
Q1.

You're developing an app that will allow people to take photos of colour-coded stickers to identify real-world objects in a virtual treasure hunt game.

(a) If the colour codes are made up of 4 coloured squares, each of which can be one of 3 colours, how many unique objects can you support in the game?

> **Solution:** With 4 squares of 3 colours there are $3^4 = 81$ possible codes, so the game can support up to 81 unique objects.
>
> Or you could just look at the next part of the question (don't expect exam questions to give things away quite that easily–plus you still need to show your reasoning in the exam)

(b) If the app also requires a unique *binary* code for each of the 81 items, how many bits are required in that code?

> **Solution:** One way to determine this is by repeatedly trying possible powers of 2: $2^6 = 64$, which is too small. $2^7 = 128$, which gives us more than enough codes. So 7 bits would be required in the code.
>
> A more direct approach is to apply some basic maths. To find the exponent we need to take log_2 of the number of items (since $x = log_2(2^x)$). If your calculator doesn't support log_2, you can make use of the fact that $log_2 x = \frac{log_{10} x}{log_{10} 2}$ (more generally, $log_N x = \frac{log_{10} x}{log_1 0 N}$). So
>
> $$\begin{aligned} n\_bits &= log_2(81) \\ &= \frac{log_{10} 81}{log_{10} 2} \\ &= 6.3399 \end{aligned}$$
>
> which rounds up to $n\_bits = 7$ (as a check, $2^{6.3399} = 81$).
>
> *Note that you won't be allowed to have a calculator in the final. So unless you can do logarithms in your head, the "try powers of two" method is what you'll likely use in the exam if there's a question like this.*

(c) Your app will be working with integers represented in 4-bit *2's complement* format, for which you need to implement the arithmetic. To make sure you understand two's complement arithmetic well enough to code the arithmetic functions, work through the summation $sum = 5 + 4 + 2 - 8$ in binary.

> **Solution:**
>
> ```
> +5      0101
> +4      0100
> +2      0010
> +(-8)   1000
> ```

```
-------------------
sum   1 0011
```

Just retaining the least significant 4 bits (i.e., ignoring the carried 1) we get `sum = 0011 = +3` (which is what we'd expect if we computed the sum in base 10).

(d) Part of your app is written in C, and includes the declarations:

```
int8_t  var8bit_signed;
int16_t  var16bit_signed;
uint8_t  var8bit_unsigned;
```

What are the maximum +ve and maximum -ve numbers that can be represented in each variable?

Solution: For N-bit variables, signed values range from -2^{N-1} to $2^{N-1}-1$. Unsigned values range from 0 to $2^N - 1$

```
-128    <= var8bit_signed    <= +127
-32768  <= var16bit_signed   <= +32767
 0      <= var8bit_unsigned  <= +255
```

Q2.

You're working on the control system for a quadrotor drone. Part of the controller is designed to be highly reliable, so it has redundant components. The controller should only be active if at least two out of the three redundant components are active, so you need to implement a triple-majority voting scheme to turn the controller on or off.

(a) Draw the truth table for $z = majority(a, b, c)$, i.e, z is TRUE (1) only if a majority (at least 2 out of 3) of the inputs is TRUE.

Solution: You can construct the truth table by putting a 1 in the output column on any row where at least 2 of the input columns contain a 1:

a	b	c	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(b) Write a small C function that implements the majority vote using C's logical operators.

Solution: From the truth table, $z = a(b + c) + bc$:

- The last 3 rows of the truth table are 1. Looking at the inputs, these are rows where $a = 1$ and either or both of b and c are 1 (thus $a(b+c)$).
- Both rows in which $b = 1$ AND $c = 1$ (row 4 and row 8) have a 1 in the output column (thus bc).

In C the Boolean expression above translates to:

```
#include <stdbool.h>

bool majority(bool a, bool b, bool c) {
    return (a && (b || c)) || (b && c);
}
```

Some other alternative expressions of the same logical function are:

- $z = ab + ac + bc$ (i.e., `(a && b) || (a && c) || (b && c)`)
- $z = \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$

(c) You decide that you want to keep track of the state of the control system (three possible states: fully functional, one component down, system failure). How many bits do you need to encode the three possible states as a binary sequence?

Solution: Three states can be encoded using just 2 bits (since $2^2 = 4 > 3$).

