Computer Science

Verification of Combinational and
Sequential Circuits in LEAN3

Zahir A. Bingen
S2436647

Supervisors:
Drs. H.A. Hiep
Dr. A.W. Laarman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

24/08/2023

Abstract

In this thesis, we present a study on the formal specification and formal verification of combinational and sequential circuits utilizing the interactive theorem prover LEAN3. The main focus of this research is to gain insight into specifying circuits, effectively implementing these specifications, and verifying circuit implementations to ensure compliance with the given specifications. Initially, we examine fundamental circuits such as logic gates, multiplexers, adders, and memory. Subsequently, we explore a more complex circuit, namely a sequence recognizer implemented as a finite state machine, capable of identifying the bit pattern '101' in a data stream. The efforts in verifying circuits using LEAN3 have demonstrated that the process, depending on the amount of experience, can be highly time-consuming. While certain proof aspects such as rewriting and simplifying can be automated, the verification may need human guidance, depending on the complexity of the specification and implementation of the circuit. Taking all of this into consideration, we will reflect on the advantages and disadvantages of formal verification.

Acknowledgements

I would like to thank both my supervisors Hans-Dieter Hiep and Alfons Laarman for providing me the guidance, support and supervision needed to complete this thesis. Furthermore, I would like to thank the LEAN community for providing me with advice on tackling technical difficulties I encountered during my research. Finally, I would like to thank my family for their support and giving me the opportunity to get to the point where I am right now.

Contents

1 Introduction **1**

- 1.1 Research Motivation . . . 1
- 1.2 Specifications and Implementations . . . 3
- 1.3 LEAN3 Theorem Prover . . . 4
- 1.4 Related Work . . . 5

2 Methodology **6**

3 Combinational Circuits **7**

- 3.1 Logic Gates . . . 7
  - 3.1.1 NOT . . . 7
  - 3.1.2 OR . . . 9
  - 3.1.3 AND . . . 11
  - 3.1.4 XOR . . . 13
- 3.2 Multiplexer (2-to-1) . . . 16
  - 3.2.1 Specification . . . 16
  - 3.2.2 Specification Functionality . . . 16
  - 3.2.3 Implementation (1-bit inputs) . . . 17
  - 3.2.4 Implementation (n-bit inputs) . . . 17
  - 3.2.5 Verification . . . 18
- 3.3 Full Adder . . . 20
  - 3.3.1 Specification . . . 20
  - 3.3.2 Specification Functionality . . . 21
  - 3.3.3 Implementation full binary adder . . . 22
  - 3.3.4 Implementation full adder . . . 22
  - 3.3.5 Verification . . . 23
- 3.4 4-bit Shifter . . . 25
  - 3.4.1 Specification . . . 25
  - 3.4.2 Specification Functionality . . . 26
  - 3.4.3 Implementation . . . 26
  - 3.4.4 Verification . . . 27

4 Sequential Circuits **29**

- 4.1 Signals . . . 29
- 4.2 Memory . . . 29
  - 4.2.1 Specification . . . 30
  - 4.2.2 Specification Functionality . . . 31
  - 4.2.3 Implementation . . . 33
  - 4.2.4 Verification . . . 34
- 4.3 Program Counter . . . 35
  - 4.3.1 Specification . . . 35
  - 4.3.2 Implementation . . . 35
  - 4.3.3 Verification . . . 36

5 Case Study: Verification of a FSM **37**
5.1 FSM Schematic / Design 37
5.2 Specification 38
5.3 Implementation 39
5.4 Verification 40

6 Conclusions and Future Work **42**
6.1 Conclusion 42
6.2 Discussion and Future Work 42

References **44**

1 Introduction

Digital circuits are the foundation of all modern electronic devices. These circuits are made to execute logical operations using transistors, which act as electrical switches to manipulate digital signals. The basic idea stems from Claude Shannon and his thesis published in 1938, 'A Symbolic Analysis of Relay and Switching Circuits' [Sha38]. Shannon introduced the concept of using electrical switches to solve boolean algebra problems. Our interest in digital circuits comes from the fact that the reliance on digital circuits in our current society grows and ensuring the correctness of these circuits becomes increasingly vital.

This thesis serves as a guide for learning how to formally verify both combinational (combinatorial) and sequential circuits using the interactive theorem prover LEAN3. Written in a tutorial style, it is designed to be accessible to fellow students, even those who may not have had any experience with interactive theorem provers before. We will explore some basic combinational and sequential circuits, followed by a more complex finite state machine. We will use the term 'combinational' instead of 'combinatorial' in this thesis, as this is the terminology used in many of the sources we will be referencing.

In this section we will introduce our research motivation (Section 1.1). We will explain what specifications and implementations are (Section 1.2) and give a brief introduction to the LEAN3 theorem prover (Section 1.3).

1.1 Research Motivation

Digital circuits are crucial for modern electronic devices and systems, and the correctness of these circuits is of importance. There are different correctness criteria, such as 'functionality and reliability', which ensure that the circuit behaves as intended. Other criteria include 'safety and security', which aim to prevent adverse side-effects or the leakage of sensitive information. An example illustrating the failure to meet the 'functionality and reliability' criteria is the Intel Pentium FDIV bug, resulting from a design flaw in the lookup table of the floating-point unit (FPU). Five entries of this lookup table contained the wrong data, which caused incorrect results for specific division operations for floating point numbers. It resulted Intel a loss of 475 million dollars [Cor]. An example illustrating the failure to meet the 'safety and security' criteria is the AMD 'Zenbleed' bug. In specific microarchitectural conditions, when the 'vzeroupper' instruction was used and a branch misprediction occured, the processor incorrectly rolled back the instruction. This caused the upper part of the YMM register to be in an undefined state. This part of the register then contained random data from the register file, which allowed the registers of other processes to be read and caused a vulnerability where a potential attacker could access sensitive data [AMD, Orm]. In this thesis we restrict ourselves to correctness in the sense of functional correctness. The other criteria are out of scope.

There are two common methods to ensure the correctness of a circuit. These are validation and verification. With validation, we check whether a design conforms to minimal requirements. We use a set of test cases to check whether the design has the correct behavior. This should also give us the confidence that it behaves correctly in other cases that are not tested. However, in most

cases this approach does not give any correctness guarantees. This is because validation might still miss edge cases or unique conditions that lead to errors. The Intel Pentium FDIV and AMD's 'Zenbleed' bugs are notable examples of these oversights. Verification on the other hand is used to prove that for all possible cases, the design conforms to all requirements.

Simulation is the most common method used to date to validate circuit implementations. Given a set of inputs and a reference output, the output of the design is compared with the reference output [Lam05]. For specific input sizes and types of circuits, such as 32-bit counters and multipliers, simulating the entire input space can be time-consuming. Specifically, we need to check 2^{32} inputs for the counter or 2^{64} inputs in the case of a 32-bit multiplier, which can take days to complete [Any]. If we consider circuits of even larger bits, such as 512-bit, then it becomes impractical to simulate all inputs. That is why in practice not all inputs are checked, but only a specific set is tested until one has enough confidence that the circuit is behaving correctly.

Formal verification uses logical properties, expressed in a formal language, and logical techniques such as deduction, to verify that a circuit is correct with respect to a specification. A collection of these properties is called a 'specification' and verification amounts to showing a deduction that all actual properties of a circuit comply to the intended properties as specified. In contrast to simulation, which examines individual outputs for each input, formal verification uses properties to reason about a set of outputs all at once, considering the entire input space. Because of this 'completeness', formal verification can offer us a higher degree of confidence that the implementation is correct.

Unfortunately, there are also some limitations for formal verification. Some examples are: user errors, errors in specification, and software bugs [Lam05]. User errors can for example be over-constraining and incorrectly representing the specifications. Errors in specifications can happen when a low level specification does not fully capture the requirements set by a higher-level specification. Finally, bugs in formal verification software can cause false confirmation of the design.

We will make use of the LEAN3 theorem prover to formally verify combinational and sequential circuits. We have chosen the LEAN3 theorem prover because it is relatively modern and it has an extensive library of lemmas. LEAN3 also provides a very well documented user manual. Similar work on verifying circuits has been done in the theorem provers HOL [SH17], PVS [ORSS92] and Nuprl [Lee92]. In Section 1.4 we will briefly discuss the difference between this thesis and the already existing work.

This brings us to the following research question:

> *"How can we verify the correctness of a combinational and sequential circuit implementation, with respect to a given formal specification, using the LEAN3 interactive theorem prover?"*.

The exact methodology we apply to answer this question is described in Section 2.

1.2 Specifications and Implementations

In this thesis we make a distinction between specifications and implementations. Specifications describe the high-level behavior of a system, in our case a circuit. The specifications contain the relationship of the various values that can be observed at specific data lines in a circuit, without prescribing how the implementation should achieve that behavior. These specifications are propositions, type 'Prop' in the LEAN3 theorem prover. Because the specifications are defined in 'Prop', they do not have a direction of input and output. On the other hand, implementations are represented as computable functions in LEAN3, which do have an input and output direction. An implementation is a computable function that provides the clear steps to achieve the behavior described in the specification. In essence, this means that it is possible for two implementations, with both different input and output direction, to satisfy the same specification.

We will illustrate that two implementations with different input and output directions can satisfy the same specification with a small example. The specification is defined by having 3 wires `A`, `B`, and `C`. These wires must satisfy the logical relation `A XOR B` = `C`. The first implementation has inputs `A` and `B`, and output `C`. This implementation calculates the output `C` by applying the XOR operation on inputs `A` and `B`. The second implementation has inputs `B` and `C`, and output `A`. The output `A` is given by applying the XOR operation on inputs `B` and `C`. Both of these implementations satisfy the specification given, because of the commutative property of XOR and the equivalence of the relations `A XOR B` = `C`, `B XOR C` = `A`, and `A XOR C` = `B`. The equivalence of these relations can be derived from the table in Table 1. By choosing a pair of input values and observing the corresponding output value, it becomes clear that results of the XOR operation are consistent with these relations.

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: Table of wires A, B, C, and their values

An important difference between implementations and specifications is that the implementations must be computable. On the other hand, specifications can require constraints which are not necessarily computable.

There are two kinds of specifications, namely under-specifications and over-specifications. Under-specifications are specifications that do not completely describe the expected behavior. Some constraints in an under-specification are left out which gives the implementations the freedom to define these unspecified cases as they see fit. Over-specifications are specifications which have 'too strict' constraints. This means that the specification has specific constraints, which cannot be implemented in practice. An example can be a circuit that has to operate at an unattainable high speed. We will not consider over-specifications in this thesis.

