Formal Verification of Hardware using HOL

Marco Benini
Department of Computer Science
University of Milan

October 30, 2002

Contents

Chapter 1

Preface

Nowaday electronic digital circuits are used everywhere. Most of them are simple, low cost and solid circuits, but technological advances put every year on the market new, complex, high cost circuits, which are used to build kernels of sensitive systems like computers.

In early '80 most microprocessors had a maximum word of 8 bits; now most microprocessors are 32 bits and some even 64 bits. This growth in complexity arose a problem: how to ensure that a digital circuit is correct.

By correcntess we intend that every circuit has an intended behaviour and the goal of a project is to ensure the the *real* circuit behaves exactly how it is intended.

The traditional solution to this problem was: let us build the circuit, let us test it and if it does not do what we mean, let us correct the project. This solution is unsatisfactory for three reasons:

- it costs: producing a wrong VLSI circuit implies to spend a lot of money.
- we are not able to test a circuit extensively when it is too complex, because there are too many cases.
- project and testing are separate activities and they cannot benefit immediately from mutual cooperation.

Now electonic engineers use a different approach: simulation.

Simulation solves two of the three problems cited above. It reduces costs because a circuit is produced only when all testings are passed. It can be performed during design, so that engineers can discover major faults before producing the whole project.

But complexity is still a challenge for simulation.

Researchers tried a completely different approach: not testing correctness of a circuit by trying how it behaves when certain inputs are given, but proving mathematically from its structure that it enjoys enough properties to assert it is correct. This approach is called *formal verification*.

Some of these techniques have already reached a point such that they are no more research, but they can be applied successfully to real design. The goal of this book is to present one of these methods, the most known, and to show how it can be applied to prove correctness of arbitrary digital circuits.

Because it does not matter behavioral complexity (that is, the number of possible behaviours of a circuit), but only structural complexity (that is, the number of components), formal verification is a powerful technique.

In chapter 2 we will describe from a theoretical point of view how to represent circuits by means of higher-order logic formulas, and properties and limits of these descriptions.

In chapter 3 we will describe how to represent logical gates and combinational circuits, while chapter 4 is devoted to the description of how to formalize and prove correctness of sequential circuits.

Chapter 5 describes finite state machines, while chapter 6 is an overview on how to verify microprocessors.

Chapter 7 gives account to other important techniques that are part of the general framework of formal verification.

This book assumes confidence with mathematical logic , hardware (of course) and a small training in the use of HOL, a generic theorem prover useful to reason about our representations.

Acknowledgments

Financial support for the development of this book was given by SGS-Thomson, and in particular Ing. Di Bona.

Coordination and main ideas are from Prof. Degli Antoni, Departement of Computer Science, University of Milan.

A special thank to Computational Architectures Laboratory (Departement of Computer Science – University of Milan) and in particular to Dott. Vaccari and Dott. Buffoli.

Chapter 2

Introduction to Formal Verification

Nowadays, hardware is everywhere. Watches, pocket calculators, washing machines, computers, cars, and so on are built on the top of electronic digital circuits.

In most cases, correctness of these circuits is not a major issue: if a watch gains or loses a little time per day, you can simply reset it, for example, once per month.

Some applications of hardware are not error sensitive, that means, if they provide some basic functionality, then errors are not so important, as is the case of watches.

But other applications are more critical: if the microprocessor in a washing machine halts for a bug, then dresses you are washing, will be damaged, and this is not an acceptable behavior.

We will distinguish the case of a broken circuit from an erroneous one: the former is a problem of electronic engineering, the latter is a design problem. In everyday life, we accept that things get broken, but we don't like that things normally don't produce a correct behavior.

There are hardware applications where a circuit has to be perfect: you declare it can do something, and it must do it. An example of such an application is the microprocessor in a computer: if it produces a wrong result every 100.000.000.000 instructions, then, with a clock frequency of 50MHz, you, the user, will see an error every 33 minutes.

Formal verification is a mathematical technique to assure that a circuit do something in the right way. It does not simulate the circuital behavior, nor it tries all possible cases, but it derives, from the structure of the circuit, by means of a mathematical proof, that a property holds.

