Discussion 14: I/O, ECC/Parity, RAID

Hamming ECC

Recall the basic structure of a Hamming code. Given bits 1, \ldots, m, the bit at position 2n is parity for all the bits with a 1 in position n. For example, the first bit is chosen such that the sum of all odd-numbered bits is even.

1. How many bits do we need to add to 0011\textsubscript{2} to allow single error correction?
   Parity Bits: 3
2. Which locations in 0011\textsubscript{2} would parity bits be included?
   Using P for parity bits: PP0P011\textsubscript{2}
3. Which bits does each parity bit cover in 0011\textsubscript{2}?
   Parity bit #1: 1, 3, 5, 7
   Parity bit #2: 2, 3, 6, 7
   Parity bit #3: 4, 5, 6, 7
4. Write the completed coded representation for 0011\textsubscript{2} to enable single error correction.
   \underline{1000111}_2
5. How can we enable an additional double error detection on top of this?
   Add an additional parity bit over the entire sequence.
6. Find the original bits given the following SEC Hamming Code: 0110111\textsubscript{2}
   Parity group 1: error
   Parity group 2: okay
   Parity group 4: error
   Incorrect bit: 1 + 4 = 5, change bit 5 from 1 to 0: 0110011\textsubscript{2}
   \underline{0100011}_2 \rightarrow 1011\textsubscript{2}
7. Find the original bits given the following SEC Hamming Code: 1001000\textsubscript{2}
   Parity group 1: error
   Parity group 2: okay
   Parity group 4: error
   Incorrect bit: 1 + 4 = 5, change bit 5 from 1 to 0: 1001100\textsubscript{2}
   \underline{1001100}_2 \rightarrow 0100\textsubscript{2}
8. Find the original bits given the following SEC Hamming Code: 010011010000110\textsubscript{2}
   Parity group 1: okay
   Parity group 2: error
   Parity group 4: okay
   Parity group 8: error
Incorrect bit: $2 + 8 = 10$, change bit 10 from 0 to 1: $0100110101001102 \rightarrow 011001001102$

**RAID**

Fill out the following table:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Pro / Good for…</th>
<th>Con / Bad for…</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID 0</td>
<td>Data disks without check information</td>
<td>No overhead</td>
</tr>
<tr>
<td>RAID 1</td>
<td>Mirrored Disks: Extra copy of disks</td>
<td>Fast read / write</td>
</tr>
<tr>
<td>RAID 4</td>
<td>Transfer units = a sector within a single disk Errors are detected within a single transfer unit Independent reads/writes per disks</td>
<td>Higher throughput of small reads</td>
</tr>
<tr>
<td>RAID 5</td>
<td>Check information is distributed across all disks in a group</td>
<td>Higher throughput of small writes</td>
</tr>
</tbody>
</table>

**Note:** RAID 2 and 3 are conceptually the same as RAID 4, but with bit-striping and byte-striping instead.

Small accesses = an access to a single disk in a group

**I/O**

1. Fill this table of polling and interrupts.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
<th>Pro/Good for</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>Polling</td>
<td>Forces the hardware to wait on ready bit (alternatively, if timing of device is known – the ready bit can be polled at the frequency of the device). It basically means manually checking the ready bit regularly.</td>
<td>PRO: -easy to write -poll handler does not have excessively high overhead -deterministic -doesn’t require additional hardware Good for: -Mouse, keyboard</td>
<td>Infeasible on hardware with fast transfer rates that is actually rarely ready (e.g. Ethernet card receiver)</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Hardware fires an “exception” when it becomes ready. CPU changes SPC to execute code in the interrupt handler when this occurs.</td>
<td>PRO: -Necessary for fast devices that are rarely ready. Good for: Fast devices - Hard drives, Network cards</td>
<td>-nondeterministic when interrupt occurs -interrupt handler has some overhead (saves all registers), meaning polling can actually be faster for slow, often ready devices such as mice</td>
</tr>
</tbody>
</table>
2. Memory Mapped I/O
   Certain memory addresses correspond to registers in I/O devices and not normal memory.
   **0xFFFF0000 – Receiver Control:**
   Lowest two bits are interrupt enable bit and ready bit.
   **0xFFFF0004 – Receiver Data:**
   Received data stored at lowest byte.
   **0xFFFF0008 – Transmitter Control**
   Lowest two bits are interrupt enable bit and ready bit.
   **0xFFFF000C – Transmitter Data**
   Transmitted data stored at lowest byte.

   Write MIPS code to read a byte from the receiver and immediately send it to the transmitter.

   ```mips
   lui $t0 0xffff
   receive_wait: #poll on ready of receiver
       lw $t1 0($t0)
       andi $t1 $t1 1
       beq $t1 $zero receive_wait
   lb $t2 4($t0) #load data
   ```

   ```mips
   transmit_wait: #poll on ready of transmitter
       lw $t1 8($t0)
       andi $t1 $t1 1
       beq $t1 $zero transmit_wait
   #write to transmitter
   sb $t2 12($t0)
   ```