CS 61C Spring 2019

I/O, Dependability

Discussion 14: May 1, 2019

1 Polling & Interrupts

1.1 Fill out this table that compares polling and interrupts.

Operation	Definition	Pro/Good for	Con
Polling	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).	• Low Latency • Low overhead when data is available • Good For: devices that are always busy or when you cant make progress until the device replies	• Cant do anything else while polling • Cant sleep while polling (CPU always at full speed)
Interrupts	Hardware fires an "exception" when it becomes ready. CPU changes PC register to execute code in the interrupt handler when this occurs.	• Can do useful work while waiting for response • Can wait on many things at once • Good for: Devices that take a long time to respond, especially if you can do other work while waiting.	• Nondeterministic when interrupt occurs • interrupt handler has some overhead (e.g. saves all registers, flush pipeline, etc.) • Higher latency per event • Worse throughput

2 Memory Mapped I/O

2.1 For this question, the following addresses correspond to registers in some I/O devices and not regular user memory.

- `0xFFFF0000`—Receiver Control: LSB is the ready bit (in the context of polling), there may be other bits set that we dont need right now.
- `0xFFFF0004`—Receiver Data: Received data stored at lowest byte.
- `0xFFFF0008`—Transmitter Control: LSB is the ready bit (in the context of polling), there may be other bit set that we dont need right now.
- `0xFFFF000C`—Transmitter Data: Transmitted data stored at lowest byte.

Recall that receiver will only have data for us when the corresponding ready bit is 1, and that we can only write data to the transmitter when its ready bit is 1.

Write RISC-V code that reads byte from the receiver (busy-waiting if necessary) and writes that byte to the transmitter (busy-waiting if necessary).

```
                lui t0 0xffff0
receive_wait:   lw t1 0(t0)
                andi t1 t1 1            # poll on ready of receiver
                beq t1 x0 receive_wait
                lb t2 4(t0)             # load data
transmit_wait:  lw t1 8(t0)             # poll on ready of transmitter
                andi t1 t1 1
                beq t1 x0 transmit_wait # write to transmitter
                sb t2 12(t0)
```

3 Hamming ECC

Recall the basic structure of a Hamming code. We start out with some bitstring, and then add parity bits at the indices that are powers of two (1, 2, 8, etc.). We don't assign values to these parity bits yet. **Note that the indexing convention used for Hamming ECC is different from what you are familiar with.** In particular, the 1 index represents the MSB, and we index from left-to-right. The ith parity bit $P\{i\}$ covers the bits in the new bitstring where the *index* of the bit under the aforementioned convention, j, has a 1 at the same position as i when represented as binary. For instance, 4 is `0b100` in binary. The integers j that have a 1 in the same position when represented in binary are 4, 5, 6, 7, 12, 13, etc. Therefore, $P4$ covers the bits at indices 4, 5, 6, 7, 12, 13, etc. A visual representation of this is:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗		
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗		...
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗	
	p8								✗	✗	✗	✗	✗	✗	✗	✗						
	p16																✗	✗	✗	✗	✗	

Source: `https://en.wikipedia.org/wiki/Hamming_code`

3.1 How many bits do we need to add to 0011_2 to allow single error correction?

3 parity bits

3.2 Which locations in 0011_2 would parity bits be included?

Using P to represent parity bits: $PP0P011_2$

3.3 Which bits does each parity bit cover in 0011_2?

Parity bit 1: 1, 3, 5, 7
Parity bit 2: 2, 3, 6, 7
Parity bit 3: 4, 5, 6, 7

3.4 Write the completed coded representation for 0011_2 to enable single error correction. Assume that we set the parity bits so that the bits they cover have even parity.

1000011_2

3.5 How can we enable an additional double error detection on top of this?

Add an additional parity bit over the entire sequence.

3.6 Find the original bits given the following SEC Hamming Code: 0110111_2. Again, assume that the parity bits are set so that the bits they cover have even parity.

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_2
$0110011_2 \rightarrow 1011_2$

3.7 Find the original bits given the following SEC Hamming Code: 1001000_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_2
$1001100_2 \rightarrow 0100_2$

4 RAID

4.1 Fill out the following table:

	Configuration	**Pro/Good for**	**Con/Bad for**
RAID 0	Split data across multiple disks	No overhead, fast read / write	Reliability
RAID 1	Mirrored Disks: Extra copy of data	Fast read / write, Fast recovery	High overhead → expensive
RAID 2	Hamming ECC: Bit-level striping, one disk per parity group	Smaller overhead	Redundant check disks
RAID 3	Byte-level striping with single parity disk.	Smallest overhead to check parity	Need to read all disks, even for small reads, to detect errors
RAID 4	Block-level striping with single parity disk.	Higher throughput for small reads	Still slow small writes (A single check disk is a bottleneck)

RAID 5	Block-level striping, parity distributed across disks.	Higher throughput of small writes	The time to repair a disk is so long that another disk might fail in the meantime.