Let's examine what we need to do so:

- **a specification**
  we need the description of what the circuit is intended to do. Such a description must be unambiguous, and very precise, and it must not refer

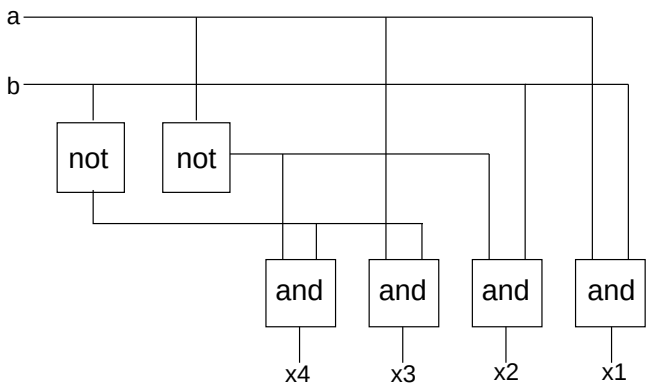


Figure 2.1: The 2-to-4 multiplexer

to hardware, but only on the functions it is intended to implement. For example, a specification of a watch may be "after an hour in real time, it has to show the initial time plus exactly one hour".

In general, natural language descriptions are not unambiguous, nor precise, so we will use mathematical descriptions by means of the language of logic.

- **an implementation**
  we need the circuit. We use it to produce a description of its structure in the language of mathematical logic. We claim that such a description is equivalent under some general hypothesis to the real circuit.

- **a correctness property**
  we need a statement that says what is our notion of correctness for a particular circuit, specification pair. Again this will be a logic formula, so all the description in the verification problem will be stated using the language of mathematical logic.

When we have these three element, we have to prove the formula representing the correctness argument. Such a proof can be conducted by hand, but it is better to use a theorem prover. We choose *HOL* because it has the right logic, a lot of tools, and because it has a well established tradition in formal verification of hardware.

In the rest of this chapter, we will analyze from a theoretical point of view some aspects of our approach. In next chapters, we will present the technical part behind the words that follow.

2.1 Describing Circuits

Now we present the approach we use to describe circuits as logical formulas.

Let's see an example: the 2-to-4 multiplexer. Its schema is in figure 2.1. Our goal is to translate this schema in a logical formula by means of a strict representation.

2.1.1 Basic Elements

To represent the circuit, we need to represent its basic elements. In our example this means to represent AND and NOT gates.

To represent a basic elements means:

- to define a predicate for such a component.

  In our example we have: $\mathrm{AND}(a, b, out)$ and $\mathrm{NOT}(a, out)$.

  Notice that defining a predicate means to give it a name and to declare the number of arguments along with their types. The number of argument is the number of lines entering in or exiting from the component. The type of an argument depends on some general choices on how to represent circuits. In general, the type of a line in a combinational circuit is $boolean$, and in a sequential circuit, it is $Time \longrightarrow boolean$, a function from time domain to boolean values.

- to give a definition for such a predicate.

  In our example, we define:

  $$\mathrm{AND}(a, b, out) = (out = a \wedge b) \tag{2.1}$$

  and

  $$\mathrm{NOT}(a, out) = (out = \neg a) \tag{2.2}$$

  Notice the type forcing: in the latter definition, we infer that a: $boolean$ and out: $boolean$. Our reasoning is the following:

  1. $\neg$: $boolean \longrightarrow boolean$, so we deduce a: $boolean$.
  2. $=$: $X \times X \longrightarrow boolean$, but the right hand side has type $boolean$, so $X = boolean$.
  3. out: X, so out: $boolean$.
  4. NOT: $boolean \times boolean \longrightarrow boolean$.

Normally a complex project will be verified bottom-up: we begin verifying very small components, and then we use them to verify bigger ones, until we are able to verify the complete circuit.

Usually we don't verify gates, nor we define them. They have their standard definition in a library, as almost all usual components.

2.1.2 Wires

Now we need to model wires. The simplest way is to define a wire as a variable. In this way, components can be combined simply by the $\wedge$ operator.