1.3 LEAN3 Theorem Prover

LEAN3 is an interactive theorem prover that has been developed by Microsoft Research and community contributors. The main goal of LEAN3 is to help computer scientists and mathematicians to create and verify mathematical proofs. LEAN3 provides an extensive library of lemmas which can be used to construct proofs. We will give a short explanation on how to define functions, theorems and how to prove those theorems, by the means of an example which illustrates that adding two even numbers results in an even number.

We will start by defining a predicate that asserts a number `n` to be even.

```
def is_even (n : ℕ) : Prop :=
  ∃ k, n = 2*k

def is_even' : ℕ → Prop
  | n := ∃ k, n = 2*k
```

Figure 1: Definition is_even

In Figure 1 we see two methods by which we can define functions using the `def` keyword. The function `is_even` takes an explicit argument `n`, and returns a proposition that defines the property of that number being even. The second function `is_even'` uses pattern matching to match the argument `n`. This method is useful if we want to define a function by recursion or need multiple separate cases.

```
theorem even_plus_even {a b : ℕ} (h1 : is_even a) (h2 : is_even b) : is_even (a + b) :=
begin
    cases h1 with k hk,
    cases h2 with l hl,
    existsi k + l,
    rw hk,
    rw hl,
    rw mul_add,
end
```

Figure 2: Theorem `even_plus_even`

The theorem in Figure 2 is defined with the `theorem` keyword. It has two hypotheses `h1` and `h2` in the declaration, which assumes that `a` and `b` are even. After the last `:` symbol, we have our goal, which expresses that `a + b` is even. Within the `begin` and `end` keywords is the part where we can interactively construct our proof through the use of tactics. The exact meaning and use of these tactics can be found in the official documentation of LEAN3 [Pro], while the definitions of the imported lemmas can be found in the documentation of the Mathlib library [Com].

In the context of this thesis, LEAN3 will be utilized to formalize circuit specifications, implementing circuits and proving that the implementations comply with the specifications.

1.4 Related Work

Quite some research has been done on the verification of combinational and sequential circuits using theorem provers. One of the examples is the research by Shiraz and Hasan [SH17]. In their research, they created a library in HOL that can be used for theorem proving. Specifications and implementations were formulated for combinational circuits, but the implementations were not computable functions. The implementations were propositions that described the structural connections between components. There was no distinct difference between specifications and implementations. In essence, they have shown that the specifications are equivalent to each other. Constrastingly, in our research, we will clearly outline the differences between specifications and implementations and we will have a proof obligation that the implementation is computable.

In a paper by Leeser they used Nuprl to verify hardware [Lee92]. One of the examples they have shown was for a sequential circuit, namely a program counter. They defined behavioral specifications for subcomponents of this counter, which they said could be proven to be implemented at a lower level of detail such as a latch. Furthermore, they gave a proof of the correctness of this circuit. In contrast, our thesis implements a program counter and not just verifies the equivalence of specifications. Our implementation of a program counter will consist of a memory module with a built in full adder.

In a thesis by Mario Carneiro called 'The Type Theory of Lean' [Car], he presents the dependent type theory which Lean is built on. The axiom system is shown in complete detail, including optional features such as let binders and definitions. Carneiro also provides a reduction of the theory that uses a fixed set of inductive types, which can be helpful for people who are studying the framework. Furthermore, Carneiro explores the metatheory of the Lean theorem prover and proves the unique typing of the definitional equality. In our thesis we make use of the LEAN3 interactive theorem prover which is built upon the dependent type theory detailed by Carneiro. His work is a useful resource for those wanting to understand the foundation of LEAN3.

2 Methodology

In this section we will describe the methodology that we have applied in this thesis. Let us recall the research question:

> *"How can we verify the correctness of a combinational and sequential circuit implementation, with respect to a given formal specification, using the LEAN3 interactive theorem prover?"*.

The first step we take is creating a formal specification. We can accomplish this through multiple methods. One possibility is directly translating truth tables in LEAN3, or trying to find a relationship within these truth tables and making a specification out of this using symbolic logic. We can also enumerate properties of the behavior until we feel confident enough that we have the correct specification. In essence, we translate the informal description to a formal description.

Recall that specifications do not have an input and output direction. Therefore, in the next step, we will analyse the properties of the formal specification and decide which input and output direction could be suitable. For this chosen input and output direction, we will investigate if there exists an output for all inputs (existence) and if this output is precisely defined (unique). We will accomplish this by providing an existence and uniqueness proof in LEAN3.

In the following step we will investigate whether there exists a computable implementation designated to comply to the formal specification. We will do this by providing a functional implementation in LEAN3. If we succeed in giving an implementation, then we also know that the output is computable given the inputs. If we are not able to give an implementation, then it might be the case that there exists no computable implementation for our specification. This comes from the fact that our specifications can have requirements, that do not necessarily have to be computable.

Finally, we will try to provide a proof in LEAN3 which shows that the implementation complies with the specification. In some cases we are able to prove a stronger property which shows that the specification precisely defines the implementation, there is only one.

3 Combinational Circuits

Combinational circuits are circuits that only depend on their current inputs, and not on memory or previous inputs and outputs. In real-world applications, these circuits may have delays due to factors such as signal propagation and gate switching. We will abstract away from these details and assume that the circuits have no delay.
We will take a look at logic gates (Section 3.1), a 2-to-1 multiplexer (Section 3.2), a full adder (Section 3.3) and a 4-bit shifter (Section 3.4).

3.1 Logic Gates

Logic gates are fundamental building blocks in the design of combinational and sequential circuits. These logic gates operate on binary signals to produce another binary signal. There exist multiple logic gates such as: `OR`, `AND`, `XOR`, `NOT`, `NAND`, `NOR` and `XNOR` gates. In this thesis we will only implement `OR`, `AND`, `XOR` and `NOT` gates in LEAN3. While there are other gates such as the universal gates `NAND` and `NOR` which are capable of implementing any boolean function [End01], we will limit ourselves to the previously mentioned gates as they can be combined into any other gate.
In this section, we will transition from logic gates with a fixed number of inputs, to an arbitrary amount of inputs. In the paper by Sumayya Shiraz and Osman Hasan [SH17] they used a recursive approach to define logic gates in the theorem prover HOL. We will use a similar recursive approach where we will implement the logic gates by applying logical operators on a list of boolean values. Furthermore we will give specifications for the behavior of the gates and a proof that our implementations comply with these specifications.

3.1.1 NOT

A	OUT
0	1
1	0

Table 2: Truth table of a NOT-gate

Table 2 shows the truth table for a NOT-gate. We can see that if the input `A` is 0, then the output must be 1, else if the input `A` is 1 then the output must be 0. The NOT-gate simply inverts the signal and can be specified by the following specification shown in Figure 3.

```
def NOT_spec (A : bool) (OUT : bool) : Prop :=
  out = ¬A
```

Figure 3: Specification of a NOT-gate

Next we will proof the uniqueness of the output of the `NOT_spec`. The theorem in Figure 4 illustrates that given all inputs `A`, there exists a unique output `OUT`. This proof is done by introducing the `A` variable. Then using the `exists_unique_of_exists_of_unique` lemma, we can prove the existence

and uniqueness of the output `OUT`. The existence is trivially proven by applying the `exists_eq` lemma. The uniqueness of the output pair is proven by showing that, for all pairs of outputs y_1 and y_2, if they both satisfy the specification `NOT_spec` for the same inputs, then they must be equal to each other. We do this by introducing the variables y_1 and y_2, unfolding our specification and introducing the hypotheses that assume y_1 and y_2 satisfy the specification. We can show with a simple rewrite that y_1 and y_2 must be the same, finishing our proof. This uniqueness proof is the same for the other logic gates we will see later on, therefore we will not explain this proof again.

```
theorem NOT_unique : ∀ (A : bool), ∃! (OUT : bool),
  NOT_spec A OUT :=
  begin
    intros A,
    apply exists_unique_of_exists_of_unique,
    {
      exact exists_eq,
    },
    {
      intros y₁ y₂,
      unfold NOT_spec,
      intros h₁ h₂,
      rw ←h₂ at h₁,
      exact h₁,
    }
  end
```

Figure 4: Theorem exists unique of `NOT_spec`

To implement the NOT-gate, we can define a function `NOT : bool → bool`. This function matches the input argument with `tt` and `ff`, and returns the complement of the input. The implementation is given in Figure 5.

```
def NOT : bool → bool
  | tt := ff
  | ff := tt
```

Figure 5: Implementation of a NOT-gate

To prove that the implementation of the NOT-gate (`NOT`) complies with the specification (`NOT_spec`), we need to show that the statement in Figure 6 holds.

```
    ∀ (A : bool), ∀ (OUT : bool), NOT A = OUT ⟺ NOT_spec A OUT
```

Figure 6: Equivalence `NOT` and `NOT_spec`

We can complete this proof by considering all possible values of the input `A`. With some unfolding and simplifying of our goal, we can apply the `eq_comm` lemma, which proofs that the left and right side of the equivalence are equal to each other. The proof is shown in Figure 7.

```
theorem NOT_correct : ∀ (A : bool), ∀ (OUT : bool),
  NOT A = OUT ⟺ NOT_spec A OUT :=
begin
  intros A OUT,
  cases A; unfold NOT NOT_spec; simp; rw eq_comm,
end
```

Figure 7: Theorem `NOT` complies with `NOT_spec`

3.1.2 OR

a_1	a_2	...	OUT
0	0	...	0
0	1	...	1
1	0	...	1
1	1	...	1

Table 3: Extended truth table of an OR-gate

The truth table of an OR-gate in Table 3 shows that if any of the inputs a_n is equal to 1, the output of the OR-gate must also be 1. In other words, given a list `A` of boolean values a_n which correspond with the inputs of the OR-gate, there must exist at least one element a_n that is equal to 1 for the output to also be equal to 1. The gives us the following specification shown in Figure 8. The term '`a` ∈ `A`' means that there exists an index within the bounds of the list `A`, such that `a` is equal to the value at that specific index. We have chosen to specify our OR-gate this way because, directly translating the truth table would become unfeasible if the size of the inputs grows larger.

```
def OR_spec (A : list bool) (OUT : bool) : Prop :=
  OUT = (∃ (a ∈ A), a = tt)
```

Figure 8: Specification of an n-bit OR-gate

```
theorem OR_unique : ∀ (A : list bool), ∃! (OUT : bool),
  OR_spec A OUT :=
  begin
    intros A,
    apply exists_unique_of_exists_of_unique,
    {
      exact exists_eq,
    },
    {
      intros y₁ y₂,
      unfold OR_spec,
      intros h₁ h₂,
      rw ←h₂ at h₁,
      exact h₁,
    }
  end
```