(d) Your team lead asks you to specify the code used to represent the control system states (i.e., to create an assignment of states to binary sequences). As a design constraint, the lead wants the code be one that will let your hardware designer use an already-designed majority function circuit to generate one bit of the code. Write out a code (a table of sequences and the state that each sequence represents), and also write Boolean expressions in terms of a, b, c, and $majority(a, b, c)$, for the individual bits, b_i, of the code.

Solution:

If we use the code

b_1	b_0	State
0	0	System failure
0	1	One down
1	1	Fully functional

then the Boolean expressions for the code bits could be

$$\begin{aligned} b_1 &= abc \\ b_0 &= majority(a, b, c) \end{aligned}$$

Q3.

You're developing a real-time application for a custom microprocessor design. The microprocessor you're working with has a single 12-bit address bus to access memory, and a single 16-bit data bus to transport data to and from memory.

(a) What must the *memory architecture* of this custom microprocessor be? Explain your reasoning.

> **Solution:** Since there is only a single address bus and a single data bus, there must be only a single memory. Therefore, the microprocessor must be using a *von Neumann* (aka Princeton) memory architecture.

(b) How many *bytes* of memory can the custom microprocessor address?

> **Solution:** The address space is determined by the size of the *address bus*. Since the address bus carries a multi-bit binary sequence, we can determine the number of possible addresses by figuring out how many possible values the binary sequence can represent. A 12-bit address bus implies that a total of $S = 2^{12} = 4096$ bytes of memory are in the address space (memory is addressed in bytes, regardless of wordlength)

(c) Based on the information you have, and the purpose of the Program Counter (PC) register, what size do you expect the PC register to be?

> **Solution:** Since the PC register stores the address of the next instruction to be executed, it must be capable of containing an address. The address bus is specified as being 12 bits, so addresses are 12-bit binary values. In the absence of any other information, it seems reasonable to expect that the PC register, which must contain an address, will be a 12 bit register.

(d) How many read operations would it take to read a 64-bit floating point number from memory into a General Purpose Register?

> **Solution:** Data moves from memory into GPRs via the data bus. Given a 16-bit data bus, which allows reading 2 bytes at a time, reading a 64-bit value from memory would take $64/16 = 4$ reads.

Q4.

Fig. 1 shows a Set-Reset (SR) Latch built using two cross-coupled NOR gates. Note that a NOR gate implements a NOT-OR operation (i.e., it outputs a 1 when both inputs are 0, and a 0 otherwise).

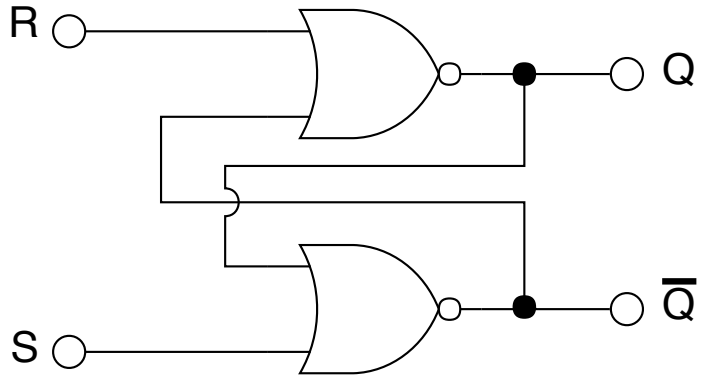
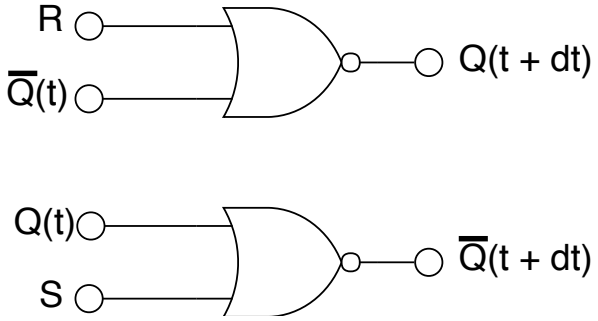


Figure 1: An SR Latch implemented using NOR gates

(a) If $Q(t)$ and $\overline{Q}(t)$ represent the outputs at a time t, write Boolean expressions for $Q(t + \Delta t)$ and $\overline{Q}(t + \Delta t)$ (the outputs some small amount of time, Δt, later) in terms of S, R, $Q(t)$, and $\overline{Q}(t)$.