In our example, the circuit can be described as:

$$\begin{array}{ccccc} \mathrm{NOT}(a,c) & \wedge & \mathrm{NOT}(b,d) & \wedge & \\ \mathrm{AND}(a,b,x_1) & \wedge & \mathrm{AND}(c,b,x_2) & \wedge & \\ \mathrm{AND}(a,d,x_3) & \wedge & \mathrm{AND}(c,d,x_4) & & \end{array} \tag{2.3}$$

Some aspects of this description choice are important:

- is possible to join components by conjunction only if the type of variables (wires) is shared between predicates. Here is where typing vincles are generated.
- the meaning of our representation is: if we substitute an arbitrary set of values to variables, then the logic formula we claim to represent our circuit, is true if and only if output values are generated by input values, and internal values are the ones computed by basic components.

2.1.3 I/O versus Internal Wires

In the description we have given above (equation 2.3), there is no distinction between I/O and internal wires. This is a problem because the behavior of a circuit has to be described only from inputs and outputs (and states).

We have seen that the meaning of equation 2.3 is: given a substitution of boolean values for a, b, c, d, x_1, x_2, x_3, x_4, the formula is true iff it describes a valid behavior of the circuit, i.e. iff the values on wires are right according to the meaning of elementary components. Internal wires values are computed by components and the only thing we need to know is that such values exist. So the formal solution to the problem of internal wires is to quantify existentially their variables.

In our example this means:

$$\begin{array}{cccccc} \exists c,d. & \mathrm{NOT}(a,c) & \wedge & \mathrm{NOT}(b,d) & \wedge & \\ & \mathrm{AND}(a,b,x_1) & \wedge & \mathrm{AND}(c,b,x_2) & \wedge & \\ & \mathrm{AND}(a,d,x_3) & \wedge & \mathrm{AND}(c,d,x_4) & & \end{array} \tag{2.4}$$

2.1.4 Black Boxing

Our descriptions can become quickly very long. A way to treat them symbolically will help.

Another reason to hide descriptions is that we would like that complex circuits can be described in the same way their components are. Such a wish is interesting because it will help in developing the verification of very complex circuits: we could begin to verify smaller components, then we could verify bigger ones, knowing that smaller are correct.

There is another reason, much more theoretical: a circuit has a structure, which means a behavior, and it computes a function, that we assume to know. The goal of formal verification is to prove that the behavior implements the function. A valid description of a correct circuit is either the behavior i.e. its structure, or the function, i.e. its specification. It is the same as computer programming: we can develop a program having its subroutines already written, or supposing it. If their implementation is right it makes no difference in the development of an higher level subroutine when we write them.

To produce a black box, which hides the implementation behind a predicate, all we have to do is to write a definition. In our example:

$$\begin{aligned} \mathrm{CIRCUIT}(a,b,x_1,x_2,x_3,x_4) = & \qquad (2.5)\\ \exists c,d.\quad & \mathrm{NOT}(a,c) \wedge \mathrm{NOT}(b,d) \wedge \\ & \mathrm{AND}(a,b,x_1) \wedge \mathrm{AND}(c,b,x_2) \wedge \\ & \mathrm{AND}(a,d,x_3) \wedge \mathrm{AND}(c,d,x_4) \end{aligned}$$

2.2 Description Limits

In this section we will analyze what are the limits of our approach.

The reasons for limits are two: formalization and physics. Physics is a problem because circuits obey to its laws, and our mathematical descriptions do not. We introduced simplifications of physical laws to describe the behavior of circuits in term of mathematical types. It is important to know what we can describe and what we cannot. In every case, formalization implies not to model some behaviors: a more complete model implies a more complex formal structure.

2.2.1 Interference and Analogical Components

Real digital circuits are analogical: an AND gate will not take 0 and 1 as input values, but it takes tension values, i.e. continuous quantities. Every gate can work only in well defined range of its physical dimensions: too high temperature, or too low distance from another gate are reason by which a component does not work properly.

Our approach was simple: we assume that a circuit does not have any interference among components, and we assume that dimensions of the original circuit are in the "natural" ranges. In other words we assume that every component can communicate with others or with the external world only through its I/O wires, and we assume that our (discrete) types are an acceptable model for the continuous quantities that make a real circuit work.