Figure 9: Theorem exists unique of `OR_spec`

To recursively implement the OR-gate we can define a function `OR : list bool → bool`. The function applies the $\vee$ operator to the head of the list and recursively calls itself with the tail of the list until it reaches the base case, which is the empty list. In the base case, the function will return the value `ff`. The implementation of this function is given in Figure 10.

```
def OR : list bool → bool
  | [] := ff
  | (h::t) := h ∨ OR t
```

Figure 10: Implementation of an n-bit OR-gate

The final step is to prove that our implementation complies with our specification. This can be done by proving the following statement in Figure 11.

```
    ∀ (A : list bool), ∀ (OUT : bool), OR A = OUT ⟺ OR_spec A OUT
```

Figure 11: Equivalence `OR` and `OR_spec`

To complete the proof, we can use induction on the length of the list `A`. This will cover both the base case where the list is empty and the inductive case where the list is non-empty. The base case is straightforward to prove as it follows directly from the definitions of `OR` and `OR_spec`. For the inductive case, we can use a proof by cases to consider all possible values of the head and tail of the list. We use the finish tactic to automatically find the appropriate lemmas to complete our proof. The complete proof can be found in Figure 12.

```
theorem OR_correct : ∀ (A : list bool), ∀ (OUT : bool),
  OR A = out ⟺ OR_spec A out:=
begin
  intros A OUT,
  induction A,
  { --base case
    finish [OR, OR_spec],
  },
  { --inductive case
    cases A_hd; cases A_tl; finish [OR, OR_spec],
  }
end
```

Figure 12: Theorem `OR` complies with `OR_spec`

3.1.3 AND

a_1	a_2	...	OUT
0	0	...	0
0	1	...	0
1	0	...	0
1	1	...	1

Table 4: Extended truth table of an AND-gate

The truth table of an AND-gate in Table 4 shows that if and only if all of the inputs a_n are equal to 1, the output of the AND-gate must also be `1`. In other words, given a list `A` of boolean values a_n which correspond with the inputs of the AND-gate, all elements in that list must be equal to 1. This results in the following specification illustrated in Figure 13.

```
def AND_spec (A : list bool) (OUT : bool) : Prop :=
  OUT = (∀ (a ∈ A), a = tt)
```

Figure 13: Specification of an n-bit AND-gate

```
def AND_unique : ∀ (A : list bool), ∃! (OUT : bool),
  AND_spec A OUT :=
  begin
    intros A,
    apply exists_unique_of_exists_of_unique,
    {
      exact exists_eq,
    },
    {
      intros y₁ y₂,
      unfold AND_spec,
      intros h₁ h₂,
      rw ←h₂ at h₁,
      exact h₁,
    }
  end
```

Figure 14: Theorem exists unique of `AND_spec`

To recursively implement the AND-gate we can define a function `AND : list bool → bool`. The function applies the ∧ operator to the head of the list and recursively calls itself with the tail of the list until it reaches the base case, which is the empty list. In the base case, the function will return the value `tt`. The implementation of this function is given in Figure 15.

```
def AND : list bool → bool
  | [] := tt
  | (h::t) := h ∧ AND t
```

Figure 15: Implementation of an n-bit AND-gate

To show that our implementation complies with our specification we need to prove the following statement in Figure 16.

```
    ∀ (A : list bool), ∀ (out : bool), AND A = out  ⟺  AND_spec A out
```

Figure 16: Equivalence `AND` and `AND_spec`

The steps to prove this statement are analogue to the one in Section 3.1.2 and are shown in Figure 17.

```
theorem AND_correct : ∀ (A : list bool), ∀ (OUT : bool),
  AND A = OUT ⟺ AND_spec A OUT:=
begin
  intros A OUT,
  induction A,
  { --base case
    finish [AND, AND_spec],
  },
  { --inductive case
    cases A_hd; cases A_tl; finish [AND, AND_spec],
  }
end
```

Figure 17: Theorem AND complies with AND_spec

3.1.4 XOR

a_1	a_2	a_3	...	OUT
0	0	0	...	0
0	0	1	...	1
0	1	0	...	1
0	1	1	...	0
1	0	0	...	1
1	0	1	...	0
1	1	0	...	0
1	1	1	...	1

Table 5: Extended truth table of an XOR-gate

If we take a look at the truth table of an XOR-gate in Figure 5, we notice that the output is 1 if the inputs have an odd number of 1's. This observation is confirmed in [HH07], where they state that an n-bit XOR-gate, also known as a parity gate, produces an output of 1 if an odd number of inputs are 1. Thus can we create our specification by counting the numbers of `tt` values in our input list `A`, and having the output being equal to `tt` if this count is odd. The count function and the specification are given in Figure 18. The exists unique proof of the `XOR_spec` is shown in Figure 19.

```
def count_tt : list bool → ℕ
  | [] := 0
  | (h::t) := (if h = tt then 1 else 0) + count_tt t

def XOR_spec (A : list bool) (OUT : bool) : Prop :=
  OUT = ((count_tt A) % 2 = 1)
```

Figure 18: Specification of a n-bit XOR-Gate

```
theorem XOR_unique : ∀ (A : list bool), ∃! (OUT : bool),
  XOR_spec A OUT :=
  begin
    intros A,
    apply exists_unique_of_exists_of_unique,
    {
      exact exists_eq,
    },
    {
      intros y₁ y₂,
      unfold XOR_spec,
      intros h₁ h₂,
      rw ←h₂ at h₁,
      exact h₁,
    }
  end
```

Figure 19: Theorem exists unique of `XOR_spec`

In Figure 20 we have recursively defined an n-bit XOR-gate. In the case that the list is empty, we return the value `ff`. If the list only contains a single element, then we will return the value of that element. In the last case of our function, we have `h1` representing the first bit in the list, `h2` the second bit and `t` the tail of the list. We apply the definition of a 2-bit XOR-gate on `h1` and `h2`, and replace both entries with the result of this operation. We then recursively call the `XOR` function on the result and the tail of the list. This in essence chains the inputs with 2-bit XOR-gates.

```
def XOR : list bool → bool
  | [] := ff
  | [b] := b
  | (h1::h2::t) := XOR((h1 ≠ h2) :: t)
```

Figure 20: Implementation of a n-bit XOR-Gate

To prove that the implementation of the XOR-gate (`XOR`) complies with the specification (`XOR_spec`), we need to show that the following statement in Figure 21 holds.

```
    ∀ (A : list bool), ∀ (OUT : bool), XOR A = OUT  ⟺  XOR_spec A OUT
```

Figure 21: Equivalence `XOR` and `XOR_spec`

The theorem in Figure 22 proves that the above statement is true. We start by introducing the variables `A` and `OUT`. Intuitively we would do a case analysis on the list `A`, but during the process of constructing the proof, we had the issue that our hypothesis of the list was too strong. Therefore we destruct the list `A` and prove two cases where one case is the empty list, and the other case is the non-empty list. The case where the list is empty is trivial to prove and just requires us to rewrite our hypothesis in our goal and unfolding the definitions. Then we can finish this goal by applying the `eq_comm` lemma. The second case, where the list is non-empty, we prove by induction on the tail of the list `tl`. The base case has the tail of the list being empty. The list only contains

the head **hd**, thus can we do a case analysis on this value, unfold the definitions and close the goal with the **eq_comm** lemma. For the inductive step, we generalized our induction hypothesis because it was too restrictive. The inductive step is proven by considering the values of the head **hd** and the tail of the head **tl_hd**. With some simplification tactics and the finishing tactic **tautology**, we can show that our goal is the same as our inductive hypothesis, completing the proof.

```
theorem XOR_correct : ∀ (A : list bool), ∀ (OUT : bool),
  XOR A = OUT ⟺ XOR_spec A OUT:=
  begin
    intros A OUT,
    destruct A,
    {
      intros h,
      unfold XOR XOR_spec,
      rw h,
      unfold count_tt XOR,
      simp,
      exact eq_comm,
    },
    {
      intros,
      induction tl generalizing A hd,
      {
        rw a,
        cases hd;
        {
          unfold XOR XOR_spec count_tt,
          simp,
          rw eq_comm,
        }
      },
      {
        rw a,
        unfold XOR count_tt,
        cases hd; cases tl_hd;
        {
          simp at *,
          unfold XOR_spec count_tt at *,
          simp at *,
          try {ring_nf, simp},
          tautology,
        }
      }
    }
  end
```

Figure 22: Theorem `XOR` complies with `XOR_spec`

3.2 Multiplexer (2-to-1)

A multiplexer is an essential digital circuit used to switch between different inputs. In this section we will specify, implement and verify a 2-to-1 multiplexer. A 2-to-1 multiplexer is a circuit that has 2 input lines, a select line and an output line. One possible logical implementation is presented in Figure 23.

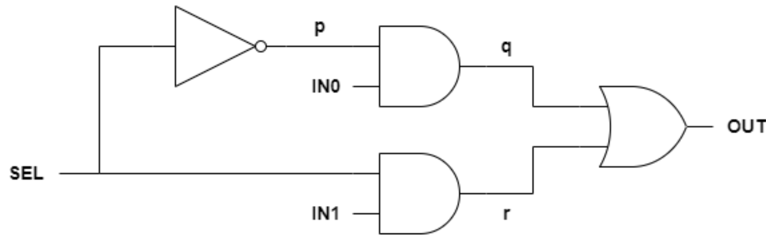

Figure 23: A logical implementation of a 2-to-1 multiplexer

3.2.1 Specification

The 2-to-1 multiplexer can be precisely defined by specifying the expected output for each value of the select line. In this case, when the select line `SEL` is `false`, we expect the output to be equal to the input signal `IN0`. Conversely, when the select line `SEL` is equal to `true`, we expect the output to be equal to the input signal `IN1`. The specification is shown in Figure 24.

```
def mux_spec {α : Type} (IN0 IN1 : α) (SEL : bool) (OUT : α) : Prop :=
        if SEL then (OUT = IN1) else (OUT = IN0)
```

Figure 24: Specification of a 2-to-1 multiplexer

The type of inputs `IN0`, `IN1` and output `OUT` is generic, and is represented by the type variable α. This allows for a flexible specification that can accommodate various data types. We will utilize this specification to verify a 2-to-1 multiplexer with single bit inputs and n-bit inputs.

