The SyncMeisters had everything working besides the test file hammer. However, hammer is the hardest test, so in a sense, their project was a bug or two away from working.

Writing back to the regfile when a stall occurs on the cache just before or after needing to write seems to mess up their processor. The other tests worked fine on board.
When cache bugs make it to product ...

(Testing our financial trading system), we found a case where our software would get a bad calculation. Once a week or so.

Eventually, the problem turned out to be a failure in a CPU cache line refresh. This was a hardware design fault in the PC.

The test suite included running the code for two weeks at maximum update rate without error, so this bug was found.

Eric Ulevik
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!
Can this happen in SRAM?

Cosmic ray discharges C: Vdd -> Gnd.

A race: Can P1 restore middle node to Vdd before P2 flips other node?
Practical effect of a cosmic ray ...

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

Address 100: 0b00...0111

Cosmic ray hit.

`LW R1, 100(R0)`

Address 100: 0b00...0011

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 computes it on every write, so that the number of 1’s in every 33 bit word is even (even parity).

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 ...


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.

\[ D_3D_2D_1D_0 \]
\[ 0 \ 1 \ 1 \ 0 \]
\[ P_2P_1P_0 \]

Each parity bit computed on a subset of bits

\[ P_2 = D_3 \ xor \ D_2 \ xor \ D_1 = 0 \ xor \ 1 \ xor \ 1 = 0 \]
\[ P_1 = D_3 \ xor \ D_2 \ xor \ D_0 = 0 \ xor \ 1 \ xor \ 0 = 1 \]
\[ P_0 = D_3 \ xor \ D_1 \ xor \ D_0 = 0 \ xor \ 1 \ xor \ 0 = 1 \]

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:
\[ D_3D_2D_1P_2D_0P_1P_0 \]
\[ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \]

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

Later, we read:
\[ D_3D_2D_1P_2D_0P_1P_0 \]
\[ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \]

On readout we compute:
\[
\begin{align*}
P_2 \ xor \ D_3 \ xor \ D_2 \ xor \ D_1 &= 0 \ xor 0 \ xor 1 \ xor 1 = 0 = C_2 \\
P_1 \ xor \ D_3 \ xor \ D_2 \ xor \ D_0 &= 1 \ xor 0 \ xor 1 \ xor 0 = 0 = C_1 \\
P_0 \ xor \ D_3 \ xor \ D_1 \ xor \ D_0 &= 1 \ xor 0 \ xor 1 \ xor 0 = 0 = C_0
\end{align*}
\]

If \( C_2C_1C_0 = 0 \)
no errors

These equations come from how we computed \( P_2P_1P_0 \)
\[
\begin{align*}
P_2 &= D_3 \ xor \ D_2 \ xor \ D_1 = 0 \ xor 1 \ xor 1 = 0 \\
P_1 &= D_3 \ xor \ D_2 \ xor \ D_0 = 0 \ xor 1 \ xor 0 = 1 \\
P_0 &= D_3 \ xor \ D_1 \ xor \ D_0 = 0 \ xor 1 \ xor 0 = 1
\end{align*}
\]
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:

$P_2 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 1 = C_2$

$P_1 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 0 = C_1$

$P_0 \text{ xor } D_3 \text{ xor } D_1 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 0 \text{ xor } 0 = 1 = C_0$

What does “5” mean?

The position of the flipped bit!

To repair, just flip it back...

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

7 6 5 4 3 2 1

$D_3D_2D_1P_2D_0P_1P_0$

0 1 0 0 0 1 1
Why did we choose “3” parity bits?

Consider 4 bit words. Add 3 parity bits.

\[ D_3D_2D_1D_0 \quad P_2P_1P_0 \quad \text{???} \]

\[ 0\ 1\ 1\ 0 \]

Observation: The \(C_2C_1C_0\) 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:

\[ d + p + 1 \leq 2^p \]
Why did we arrange bits as we did?

Consider 4 bit words. Add 3 parity bits.

Why did we arrange bits as we did?

Consider 4 bit words. Add 3 parity 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:**

\[ C_2C_1C_0 \]

**How do we re-arrange bits?**

Start by numbering, 1 to 7.

A \( C_i \) in \( C_2C_1C_0 \) exists for each \( P_i \).

An odd parity means a mistake must be in 2, 3, 6, or 7 -- the four numbers possible if \( C_1 = 1 \! \).
Why did we arrange bits as we did?

Consider 4 bit words.  Add 3 parity bits.

\[
\begin{align*}
D_3D_2D_1D_0 & \quad P_2P_1P_0 \\
D_3 \text{xor } D_2 \text{xor } D_1 & = P_2 \\
D_3 \text{xor } D_2 \text{xor } D_0 & = P_1 \\
D_3 \text{xor } D_1 \text{xor } D_0 & = P_0
\end{align*}
\]

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.

7 bits can code 128 numbers, but only 16 of these numbers are legal.
What if 2 cosmic rays hit?

We write:

\[ D_3D_2D_1P_2D_0P_1P_0 \]
\[ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \]

Later, we read:

\[ D_3D_2D_1P_2D_0P_1P_0 \]
\[ 0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \]

On readout we compute:

\[ P_2 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_1 = 0 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 1 = C_2 \]
\[ P_1 \text{ xor } D_3 \text{ xor } D_2 \text{ xor } D_0 = 0 \text{ xor } 0 \text{ xor } 1 \text{ xor } 0 = 1 = C_1 \]
\[ P_0 \text{ xor } D_3 \text{ xor } D_1 \text{ xor } D_0 = 1 \text{ xor } 0 \text{ xor } 0 \text{ xor } 0 = 1 = C_0 \]

What does “7” mean?

Note: it does do 2-bit “detect” (since C3 C2 C1 does not code 0), but it does not let us know that we can’t correct ...
Run your test vector suite on the Calinx board, display results on LEDs.

Test concurrent transactions on both buses, with randomized start times. Load, store, and verify different data word patterns.
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

NAND Flash (iPod nano) emulates disk blocks

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 or FLASH. The data you wrote you will never see again.

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

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

RAID: Was invented to solve this problem ...
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?
Simple case: Two 1KB blocks of data (A and B)

Create a third block, C:

\[ C = A \text{ xor } B \]  (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:

\[ A = C \text{ xor } B \quad \text{Why?} \quad (A \text{ xor } B) \text{ xor } B = (B \text{ xor } B) \text{ xor } A = A \]

\[ B = C \text{ xor } A \]

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 bits and Hamming codes for memory. Add "detection" for cases where retransmission is possible. Add "correction" if data is gone forever (disk crash) 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.
Example: Internet Packet Checksum

Computed over header

<table>
<thead>
<tr>
<th>Version</th>
<th>IHL</th>
<th>Type of Service</th>
<th>Total Length</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Identification</td>
<td>Flags</td>
<td>Fragment Offset</td>
<td></td>
</tr>
<tr>
<td>Time to Live</td>
<td>Protocol</td>
<td>Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Destination Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Payload data (size implied by Total Length header field)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 - don’t take the word literally.

```
01111111  10000011  00000000  00000000  00001001
127   131    0      0      9
```

Add unsigned bytes into an 8-bit register: 11

Common transmission error: string of stuck 1s.

```
01111111  10000011  00000001  11111111  00001001
127   131    1    255      9
```

Add unsigned bytes into an 8-bit register: 11 (!)

Conclusion: The modulo sum isn’t a good “checksum”
A better starting point: Division

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). Used in Ethernet. Fast hardware and software implementations.

Isn’t division slow? CRCs are not doing standard division. Instead, CRCs compute over polynomials over finite fields of integers modulo 2. This maps well to binary logic.
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