2.2.2 Stability

Another property we pretend a circuit satisfies is stability. Stability is the property such that every output value of a basic component that is part of a

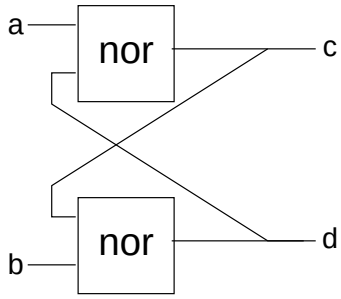


Figure 2.2: A simple flip-flop

circuit does not change during an observational phase. An observational phase is, in principle, instantaneous, but discrete.

An example of a circuit not satisfying such a requirement is a flip-flop. For example, looking at figure 2.2, we can compute the following truth table, were subscripts to output variables are observational phases:

a	b	c_0	d_0	c_1	d_1	c_2	d_2	c_3	d_3
0	0	0	1	0	1	0	1	0	1
0	0	1	0	1	0	1	0	1	0
0	1	0	1	0	0	1	0	1	0
0	1	1	0	1	0	1	0	1	0
1	0	0	1	0	1	0	1	0	1
1	0	1	0	0	0	0	1	0	1

We observe that:

- there are two cases where two observational phases were needed to reach stability: they are $a = 0$, $b = 1$, $c_0 = 0$, $d_0 = 1$, and $a = 1$, $b = 0$, $c_0 = 1$, $d_0 = 0$.
- in such cases, values of c and d wires change, while values of a and b are stable.

Our representation technique cannot describe an unstable behavior. In the case of a flip-flip, our description technique can easily represent it as a black box, but it cannot prove it works correctly, because our approach implies that there is strong stability.

In other words, we can describe in every level of detail circuits where there are no fluctuations on wires: stability must be reached in no more than one observational phase.

2.2.3 Level of Description

From the discussion on stability, an important property of our description method appears: levels. If we encapsulate a flip-flop in a black box, no stability problem arises, because its external behavior is stable: we have a *set*

command, a *reset* command, and a *read* command. They behave exactly in the way you expect.

This property works well forward, but not backward: from a complex circuit description, you can switch to a simpler one by black-boxing (a step forward, to abstraction), but, in general you cannot do the reverse (as the flip-flop example teaches).

Cleary this shows a limit to our approach: we cannot prove correctness "from silicon", because we must meet stability vincles. So, in our approach, we cannot prove that a circuit is perfect, that means correct without hypothesis, but we can prove that a circuit is correct at a certain description level, for example at gate level, and we can prove that abstractions are correct from our level up.

Our approach is well suited to describe circuits where the sentence "a component is a black box" is absolutely true. In some range of physical values, and in non-critical situation, that abstraction is met.

2.2.4 The Homomorphic Property

The homomorphic property is the name we give to the ability to represent the same circuit at different level of abstraction in a coherent way.

The homomorphic property states that, for every valid abstraction level, exists a provable correct implementation that is simpler in the sense of basic components required, till to the lowest level.

The lowest level of representation is the one where no physical problem exists and stability is reached under the usual hypothesis: for combinational circuits, the lowest level is gate level, associating 1 with $true$ and 0 with $false$, the basic component with logical operations and wires with boolean values; for sequential circuits the lowest level is gate level, except for flip-flop, represented as black boxes.

This property, and its reverse, we have seen in the previous subsection, is useful in practice because we are sure that, if we can prove correctness of a circuit represented at a certain level of abstraction, then, every abstraction (that is, every representation of the same circuit, with more black-boxed components) is also correct, and every specialization (that is, every representation of the circuit with exploded components) is correct if, and only if, the component explosion is correct (i.e. it respects vincles on the component), and the whole specialization respects condition on stability.

An important consequence of these properties is that, on an higher level of abstraction, we can substitute an implementation with the corresponding specification, while, on a lower level of abstraction, we can substitute a specification, or a high level implementation, with a low level implementation, preserving correctness, if our substitutions do.

The homomorphic property, here expressed in a somewhat theoretical way, is very important because let us work on different levels of representation of the same circuit without redoing all the verification work. It is the reason why we adopted this approach, although its limitations.

2.3 Correctness Arguments

In this section we examine some typical way to express that a circuit is correct. We assume to have an implementation, let say $\mathrm{IMPL}(\mathrm{in}, \mathrm{out})$, and a specification of what the circuit is intended to do, let say $\mathrm{SPEC}(\mathrm{in}, \mathrm{out})$.