3.2.2 Specification Functionality

The proof in Figure 25 demonstrates that for all inputs `IN0`, `IN1` and `SEL`, the specification produces a unique output `OUT`. We start by introducing the variables `IN0`, `IN1` and `SEL`. Then we apply the `exists_unique_of_exists_of_unique` lemma which results in the proof obligations of the existence and uniqueness of an output. For the existence part, we show that there exists an output by considering each possible value of SEL and closing the goals using the `exists_eq` lemma. Finally for the uniqueness part, we show that if $\texttt{y}_1$ and $\texttt{y}_2$ satisfy the specification, then they must be equal to each other. We do this by rewriting and simplifying our hypotheses, and showing that this holds for each possible value of `SEL`.

```
theorem mux_spec_unique : ∀ (IN0 IN1 SEL: bool),
  ∃! (OUT : bool), mux_spec IN0 IN1 SEL OUT :=
  begin
    intros IN0 IN1 SEL,
    apply exists_unique_of_exists_of_unique,
    { --existence of output
      cases SEL; exact exists_eq,
    },
    { --uniqueness of output
      intros y₁ y₂,
      cases SEL;
      {
        unfold mux_spec,
        simp,
        intros h₁ h₂,
        subst h₂,
        exact h₁,
      }
    }
  end
```

Figure 25: Theorem exists unique of `MUX_spec`

3.2.3 Implementation (1-bit inputs)

Similar to the approach used in the [SH17] paper, we assign variable names to each output of the logic gates, as illustrated in Figure 23. Subsequently, we define a function in LEAN3 that takes three inputs, namely `IN0`, `IN1` and `SEL` each of type `bool`, and produces an output of type `bool`. The reason for choosing type `bool` is that we only have 1-bit inputs.
Inside of this function we define the variables `p`, `q` and `r` and return the value of `(OR [q, r])` which is the gate that represents the output. The implementation is shown in Figure 26.

```
def mux_imp (IN0 IN1 SEL : bool) : bool :=
  let p := NOT sel,
      q := AND [IN1, p],
      r := AND [IN2, SEL] in
  (OR [q, r])
```

Figure 26: Implementation of a 2-to-1 Multiplexer

3.2.4 Implementation (n-bit inputs)

To create a 2-to-1 multiplexer using n-bit wide inputs, we can make use of our implementation of a 2-to-1 multiplexer for 1-bit inputs, assuming that it has been correctly implemented. Consider two input arrays `IN0` and `IN1` with elements `[1:N]`. We connect each pair `(IN0[k], IN1[k])` with $0 < k \leq N$, to a 2-to-1 multiplexer and then share a single select line with these multiplexers.

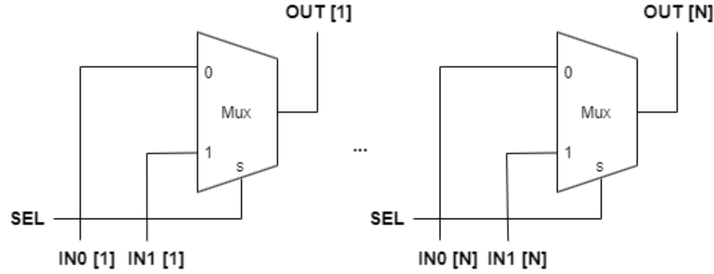

Figure 27: A description of a 2-to-1 n-bit multiplexer

First we need to define a zip function which merges two arrays, `A` of type α and `B` of type β, into a single array of type $(\alpha \times \beta)$ where the elements are pairs `(a × b)` with `a` $\in$ `A` and `b` $\in$ `B`.

```
def zip_array {n : ℕ} {α β : Type} (a : array n α) (b : array n β) : array n (α × β) :=
  ⟨λ i, (a.read i, b.read i)⟩
```

Figure 28: Zip function

Next we apply the zip function on our input arrays `IN0` and `IN1`. This will create a new array where each element is a pair of elements from `IN0` and `IN1`. Finally we can use the map function to apply the `mux_imp` which we have previously defined, on each element of this new array. The final result is a n-bit 2-to-1 multiplexer, shown in Figure 29.

```
def mux_n_imp {n : ℕ} (IN0 IN1 : array n bool) (SEL : bool) : array n bool :=
  (zip_array IN0 IN1).map (λ x, mux_imp x.fst x.snd sel)
```

Figure 29: Implementation of a n-bit 2-to-1 Multiplexer

3.2.5 Verification

The theorem `mux_complies_to_spec` in Figure 30 shows the functional correctness of the 2-to-1 multiplexer implementation. This theorem states that for all given input values `IN0`, `IN1` and `SEL`, and all values out the output `OUT`, the `mux_spec` holds if and only if the implementation `mux_imp` produces the same output.
We start the proof by introducing the variables the variables `IN0`, `IN1` and `SEL`. Then we do a case analysis on the input value `SEL`, to see if we get the expected output. For each case, we unfold the definitions, simplify them and then solve both the goals using the commutativity property of equality.

```
theorem mux_complies_to_spec : ∀ (IN0 IN1 SEL: bool),
  ∀ (OUT : bool), mux_spec IN0 IN1 SEL OUT ⟺ (mux_imp IN0 IN1 SEL) = OUT :=
  begin
    intros IN0 IN1 SEL out,
    cases SEL;
    {
      unfold mux_spec mux_imp,
      simp,
      unfold AND OR NOT,
      simp,
      rw eq_comm,
    },
  end
```

Figure 30: Theorem `mux_imp` complies with `mux_spec`

The theorem `n_mux_complies_to_spec` in Figure 31 shows the functional correctness of the n-bit 2-to-1 multiplexer implementation. This theorem states that for all given input values `IN0`, `IN1` and `SEL`, and all values out the output `OUT`, the `mux_spec` holds if and only if the implementation `n_mux_imp` produces the same output.
We start the proof by introducing the variables `IN0`, `IN1` and `SEL` into the context. Then we do a case analysis on the input value `SEL`, to see if we get the expected output. We then split the goal into a left and right implication. For both directions we simplify and unfold the definitions. Next we introduce a hypothesis h_1 which assumes the correct value of the output value `OUT`. Then by applying the `array.ext` theorem, we can introduce a new variable which represents the index of the array. Finally we can rewrite our hypothesis h_1 into our main goal and solve all goals by applying the `array.read_map` theorem.

```
theorem n_mux_complies_to_spec {n : ℕ} : ∀ (IN0 IN1 : array n bool) (SEL : bool),
        ∀ (OUT : array n bool), mux_spec IN0 IN1 SEL OUT ⟺ (mux_n_imp IN0 IN1 SEL) = OUT :=
  begin
    intros IN0 IN1 SEL OUT,
    cases SEL;
    {
      split;
      {
        unfold mux_n_imp mux_spec mux_imp zip_array,
        simp,
        unfold OR AND NOT,
        simp,
        intros h₁,
        apply array.ext,
        intros i,
        rw ←h₁,
        apply array.read_map,
      },
    },
  end
```

Figure 31: Theorem `n_mux_imp` complies with `mux_spec`

3.3 Full Adder

A full adder is a digital circuit that performs addition on three binary inputs: A, B and a carry input CIN. It produces a sum SUM and a carry output COUT. The logical implementation of a full binary adder (single bit inputs and outputs) is shown in Figure 32. In this section we will specify a full adder and a full binary adder, implement them and verify the implementation of the full binary adder with the specification.

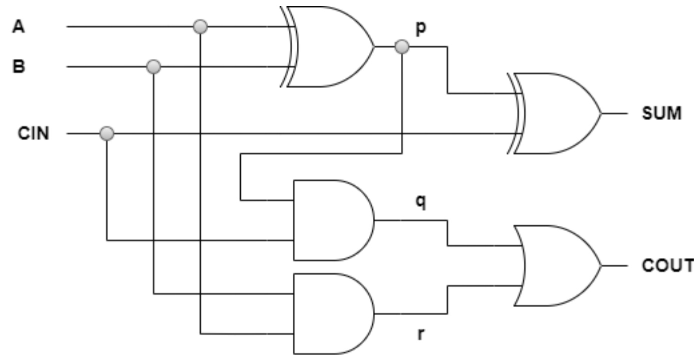

Figure 32: A logical implementation of a full binary adder

3.3.1 Specification

To specify a full binary adder, we can mathematically express the values of SUM and COUT. The sum is calculated by adding the boolean values together and using modulo 2 operation to ensure the value stays within the range of a boolean value. The COUT is assigned the value true if the sum of the boolean values is equal or greater than 2, otherwise it is assigned to false. The specification is shown in Figure 33.

```
def full_adder_spec (A B CIN SUM COUT : bool) : Prop :=
        SUM = nat_to_bool ((bool_to_nat A + bool_to_nat B + bool_to_nat CIN) % 2) ∧
        COUT = ((bool_to_nat A + bool_to_nat B + bool_to_nat CIN) ≥ 2)
```

Figure 33: Specification of a full binary adder

To specify a full adder, we can also mathematically express the values SUM and COUT as shown previously. However, the types of our inputs A and B are now `array n bool`. This means that the sum, when converted from the base-2 representation to a natural number must equal the sum of all inputs modulo 2^n, with `n` the size of the array. Conversely, the COUT is calculated by adding all inputs together and assigning the value of true if the sum of the inputs is equal or greater than 2^n, otherwise it is assigned to false. The specification is shown in Figure 34.

```
def full_n_adder_spec {n : ℕ} (A B : array n bool) (Cin : bool)
    (Sum : array n bool) (Cout : bool) : Prop :=
        bool_arr_to_nat Sum = (bool_arr_to_nat A + bool_arr_to_nat B + bool_to_nat Cin) % 2^n ∧
        Cout = ((bool_arr_to_nat A + bool_arr_to_nat B + bool_to_nat Cin) ≥ 2^n)
```

Figure 34: Specification of a full adder

3.3.2 Specification Functionality

The proof in Figure 35 demonstrates that for all inputs `A`, `B` and `CIN`, the specification produces a unique output pair `(SUM, COUT)`. For this, we proved both the existence and uniqueness of the output pair.

```
theorem full_adder_unique : ∀ (A B CIN : bool), ∃!(SUM COUT : bool),
  full_adder_spec A B CIN SUM COUT :=
  begin
    intros A B CIN,
    apply exists_unique_of_exists_of_unique,
    {
      unfold full_adder_spec,
      cases A; cases B; cases CIN;
      {
        unfold bool_to_nat nat_to_bool,
        simp,
        unfold nat_to_bool,
        simp,
      }
    },
    {
      intros y₁ y₂,
      unfold full_adder_spec,
      intros h₁ h₂,
      destruct h₁,
      destruct h₂,
      simp,
      finish,
    }
  end
```

Figure 35: Theorem exists unique of `full_adder_spec`

The existence of the output pair is proven by showing that for every possible combination of inputs `(A, B, CIN)`, there exists at least one output pair `(SUM, COUT)`, that satisfies the `full_adder_spec`. This is done by unfolding the `full_adder_spec` definition and evaluating all possible cases for the inputs. The uniqueness of the output pair is proven by showing that, for all output pairs $(y_1, \texttt{COUT})$, and $(y_2, \texttt{COUT})$, if both satisfy the `full_adder_spec` for the same inputs, and for each of these pairs there exists exactly one `COUT`, then y_1 must equal y_2. We do this by introducing the variables y_1 and y_2, unfolding our specification and introducing the hypotheses that y_1 and y_2 satisfy the specification. Then by destructing these hypotheses and simplifying, we can use the finish tactic to automatically find a proof to close the goal.

```
theorem full_n_adder_unique {n : ℕ} : ∀ (A B : array n bool) (CIN : bool),
    ∃!(SUM : array n bool)(COUT : bool), full_n_adder_spec A B CIN SUM COUT :=
  sorry,
```

Figure 36: Theorem exists unique of `full_n_adder_spec`

The proof is Figure 36 is marked with the placeholder `sorry`. This means that we were not able to provide a proof for the uniqueness of the output pair for arrays of length n. Similarly, we were unable to proof that our implementation in Section 3.3.4 complies with the specification given in Figure 34. While we believe that both the specification and implementation are correct, due to limitations in time and knowledge, we were unable to complete the proofs.

