ...

The good part: we usually know ...

The disk will tell you "this block does not exist" or "the disk is dead", by returning an error code when you do a read.

Often, applications number packets as they send them, by adding a "sequence number" to packet header. Receivers detect a "break" in the number sequence ...

If we know this will happen in advance, what can we do, at the OS or application level?

Answer: Store or send redundant data

Simple case: Two 1KB blocks of data (A and B)

Create a third block, C: "Parity codes"

C = A xor B (do xor on each bit of block)

Read all three blocks. If A or B is not available but C is, regenerate A or B:

A = C xor B

B = C xor A

The math is easy: the trick is system design!
Examples: RAID, voice-over-IP parity FEC.

What happens in an imperfect world?

Blocks show up, but not all bits are correct

This is like parity and Hamming codes for memory, in theory. Either add "detection", for cases where errors were in "transmission", or add "correction", if data is gone forever (disk) or if retransmission would arrive too late (Voice over IP -- speech that arrives too late is useless).

Error detection for big data blocks ...

Why not add just 1 parity bit to a 4KB block?

Answer: if an odd number of bits are flipped, you will never know. In practice, add "checksum" to block -- 16 or 32 bit signatures, that reflect the bits in the block.

Recompute checksum on receipt to detect errors.

Checksum algorithms ...

Can checksums detect every possible error?

Answer: No -- for a 16-bit checksum, there are many possible packets that have the same checksum. If you are unlucky enough to have your transmission errors convert a block into another block with the same checksum value, you will not detect the error!

Checksum algorithms ...

Checksum is a misnomer -- do not use the sum of the bytes in a block as a checksum!

Why? A good checksum lets any bit in the block alter any of the bits in a checksum, wherever the byte appears in the block. Addition does not have the property.

A classic good checksum: Treat the block as a giant number, divide it by a constant with "special properties", and use the remainder of the division as the checksum.

"Cyclic redundancy codes". (CRC). Uses this idea, chooses a number system that is both fast and "good". Used in Ethernet. Fast hardware and software implementations.

Summary: Computing in an imperfect world

- Detecting and correcting RAM bit errors

- Replacing lost network packets, recovering from disk drive failure
- Detecting arbitrary bit errors in network packets