Solution: Because an SR Latch is *sequential logic* (i.e., it contains feedback loops), finding Boolean expressions is a little more complicated than it would be for combinational logic. The outputs depend on the inputs, but also on ... the outputs. We can resolve this circularity by thinking of the output *now* ($Q(t)$) as being an input that allows us to determine the output *just after now* ($Q(t + \Delta t)$). That lets us "unfold" the circuit, like this:



Reading off the Boolean expressions from Fig. 1, we see that the Q output is a NOR of the R input and the $\overline{Q}$ output. Similarly, $\overline{Q}$ is a NOR of the S input and the Q output. Thus:

$$\begin{aligned} Q(t + \Delta t) &= \overline{(R + \overline{Q}(t))} \\ \overline{Q}(t + \Delta t) &= \overline{(S + Q(t))} \end{aligned}$$

(b) Given a situation in which the initial outputs are $Q(t) = 0$, $\overline{Q}(t) = 1$, and the inputs are $S = 1$, and $R = 0$, iteratively evaluate your expressions for $Q(t + \Delta t)$

and $\overline{Q}(t+\Delta t)$ until $Q(t)$ and $\overline{Q}(t)$ reach a *stable state*. Here, a "stable state" of the SR Latch is defined as $Q(t+\Delta t) = Q(t)$ and $\overline{Q}(t+\Delta t) = \overline{Q}(t)$. What are the final values of $Q(t)$ and $\overline{Q}(t)$?

Solution: Given the expressions found in the previous part, we can iteratively evaluate the outputs and use those to determine the inputs for the next iteration. If you look in the lecture 3 Python examples on Learn you'll see an implementation of this kind of iteration.

Inputs: $S = 1$ and $R = 0$

Initial state: $Q(t) = 0$ and $\overline{Q}(t) = 1$

In the first iteration we find $Q(t+\Delta t)$ and $\overline{Q}(t+\Delta t)$ using the initial values of $Q(t)$ and $\overline{Q}(t)$:

$$\begin{aligned} Q(t+\Delta t) &= \overline{(R+\overline{Q}(t))} = \overline{(0+1)} = 0 \\ \overline{Q}(t+\Delta t) &= \overline{(S+Q(t))} = \overline{(1+0)} = 0 \end{aligned}$$

Now, using $Q(t) = 0$ and $\overline{Q}(t) = 0$, the second iteration is:

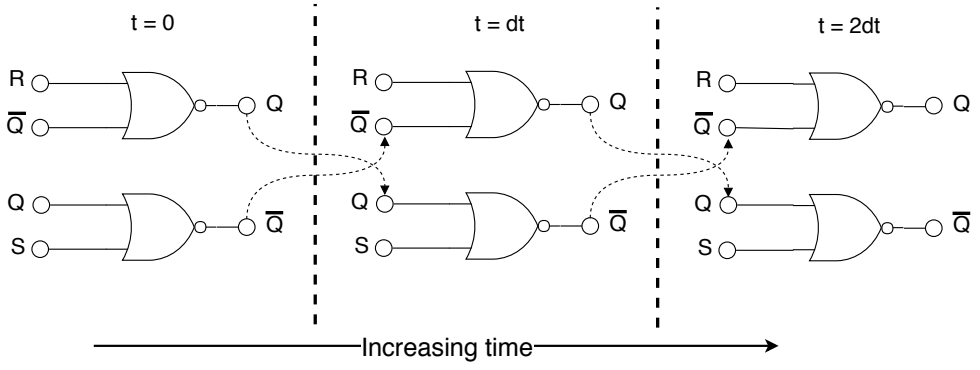
$$\begin{aligned} Q(t+\Delta t) &= \overline{(R+\overline{Q}(t))} = \overline{(0+0)} = 1 \\ \overline{Q}(t+\Delta t) &= \overline{(S+Q(t))} = \overline{(1+0)} = 0 \end{aligned}$$

Third iteration:

$$\begin{aligned} Q(t+\Delta t) &= \overline{(R+\overline{Q}(t))} = \overline{(0+0)} = 1 \\ \overline{Q}(t+\Delta t) &= \overline{(S+Q(t))} = \overline{(1+1)} = 0 \end{aligned}$$

Since $Q(t+\Delta t) = Q(t)$ and $\overline{Q}(t+\Delta t) = \overline{Q}(t)$ the state is now "stable" and we can stop iterating. The final outputs are $Q(t) = 1$ and $\overline{Q}(t) = 0$

An Additional Thought: Thinking in terms of our "unfolded" SR Latch, we're essentially forming a chain of logic circuits. Except that, because we're using feedback, we're building that chain by using time (repeatedly using the same gates) instead of by using space (repeatedly implementing the same gate circuit):



 END OF QUESTIONS