3.3.3 Implementation full binary adder

To implement the full binary adder, we will use the same approach used in Section 3.2.3. We define a function that takes three inputs, namely `A`, `B` and `C` each of type `bool`, and produces an output of type `(bool × bool)`. The first member of the pair represents the sum and the second member represents the carry-out. We will again define variables representing the output of the gates in Figure 32. Finally we will return the output gates `(XOR [p, CIN], OR [q, r])`. The implementation is shown in Figure 37.

```
def full_adder_imp (A B CIN : bool) : (bool × bool) :=
        let p := XOR [A, B],
                        q := AND [CIN, p],
                        r := AND [A, B] in
        (XOR [p, CIN], OR [q, r])
```

Figure 37: Implementation of a full binary adder

3.3.4 Implementation full adder

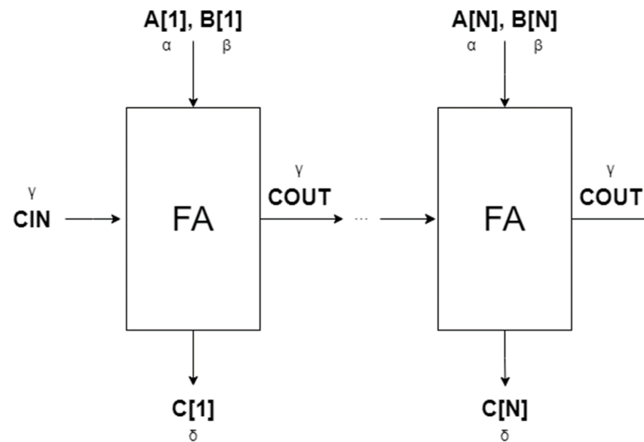

Figure 38: A description of a full n-bit adder

In Figure 38 we illustrate how we will implement a full n-bit adder. For all inputs and outputs, the corresponding types are shown. For every iteration, we generate an output bit in the array `C` which contains the sum of the bits at that iteration from arrays `A` and `B`. We also produce a `COUT` as input for the next iteration, or as output for the last iteration. After the last iteration, we will have an output array `C` containing the binary sum of the arrays `A` and `B`, and we will also have a carry-out `COUT`. This process is called 'folding'.

```
def full_n_adder_imp {n : ℕ} (A B : array n bool) (CIN : bool) : (array n bool × bool) :=
                fold_array (full_adder_imp) (zip_array A B) (mk_array n ff, CIN)

def fold_array {n : ℕ} {α β γ δ : Type} (f : α → β → γ → (δ × γ)) (a : array n (α × β))
(i : array n δ × γ) : (array n δ × γ) :=
  fold_array_aux f a n (le_refl _) i

def fold_array_aux {n : ℕ} {α β γ δ: Type} (f : α → β → γ → (δ × γ)) (a : array n (α × β)) :
                    Π (i : ℕ), i ≤ n → (array n δ × γ) → (array n δ × γ)
    | (nat.zero)      h acc := acc
    | (nat.succ j) h acc :=
      let i : fin n := ⟨j, h⟩,
        (SUM, COUT) := f (a.read i).fst (a.read i).snd acc.snd in
        fold_array_aux j (le_of_lt h) (acc.fst.write i SUM, COUT)
```

Figure 39: Implementation of a full adder

The `fold_array` function in Figure 39 takes a folding function, an input array, an initial value for the accumulator which stores the partial calculated sum and carry-out, and produces a pair where the first element is the sum and the second element is the carry-out. This function in turn calls the `fold_array_aux` function with the needed arguments which applies the actual folding logic. The `full_n_adder_imp` function is the implementation of the adder. We call the `fold_array` function with the `full_adder_imp` as our folding function, `(zip_array A B)` being our input arrays combined into a single array and `(mk_array n ff, CIN)` being the initial value of our accumulator. This results in an implementation of a full n-bit adder.

3.3.5 Verification

The theorem in Figure 40 shows that our full binary adder complies with the given specification. This is done by unfolding both the implementation and specification. We then check that the implementation and specification have the same output pair for all possible inputs. During this case analysis, we unfold as much as we can, and solve the goal automatically using the tautology tactic. As discussed in Section 3.3.2, we believe that the specification and implementation are correct, but due to limitations in time and knowledge, we were unable to finish the proofs. The unfinished proof is shown in Figure 41.

```
theorem full_adder_correct : ∀ (A B CIN : bool), ∀ (SUM COUT : bool),
  full_adder_spec A B CIN SUM COUT ⟺ full_adder_imp A B CIN = (SUM, COUT) :=
  begin
    intros,
    unfold full_adder_imp full_adder_spec,
    simp,
    cases A; cases B; cases CIN;
    {
      unfold bool_to_nat AND OR XOR,
      simp,
      unfold nat_to_bool,
      tautology,
    }
  end
```

Figure 40: Theorem `full_adder_imp` complies with `full_adder_spec`

```
theorem full_n_adder_correct {n : ℕ} : ∀ (A B : array n bool) (CIN : bool),
  ∀ (SUM : array n bool) (COUT : bool),
  full_n_adder_spec A B CIN SUM COUT ⟺ full_n_adder_imp A B CIN = (SUM, COUT) :=
  begin
    sorry,
  end
```

Figure 41: Theorem `full_n_adder_imp` complies with `full_n_adder_spec`

3.4 4-bit Shifter

A shifter circuit is a combinational circuit that shifts a data word to the left or to the right. Shifters are often used to implement multiplication and division by powers of two. In this section we will implement and verify a 4-bit shifter that is able to perform a logical left shift and logical right shift. A logical left shift moves all bits one position to the left, with the right-most bit set to 0. Similarly, a logical right shift moves moves all bits one position to the right, with the left-most bit set to 0.

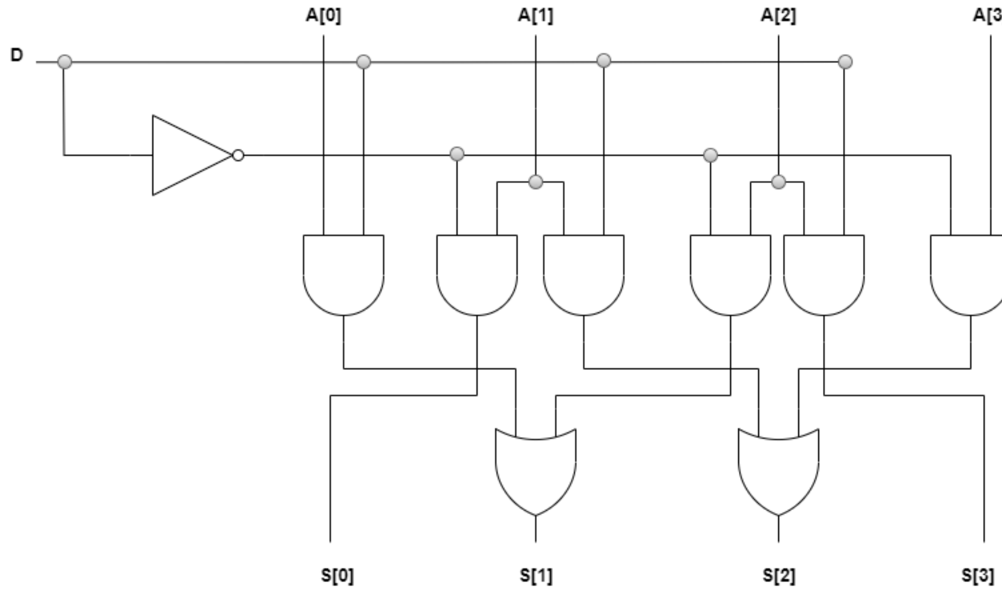

Figure 42: A logical implementation of a 4-bit shifter

The circuit shown in Figure 42 represents the logic gates circuit of a 4-bit shifter, which is a simplified version of the 8-bit shifter circuit presented in [Com21]. This simplification is to keep the theorems concise but they can be applied to any n-bit shifter circuit using the same pattern.

3.4.1 Specification

The specification of our 4-bit shifter is shown in Figure 43. This specification is generalized for n-bit arrays. The behavior of our shifter is determined by the input `D`. In the case that `D` is true, the shifter is expected to perform a logical right shift. The function uses the read function on array `A` to obtain the value at index `(i.val - 1)`. If this value falls within the bounds of the array, then the output array will receive this value at index `(i.val)`, else it is given the value false. In contrast, when `D` is false, the shifter is expected to perform a logical left shift. Again the function uses the read function on array `A` to obtain the value at index `(i.val + 1)`. If this value is within the bounds of the array, then the output array will receive this value at index `(i.val)`. Otherwise it will receive the value false.

```
def shft_spec {n : ℕ} (A : array n bool) (D : bool) (OUT : array n bool) : Prop :=
  if D = tt then OUT = ⟨λ i, if h : i.val > 0 ∧ i.val - 1 < n then A.read ⟨i.val - 1, h.right⟩
                             else ff⟩
            else OUT = ⟨λ i, if h : i.val + 1 < n then A.read ⟨i.val + 1, h⟩ else ff⟩
```

