# Isabelle coursework exercises

John Wickerson

Autumn term 2021

There are three tasks and they are ordered in roughly increasing order of difficulty. Tasks are independent from each other, so failure to complete one task should have no bearing on later tasks. Tasks labelled (⋆) are expected to be reasonably straightforward. Tasks labelled (⋆⋆) should be manageable but may require quite a bit of thinking, and it may be necessary to consult additional sources of information, such as the Isabelle manual and Stack Overflow. Tasks labelled (⋆⋆⋆) are more ambitious still.



**Marking principles.** It is not expected that students will complete all parts of all the tasks. Partial credit will be given to partial answers. If you are unable to complete a proof, partial credit will be given for explaining your thinking process in the form of `(*comments*)` in the Isabelle file.



**Submission process.** You are expected to produce a single Isabelle theory file called `YourName.thy`. This file should contain all of the definitions and proofs for all of the tasks below that you have attempted.

**Plagiarism policy.** You **are** allowed to consult the coursework tasks from previous years – the questions and model solutions for these are available. You **are** allowed to consult internet sources like Isabelle tutorials. You **are** allowed to work together with the other student in your pair. You **are** allowed to ask questions on Stack Overflow or the Isabelle mailing list, but make your questions generic (e.g. "Why isn't the `subst` method working as I expected?"); please **don't** ask for solutions to these specific tasks! And please **don't** share your answers to these tasks outside of your own pair. If you would like to share your answers to these tasks publicly, e.g. on a public GitHub repo, you

are welcome to do so, but please check with me first, because some students may still be working on the coursework with an extended deadline.

**Task 1 ($\star$)** This task builds on the `circuit` datatype from the worksheet. We shall add an extra optimisation that exploits the following Boolean identities:

$$\begin{aligned}(a \vee b) \wedge (a \vee c) &\equiv a \vee (b \wedge c)\\ (a \vee b) \wedge (c \vee a) &\equiv a \vee (b \wedge c)\\ (a \vee b) \wedge (b \vee c) &\equiv b \vee (a \wedge c)\\ (a \vee b) \wedge (c \vee b) &\equiv b \vee (a \wedge c)\end{aligned}$$

In `HSV_tasks_2021.thy`, you are given a function called `factorise`, which traverses a given circuit looking for opportunities to apply those identities (in the left-to-right direction). Each time the identity is applied, one OR-gate is removed from the circuit, thus reducing its area.



1. You have been given three test cases in `HSV_tasks_2021.thy` that show how `factorise` works. Write another test case that demonstrates a circuit from which `factorise` is able to remove *two* OR-gates.
2. Use Isabelle to prove that `factorise` is correct. That is, prove for any circuit $c$ that `factorise(`$c$`)` has the same input/output behaviour as $c$.
3. Write an improved version of `factorise`, called `factorise2`, that is *additionally* able to exploit the following identities:

$$\begin{aligned}(a \wedge b) \vee (a \wedge c) &\equiv a \wedge (b \vee c)\\ (a \wedge b) \vee (c \wedge a) &\equiv a \wedge (b \vee c)\\ (a \wedge b) \vee (b \wedge c) &\equiv b \wedge (a \vee c)\\ (a \wedge b) \vee (c \wedge b) &\equiv b \wedge (a \vee c)\end{aligned}$$

4. Use Isabelle to prove that `factorise2` is correct.

**Task 2 ($\star\star$)** This task concerns the following theorem:

> **Theorem.** *For all integers $a$ and $b$ and natural numbers $n$, $a^{2n+1} + b^{2n+1}$ is divisible by $a + b$.*

1. Sketch a proof of the theorem. (Hint: use mathematical induction on $n$.)
2. Mechanise your proof in Isabelle.
3. With the help of Isabelle, disprove the following non-theorem by providing a counterexample:

   **Non-theorem.** *For all integers $a$ and $b$ and natural numbers $n$, $a^{2n+2} + b^{2n+2}$ is divisible by $a + b$.*

**Task 3 ($\star\star\star$)** This task concerns the shift-and-add-3 algorithm for converting binary numbers into binary-coded decimal (BCD) numbers.

Let us represent binary numbers as lists of Booleans, and BCD numbers as lists of nibbles, where each nibble is a tuple of four Booleans. In what follows, let us write `0` for `False` and `1` for `True`.

In `HSV_tasks_2021.thy`, you are given a function called `binary_to_nat` that converts a binary number into a natural number. So, for instance,

```
binary_to_nat [0,1,0,1] = 5.
```



You are also given a function called `bcd_to_nat` that converts a BCD number into a natural number. So, for instance,



```
bcd_to_nat [(0,0,1,1),(0,0,0,0)] = 30.
```

1. Implement in Isabelle a function called `binary_to_bcd` that uses the shift-and-add-3 algorithm to convert a binary number into the equivalent BCD number. For instance,

   ```
   binary_to_bcd [0,1,0,1,0,1]
   ```

   should return `[(0,0,1,0),(0,0,0,1)]`.
2. Use Isabelle to prove that whenever `binary_to_bcd` produces a list of nibbles, each of those nibbles corresponds to a valid BCD digit (i.e. its value is in the range 0–9).
3. Use Isabelle to prove that `binary_to_bcd` is correct. That is, prove for all binary numbers $b$ that

   ```
   bcd_to_nat (binary_to_bcd b) = binary_to_nat b.
   ```