Figure 43: Specification of a n-bit shifter

3.4.2 Specification Functionality

The proof in Figure 44 demonstrates that for all input arrays `A` and all inputs `D`, the specification produces a unique output. We start by introducing the variables `A` and `D`. Then we apply the `exists_unique_of_exists_of_unique` lemma which has us prove the existence and uniqueness of an output. For the existence, we simply unfold the specification and show that for each value of `D`, there exists an output with the `exists_eq` lemma. Finally for the uniqueness part, we show that if $\texttt{y}_1$ and $\texttt{y}_2$ satisfy the specification, then they must be equal to each other. We do this by rewriting and simplifying our hypotheses, and showing that this holds for each possible value of `D`.

```
shft_unique {n : ℕ} : ∀ (A : array n bool) (D : bool),
        ∃! (OUT : array n bool), shft_spec A D OUT :=
  begin
    intros A D,
    apply exists_unique_of_exists_of_unique,
    {
      unfold shft_spec,
      cases D;
      {
        exact exists_eq,
      }
    },
    {
      intros y₁ y₂,
      unfold shft_spec,
      intros h₁ h₂,
      cases D;
      {
        simp at h₁ h₂,
        rewrite ←h₂ at h₁,
        exact h₁,
      }
    }
  end
```

Figure 44: Theorem exists unique of `shft_spec`

3.4.3 Implementation

To implement the shifter circuit, we define the outputs of each logic gate in the design presented in Figure 42. After that we create a list where each index represents the correct output gate. Finally we convert this list to an array as shown in Figure 45.

```
def shft_imp (A : array 4 bool) (D : bool) : array 4 bool :=
  let D':= NOT D,
      p := AND [A.read 0, D],
      q := AND [D', A.read 1],
      r := AND [D, A.read 1],
      s := AND [D', A.read 2],
      t := AND [D, A.read 2],
      u := AND [D', A.read 3] in
  [q, OR[p, s], OR[r, u], t].to_array
```

Figure 45: Implementation of a 4-bit shifter

3.4.4 Verification

The theorem in Figure 46 shows that for all inputs `A` and `D`, the shifter implementation satisfies the shifter specification. This is done by introducing the variables `A` and `D`, and unfolding the specification. Then, the two cases of the input `D` are considered, which lets us prove both the left shift and right shift. We can apply the same proof to each shift direction. Subsequently, we show that for every index within the array bounds, the value computed by the implementation matches the value specified by the specification. This is accomplished by using the reflexivity tactic. When we reach an index outside of the array bounds, our hypothesis, which states that the array index must be smaller than 4, will be false. Therefore by the principle of explosion we can complete the proof.

```
theorem shft_correct : ∀ (A : array 4 bool) (D : bool),
  shft_spec A D (shft_imp A D) :=
  begin
    intros A D,
    unfold shft_spec,
    simp,
    cases D;
    {
      simp,
      unfold shft_imp,
      apply array.ext,
      intro i,
      ring_nf,
      unfold list.to_array,
      dsimp at *,
      unfold AND NOT OR,
      simp,
      cases i,
      cases i_val with i_val1,
      {
        refl,
      },
      {
        cases i_val1 with i_val2,
        {
          refl,
        },
        {
          cases i_val2 with i_val3,
          {
            refl,
          },
          {
            cases i_val3 with i_val4,
            {
              refl,
            },
            {
              exfalso,
              repeat { rw nat.succ_eq_add_one at i_property },
              simp at i_property,
              exact i_property,
            }
          }
        }
      }
    }
  end
```

Figure 46: Theorem `shft_imp` complies with `shft_spec`

4 Sequential Circuits

Sequential circuits are circuits that do not only dependent on the current inputs, but also depend on previous inputs and ouputs, or memory, or both.
In this section we will take a look at signals which model data lines over time (Section 4.1), a memory component (Section 4.2), and a program counter (Section 4.3).

4.1 Signals

Sequential circuits operate based on a global clock. To be able to represent inputs and outputs over this global clock (time), we defined functions that map a natural number (time) to an output value such as a `bool`, `array n bool` and type α where α can represent any type. This methodology is similar to the approach used in [Lee92] where they defined functions that map natural numbers to booleans and vectors of booleans. The function definitions are shown in Figure 47.

```
def stream (α : Type) := ℕ → α

def signal := stream bool

def sig_n (n : ℕ) := stream (array n bool)
```

Figure 47: Implementations of various signals

4.2 Memory

A memory component is able to store single or multiple bits of data. Many implementations of memory, with an emphasis on flip-flops have a so called metastable state. In this state, the output of the flip-flop hovers between 1 and 0 for an indeterminate time [CM73]. To capture this behavior, we will make use of an under-specification for a memory component. Consequently, we will not be able to prove that our specification, given all inputs, has a unique output. What we can proof is the following: given two input streams, if the specification is related to two output streams that initially have an undetermined value, once the value of both output streams at a specific time are determined and are equal to each other, then for any time thereafter, these streams will have the same value. We call this 'almost equal'. The concept of streams being 'almost equal' can be compared to circuits having a metastable state. Once the output of a circuit is stable, then all future outputs will be the same.

4.2.1 Specification

T	D	S	M	O
t	*	0	d	d
t+1			d	d
...				
t'	d	1	*	*
t'+1	*	*	d	d
...				

Figure 48: Signal table of a memory component

In Figure 48 we have illustrated a table which shows the relationship between the time `T` and the signals `D` (data input signal), `S` (set input signal), `M` (memory value signal) and `O` (output signal). The behavior of the memory component is dependent on the value of S_t, in other words, the value of `S` at time `t`. If S_t is 0, then the memory component retains its current memory value into time `t+1` ($M_{t+1} = M_t$). Otherwise, if S_t is 1, the memory component updates its value at `t+1` with the value at D_t ($M_{t+1} = D_t$). The output signal `O` is always equal to the memory value signal `M`, thus can we quantify the `M` signal away.
We have marked these relations as so-called 'frames' in Figure 48. These frames relate the values of the signals to each other. The frames that are marked with the letter `C` can be interpreted as the signals `M` and `O` always being equal to each other. We can interpret the frame marked with the letter `B` as follows, if at time `t'` the signal `S` is 1, then the signal `M` at time `t'+1` must be the same value as signal `D` at time `t'`. Finally the frame marked with the letter `A` can be read as, if at time `t` the signal `S` is 0, then the signal `M` at time `t+1` receives the value of signal `M` at time `t`.

```
def mem_spec {α : Type} (D : stream α) (S : signal) (M : stream α) : Prop :=
  ∀ t : ℕ,
    (S t = tt → M (t+1) = D t) ∧  --frame 1 D S M+1
    (S t = ff → M (t+1) = M t)    --frame 2 S M M+1
```

Figure 49: Specification of a memory component

The specification shown in Figure 49 describes the frames as discussed above. We have chosen an arbitrary type α for the signals `D` and `M`, because we will use this specification for both signals of type `bool` and signals of type `array n bool`.

4.2.2 Specification Functionality

```
def almost_eq {α : Type} (N M : stream α) : Prop :=
  ∀ (t : ℕ),
    (N t = M t) → ∀ (t' : ℕ), (t' > t) → N t' = M t'

def almost_eq' {α : Type} (N M : stream α) : Prop :=
  ∀ (t : ℕ),
    (N t = M t) → N (t+1) = M (t+1)
```

Figure 50: 'Almost equal' predicates

The two functions in Figure 50 say that at any time `t`, if two streams have the same output, then for all time after `t`, they must still produce the same output. The function `almost_eq'` is a simpler formulation of the same property, and a proof of their equivalence is shown in Figures 52 and 51. The simpler property will be used in the almost uniqueness proof.

```
lemma help {t' t n: nat} (h: t' > t) (hh: t' - t = n): t' = t + n :=
  begin
    rw ←hh,
    rw nat.add_sub_of_le,
    exact le_of_lt h,
  end
```

Figure 51: Help lemma for `almost_eq_equiv`

```
lemma almost_eq_equiv {α : Type} (N M : stream α) : almost_eq N M ⟺ almost_eq' N M :=
  begin
    unfold almost_eq almost_eq',
    split,
    {
      intros h t,
      specialize h t,
      intros h₁,
      apply h,
      {
        exact h₁,
      },
      {
        simp,
      }
    },
    {
       intros h t h₁ t' h₂,
       generalize hhh: t' - t = dt, -- We have t' > t, so take the difference (dt)
       destruct dt; intros; rw a at hhh,
       { -- Suppose dt = 0, then we can close the goal
         rw (help h₂ hhh),
         simp, tauto,
       },
       { -- Suppose dt > 0, then do a strong induction
         revert t',
         apply (nat.strong_induction_on n); intros n_1 IH; intros,
         destruct n_1; intros,
         { -- Suppose dt = 1
           rw a_1 at hhh,
           rw (help h₂ hhh),
           apply h, tauto,
         },
         { -- Otherwise
           rw (help h₂ hhh),
           apply h, simp,
           apply (IH n_2),
           {
            rw a_1,
            exact nat.lt_succ_self _,
           },
           repeat{
            rw a_1,
            simp,
           },
         }
       }
     }
  end
```

Figure 52: Lemma almost_eq_equiv

The theorem in Figure 53 shows that two output streams that satisfy the memory specification must be almost equal to each other. We prove this by introducing our variables and hypotheses and rewriting our simplified definition of `almost_eq` using the `almost_eq_equiv` lemma. Next we unfold the functions and apply the specialize tactic with respect to `t` on h_1 and h_2. Then we simplify our hypotheses and split them into smaller parts. Finally, we consider each possible value of S_t and finish the proof by using simplification and rewrite tactics, till we can show that our goal is the same as one of our assumptions.

```
theorem mem_spec_almost_unique {α : Type} : ∀ (D : stream α) (S : signal) (M : stream α),
  mem_spec D S M → ∀(N : stream α), mem_spec D S N → almost_eq N M :=
  begin
    intros D S M h₁ N h₂,
    rw almost_eq_equiv,
    unfold almost_eq',
    intros t heq,
    unfold mem_spec at h₁ h₂,
    specialize h₁ t,
    specialize h₂ t,
    destruct h₁,
    intros h₁l h₁r,
    destruct h₂,
    intros h₂l h₂r,
    cases S t,
    {
      simp at h₂r,
      rw h₂r,
      simp at h₁r,
      rw h₁r,
      exact heq,
    },
    {
      simp at h₂l,
      rw h₂l,
      simp at h₁l,
      rw h₁l,
    }
```

Figure 53: Theorem `mem_spec` almost unique

4.2.3 Implementation

To implement a memory, we define a recursive function. This function recurses backwards and checks the value of the `S` signal. If `S` has the value of 1, then it will simply return the value of `D` at that specific time. If `S` has the value of 0, the function will recurse back and check the value of the memory implementation at the previous time point. The base case is the initial value of the memory module. The implementation of this memory module can be found in Figure 54.

```
def mem_imp {α : Type} (I : α) (D : stream α) (S : signal) : stream α
  | nat.zero := I
  | (nat.succ y) := if S y = tt then D y
                    else mem_imp y
```

Figure 54: Implementation of a memory component

4.2.4 Verification

```
theorem mem_correct {α : Type} : ∀ (D : stream α) (S: signal), ∀ (I : α),
  mem_spec D S (mem_imp (I) D S) :=
  begin
    unfold mem_spec,
    intros,
    split,
    {
      intros h,
      unfold mem_imp,
      rw if_pos,
      exact h,
    },
    {
      intros h,
      unfold mem_imp,
      rw if_neg,
      simp,
      exact h,
    }
  end
```

Figure 55: Theorem `mem_imp` complies with `mem_spec`

The proof in Figure 55 shows that the `mem_imp` complies with the `mem_spec`. We start by introducing the variables. Then we unfold the specification, which allows us to introduce the time variable `t`. Finally we proof each frame using the split tactic. The proof can be finished by the fact that our hypothesis of $\mathtt{S}_\mathtt{t}$ is the same as the if-statement condition, which results in an equality.

4.3 Program Counter

A program counter is a component within digital systems that holds the address of the next instruction to be executed. This address is incremented by one at every clock cycle. The program counter can also jump to a specific address, but for simplicity we have not added this functionality in our program counter. The program counter we will look at, is able to increment at every clock cycle and can also be reset based on a reset signal `R`.

4.3.1 Specification

The behavior of the program counter is dependent on the value of the reset signal `R` at time `t`. If R_t is 0, the program counter will reset the value it contains to an array of `ff`, and reflect this change in the output signal `O` at time `t+1`. This essentially sets the output of the program counter to 0. Otherwise, if R_t is 1, the program counter increments the value it contained at O_t, and returns the incremented value at O_{t+1}. The specification is given in Figure 56.

```
def pc_spec {n : ℕ} (R : signal) (O : sig_n n) : Prop :=
  ∀ t : ℕ,
    (R t = tt → O (nat.succ t) = mk_array n ff) ∧
    (R t = ff → O (nat.succ t) = (full_n_adder_imp (O t) (mk_array n ff) tt).fst)
```

Figure 56: Specification of a n-bit program counter

One thing to note about the above specification is that we are using the `full_n_adder_imp` component in our specification, because we have already introduced this component earlier. This component could be changed to a function that simply increments the value of a boolean array by 1.

4.3.2 Implementation

Because we are working with signals, we can not easily implement our program counter using two separate components (`full adder` and `memory`). This is because of the way we implemented the signals. LEAN3 can not easily proof the termination of such function compositions, thus the values of the entire stream (a signal) must be known a priori. A solution to this problem is to create a single component, similar to the one made for the memory which then has an adder integrated in it.

```
def pc_imp {n : ℕ} (I : array n bool) (R : signal) : sig_n n
  | nat.zero := I
  | (nat.succ y) := if R y = tt then mk_array n ff
                    else (full_n_adder_imp (pc_imp y) (mk_array n ff) (tt)).fst
```

Figure 57: Implementation of a n-bit program counter

As shown in Figure 57, we recursively call the adder function on the previous value of the program counter. This increments the counter each time step, unless the reset signal is set, which causes it to be set to an array of `ff`, essentially resetting the program counter to 0. This is all under the assumption that the `full_n_adder_imp` is correctly implemented.

4.3.3 Verification

```
theorem pc_correct {n : ℕ} : ∀ (R : signal) (I : array n bool),
  pc_spec R (pc_imp (I) R) :=
  begin
    intros,
    unfold pc_spec,
    intros t,
    split,
    {
      intros h₁,
      unfold pc_imp,
      rw if_pos,
      exact h₁,
    },
    {
      intros h₁,
      unfold pc_imp,
      rw if_neg,
      simp,
      exact h₁,
    }
  end
```

Figure 58: Theorem `pc_imp` complies with `pc_spec`

The proof in Figure 58 is analogue to the one for the Memory component in Figure 55. We introduce the variables `R` and `I`, unfold the specification and introduce the variable `t`. Then we split our specification into the two separate frames to prove. For each of these frames, the proof can be finished by the fact that our hypothesis of R_t is the same as the if-statement condition, which results in an equality.

5 Case Study: Verification of a FSM

In this section we will take a look at a design of a binary sequence recognizer by Alberto I. Leibovich and Pablo E. Leibovich [LL16]. This binary sequence recognizer is able to recognize the binary pattern '101' followed by any number of consecutive '1's in a bit stream. We will limit us to only the recognition of the binary pattern '101'. In their article, the authors give truth tables of logic components and a schematic of the circuit in the GreenPAK software. Our objective is to take these truth tables and turn them into specifications in LEAN3, create a specification for the expected behavior of the whole circuit and then implement these specifications. We will then prove that our implementations comply with these specifications.

5.1 FSM Schematic / Design

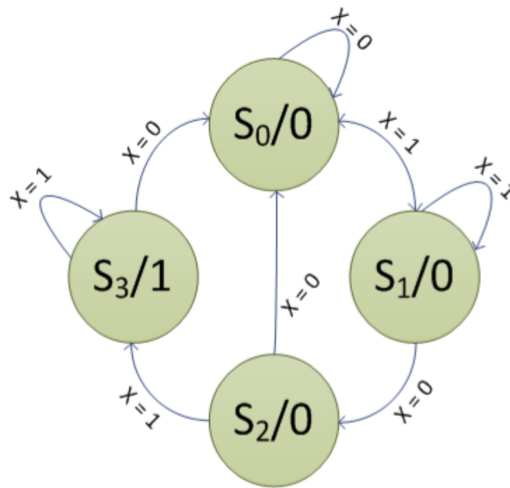

Figure 59: State diagram with initial state S_0, sourced from [LL16]

In the state diagram of the circuit, shown in Figure 59, there are a couple things to notice. First of all, there was an error in the diagram. The bi-directional transition $S_0 \longleftrightarrow S_1$ under input $\texttt{X} = 1$ should be a single transition of $S_0 \rightarrow S_1$. The transitions from state S_3 under input $\texttt{X} = 1$ will cause the state-machine to stay in an accepting state and under input $\texttt{X} = 1$, the state machine goes to the initial state S_0. Because we are only interesting in the accepting behavior of the binary sequence '101', we will not add these transitions from state S_3 in our specification.
In Figure 60 the block diagram of the sequence recognizer is shown. We will model the `CLK` with our signal inputs and outputs. Furthermore, we will refer to the '2-L0' AND-gate as `LUT0`. The in-and-outputs `Q0` and `Q1` refer to the outputs of the two flip-flops being used. The input `X` is the input stream where we will recognize the sequence '101' and the output `Z` is the output of the circuit. The truth-tables of the LUT units are shown in Figure 61.

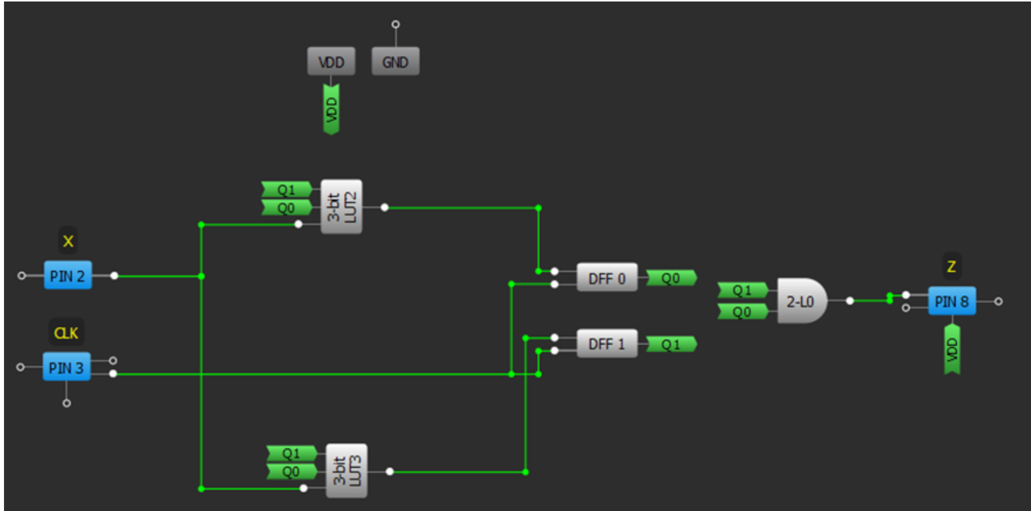

Figure 60: Flip-Flop based block diagram, sourced from [LL16]

Q1	Q0	X	OUT
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(a) LUT2

Q1	Q0	X	OUT
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

(b) LUT3

Q1	Q0	OUT
0	0	0
0	1	0
1	0	0
1	1	1

(c) LUT0

Figure 61: Truth tables of LUT units, sourced from [LL16]

5.2 Specification

We begin with making the specifications for the LUT units, which are essential parts in our final implementation to be able to recognize the binary sequence '101'. This is accomplished by looking at the truth tables in Figure 61, and identifying the rows where the combinations of inputs result in an output of 1. These combinations are captured using the 'if' condition in our specifications. The 'else' clause in our specification captures all combinations which do not meet the requirements of the output being 1. In other words, the cases where the output is 0. These specifications are shown in Figure 62. The uniqueness of the output for all input combinations for these specifications can be proven similarly to the circuits we have seen before, therefore these proofs are omitted.

```
def LUT2_spec (Q1 Q0 X OUT : bool) : Prop :=
  if (¬Q1 ∧ ¬Q0 ∧ X) ∨ (¬Q1 ∧ Q0 ∧ X) ∨ (Q1 ∧ ¬Q0 ∧ X) ∨ (Q1 ∧ Q0 ∧ X) then OUT = tt
  else OUT = ff

def LUT3_spec (Q1 Q0 X OUT : bool) : Prop :=
  if (¬Q1 ∧ Q0 ∧ ¬X) ∨ (Q1 ∧ ¬Q0 ∧ X) ∨ (Q1 ∧ Q0 ∧ X) then OUT = tt
  else OUT = ff

def LUT0_spec (Q1 Q0 OUT : bool) :=
  if Q1 ∧ Q0 then OUT = tt else OUT = ff
```

Figure 62: Specifications of LUT units

The behavior of the circuit can be described as follows: when the circuit encounters a 1 at time `t`, followed by a 0 at time `t+1` and then a 1 at time `t+2`, then the output at time `t+3` must be 1. In all other cases the output must be 0 at time `t+3`. This effectively only allows the pattern '101' to be accepted. In Figure 63, the first conjunct describes the sequence of '101', which results in an output of 1 at time `t+3`. The other conjuncts correspond to the invalid sequences and result in an output of 0 at time `t+3`. The last conjunct does not explicitly describe the values of X_{t+1} and X_{t+2}. These values will later be assumed when we create our proof.

```
def SeqRec_spec (X OUT :signal) : Prop :=
  ∀ t : ℕ,
    (X t = tt → X (t+1) = ff → X (t+2) = tt → OUT (t+3) = tt) ∧
    (X t = tt → X (t+1) = tt → X (t+2) = tt → OUT (t+3) = ff) ∧
    (X t = tt → X (t+1) = tt → X (t+2) = ff → OUT (t+3) = ff) ∧
    (X t = tt → X (t+1) = ff → X (t+2) = ff → OUT (t+3) = ff) ∧
    (X t = ff → OUT (t+3) = ff)
```

Figure 63: Specification of a Sequence Recognizer

5.3 Implementation

```
def LUT2 (Q1 Q0 X : bool) : bool :=
  OR [AND [Q1, Q0, X], AND [Q1, NOT Q0, X], AND[NOT Q1, Q0, X], AND[NOT Q1, NOT Q0, X]]

def LUT3 (Q1 Q0 X : bool) : bool :=
  OR [AND [NOT Q1, Q0, NOT X], AND [Q1, NOT Q0, X], AND [Q1, Q0, X]]

def LUT0 (Q1 Q0 : bool) : bool :=
  AND[Q1, Q0]
```

Figure 64: Implementation of LUT units

The implementations of the LUT units is similar to creating the specifications. We use a n-bit OR gate for all the possible entries of the truth tables in Figure 61, and for each of these entries, we connect the input combinations with a n-bit AND gate, negating the inputs with a NOT gate if necessary.

```
def SeqRecAux {n : ℕ} (X : signal) : Π (i : ℕ), i ≤ n → bool → bool → bool → ℕ → bool
  |(nat.zero) h Q1 Q0 O k :=
    if O = tt then tt
    else ff
  |(nat.succ t) h Q1 Q0 O k :=
    if O = tt then tt
    else if k < 3 then
      let Q1_next := LUT3 Q1 Q0 (X t),
          Q0_next := LUT2 Q1 Q0 (X t),
          O_next := LUT0 Q1_next Q0_next in
      SeqRecAux (t) (le_of_lt h) Q1_next Q0_next (O_next) (k+1)
    else ff

def SeqRec (X : signal) : signal :=
  λ n, SeqRecAux X (n) (le_refl _) ff ff ff 0
```

Figure 65: Implementation of a Sequence Recognizer

In LEAN3, recursive functions must be proven to terminate. We are making use of pattern matching for our sequence recognizer implementation, where in each recursive call a value decreases, which LEAN3 can recognize and automatically proof for us that it will terminate. We can use this method because, if we approach the pattern '101' from both sides, it will still result in the same pattern. This allows us to keep the current LUT unit implementations and specifications. If we want to recognize any other pattern which is not a palindrome, we would have to apply a different strategy. We can for example apply forwards recursion, but we would have to use `termination_by` to proof the termination of the function.
The definitions in Figure 65 implement a sequence recognizer for the pattern '101'. The `SeqRec` function calls the recursive auxiliary function `SeqRecAux` on the argument n of the output signal (`SeqRec(X) n`), with the recursion parameters being initialized to 0 and `ff`. For each iteration, we keep track of the flip-flop values `Q1` and `Q0`, the output value `O` and the number of steps we went backwards `k`. While the number of steps is smaller than 3, we will calculate the next values of the flip-flops and output. This simulates a cycle of the circuit diagram illustrated in Figure 60. If after 3 time steps the output is 1, then the output of this function is also 1, else it is 0.

5.4 Verification

The theorem in Figure 66 shows the correctness proof of our implementation for the given specification. We start by introducing the input stream `X`, unfolding the specification and introducing the time variable `t`. Next we use the repeat tactic to split the conjuncts and proof them using the same proof. For each conjunct we unfold the implementation as much as we can. At some point, we can not unfold it any further because we need to rewrite the assumptions of the values X_t, X_{t+1} and X_{t+2} in our goal. Then by using simplification tactics and considering the possible values of `t`, we can close the proof using the reflexivity tactic, or in some cases by simply unfolding. The last part of the proof is specifically for the last conjunct. This proof is the same but instead of being able to rewrite the hypotheses of the values X_{t+1} and X_{t+2}, we directly do a case analysis on all possible values.

```
theorem SeqReq_correct : ∀ (X : signal),
  SeqRec_spec X (SeqRec X) :=
  begin
    intros X,
    unfold SeqRec_spec,
    intros t,
    repeat{
      split,
      intros h₁ h₂ h₃,
      unfold SeqRec,
      repeat {
        unfold SeqRecAux,
        simp,
      },
      simp [LUT0, LUT2, LUT3, AND, OR, NOT],
      rw h₃,
      rw h₂,
      rw h₁,
      simp [NOT],
      ring_nf,
      simp,
      cases t;
      {
        unfold SeqRecAux,
        refl,
        try {unfold NOT},
      },
    },
    intro h₁,
    unfold SeqRec,
    repeat {
      unfold SeqRecAux,
      simp,
    },
    simp [LUT0, LUT2, LUT3, AND, OR, NOT],
    cases X (t+1); cases X(t+2);
    {
      simp [NOT],
      ring_nf,
      simp,
      cases t;
      {
        unfold SeqRecAux,
        finish,
      },
    },
  end
```

Figure 66: Theorem SeqRec complies with SeqRec_spec

6 Conclusions and Future Work

In this section we will conclude the thesis, discuss our research and look at possibilities for future work.

6.1 Conclusion

Let us recall the research question:

> *"How can we verify the correctness of a combinational and sequential circuit implementation, with respect to a given formal specification, using the LEAN3 interactive theorem prover?"*

Throughout this thesis, we have seen quite a few combinational and sequential circuits, with some circuits being a bit more complex than others. For nearly all circuits, with the exception of the n-bit full adder, we have successfully shown how we can specify the behavior, implement the circuit, and verify our implementation with our specification using the LEAN3 theorem prover.

LEAN3 has shown to be a very useful tool in helping us formalizing specifications, implementations and creating proofs. Many of the reasoning steps, such as term rewriting, could be partially automated and even some small parts of the proof could be fully automated. We were also able to use tactics that were automatically suggested, which helped us in finding the correct lemmas to apply. The mathlib library for LEAN3 contained a lot of useful existing lemmas which eliminated the need to write many lemmas ourselves.

However, one observation we have made during the process of formalizing our circuits was the significant time it took to accomplish this. This observation was also made by Shiraz and Hazan [SH17], where they have spent approximately 12 man-months to formalize and proof combinational circuits. It is crucial to consider whether the time and effort invested in formally verifying circuits is worthwhile. Even though this process is time-consuming, it is important to weigh it against the potentially high costs of errors in circuit designs, so that risks, for example the Intel FDIV bug discussed in Section 1.1, can be significantly reduced.

We illustrated the steps on how to make formal specifications for combinational and sequential circuits. Furthermore, we have also shown how to create functional implementations for these specifications and how we can utilize LEAN3 to prove that the implementations comply with the specifications. This has allowed us to answer our research question by demonstrating the process of formal specification and verification of circuits using LEAN3.

6.2 Discussion and Future Work

Challenges with Proofs

One of the biggest challenges faced during this research was with the proof of the n-bit full adder. While the steps to design the specification and implementation were manageable, the steps towards the verification became too complex. The amount of time reserved for this research, combined with the steep learning curve of the LEAN3 theorem prover, was not enough to fully understand the

concepts needed for verifying non-trivial properties of arrays.

Signals in Sequential Circuits
The way we used signals for sequential circuits in LEAN3 made it very non-intuitive to implement sequential circuits. While trying to design a program counter, we wanted to use two separate components and compose them. Unfortunately, the specific usage of our signals prevented us to do this. Future work might take a look at a different approach, where we can create circuits more intuitively, based on single clock times.

Extending the Research
While this thesis may be the initial step of formal verification of circuits in LEAN3, this research can be extended to more complex circuits. A couple examples are: specifying and implementing a processor, cache memory and a memory management unit (MMU). This research can be of particular interest to engineers and designers at companies such as ARM, Intel, and AMD. The implementation of formal verification can contribute to the functionality, reliability, safety, and security of digital circuits.

Memory Overhead
An interesting observation made while working with the LEAN3 theorem prover was the considerable amount of memory being used on the system running LEAN3, even for relatively small circuits. This might become a bottleneck if we want to verify larger, more complex circuits and is a direction for potentially fruitful future research.

Improving automation in the prover
An interesting subject for future research is to further improve the automation of the prover, specifically for proofs that are related to combinational and sequential circuits.

References

[AMD] Inc. Advanced Micro Devices. Amd product security bulletin: Cross-process information leak. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7008.html. Accessed: 2023-08-16.

[Any] AnySilicon. Understanding formal verification. https://anysilicon.com/understanding-formal-verification/. Accessed: 2023-08-10.

[Car] Mario Carneiro. The type theory of lean. https://github.com/digama0/lean-type-theory/releases. Accessed: 2023-08-18.

[CM73] Thomas J. Chaney and Charles E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22:421–422, 1973.

[Com] Lean Prover Community. Mathlib documentation. https://leanprover-community.github.io/mathlib_docs/. Accessed: 2023-08-10.

[Com21] 101 Computing. Binary shifters using logic gates. https://www.101computing.net/binary-shifters-using-logic-gates/, 02 2021. Accessed: 2023-08-05.

[Cor] Intel Corporation. Intel history: 1994 annual report. https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html. Accessed: 2023-08-16.

[End01] Herbert Enderton. *A Mathematical Introduction to Logic*. Academic Press, 2nd edition, 2001.

[HH07] David Money Harris and Sarah L. Harris. *Digital Design and Computer Architecture*. Morgan Kaufmann, 2007.

[Lam05] William K. Lam. *Hardware Design Verification: Simulation and Formal Method-Based Approaches*. Pearson, 2005.

[Lee92] Miriam Leeser. Using nuprl for the verification and synthesis of hardware. *Philosophical Transactions of the Royal Society of London. Series A: Physical and Engineering Sciences*, 339:49–68, 1992.

[LL16] Ing. Alberto I. Leibovich and Ing. Pablo E. Leibovich. Binary sequence detector. Technical report, AN-1139, 11 2016.

[Orm] Tavis Ormandy. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html. Accessed: 2023-08-22.

[ORSS92] S. Owre, J. M. Rushby, N. Shankar, and M. K. Srivas. A tutorial on using pvs for hardware verification, 1992.

[Pro] Lean Prover. Lean3 documentation. https://leanprover.github.io/reference/. Accessed: 2023-08-10.

[SH17] Sumayya Shiraz and Osman Hasan. A hol library for hardware verification using theorem proving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP, 05 2017.

[Sha38] Claude E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions of the American Institute of Electrical Engineers*, 57(12):713–723, 1938.