We will present below some approaches to write correctness properties, and we will discuss their merits and their flaws.

2.3.1 Equivalence

The equivalence argument is very simple:

$$\forall \mathrm{in}, \mathrm{out}.\mathrm{IMPL}(\mathrm{in}, \mathrm{out}) \equiv \mathrm{SPEC}(\mathrm{in}, \mathrm{out}) \tag{2.6}$$

Its meaning is quite simple: if $\mathrm{in}, \mathrm{out}$ satisfy IMPL, then they are values that the circuit may produce, and so they must satisfy SPEC, i.e. in such a case the circuit behaves correctly; if $\mathrm{in}, \mathrm{out}$ do not satisfy IMPL, then they are not something the circuit may produce, so they must not satisfy the specification.

This is a very good way to specify correctness: it say the circuit behaves exactly in the way its specification says. This means that the set of possible behaviors (the one specified with IMPL) is exactly the same of the set of acceptable behaviors (the one specified with SPEC).

Advantages of this kind of argument are obvious: when we verify the argument, we are absolutely sure that the circuit will not give incorrect answers, but we are also sure that the circuit may give all possible correct answers.

Disadvantages are obvious too: in a lot of cases we are not interested in behavioral completeness, but only in correctness of all possible behaviors. It seems simpler to prove that all behaviors are correct, than to prove that all correct behaviors are produced. It is also difficult to characterize, in some cases, the exact set of possible behaviors, but can be much more simple to give a superset of possible behaviors.

In almost all the elementary examples in this book, we will use this kind of argument. It works well on small examples, but not on complex ones.

2.3.2 Implication

The implicational argument is similar to the equivalence one, but its meaning is simpler. The form of the argument is:

$$\forall \mathrm{in}, \mathrm{out}.\mathrm{IMPL}(\mathrm{in}, \mathrm{out}) \Rightarrow \mathrm{SPEC}(\mathrm{in}, \mathrm{out}) \tag{2.7}$$

Its meaning is simple: for all in, out pairs satisfying the IMPL predicate, they have to satisfy also the SPEC predicate. In other words, every possible behavior of the circuit must be an acceptable one. From another point of view, we can express the meaning as: the set of possible behaviors (the one denoted by IMPL) is a subset (possibly equal) of the set of acceptable behaviors (that is SPEC).

Advantages of such an approach are obvious: we are sure that a circuit satisfying an implicational argument will always produce a correct answer. This kind of arguments, most of the time, are simpler to prove than their equational counterparts.

Disadvantages are less obvious: if we prove that IMPL $\subseteq$ SPEC, then all behaviors in IMPL must be acceptable, but no one assure us that IMPL contains some useful behavior. An (extreme) example is the following: let our circuit be the null circuit, that is, the circuit with no output lines. Obviously, in this case IMPL represents the $\emptyset$ set, but $\emptyset \subseteq$ SPEC, so we deduce that our circuit is correct!

The disadvantage is in the fact that, in this argument, correctness is defined as no violation.

The implication argument is the most used, and problems regarding the significance of possible behaviors set are solved by intuition, i.e. we say something like: "obviously this circuit produce a lot of things" and the correctness of these "things" is assured by a formal verification using an implicational argument.

2.3.3 Boundary Conditions

Another way to express correctness is to set an argument and to limit its scope to a well known range. The form of an implicational argument with a boundary condition is:

$$\forall \mathrm{in}, \mathrm{out}.\mathrm{BCOND}(\mathrm{in}, \mathrm{out}) \Rightarrow (\mathrm{IMPL}(\mathrm{in}, \mathrm{out}) \Rightarrow \mathrm{SPEC}(\mathrm{in}, \mathrm{out})) \tag{2.8}$$

and the form of an equivalence argument with a boundary condition is:

$$\forall \mathrm{in}, \mathrm{out}.\mathrm{BCOND}(\mathrm{in}, \mathrm{out}) \Rightarrow (\mathrm{IMPL}(\mathrm{in}, \mathrm{out}) \equiv \mathrm{SPEC}(\mathrm{in}, \mathrm{out})) \tag{2.9}$$

In the development of proofs the $\mathrm{BCOND}(\mathrm{in}, \mathrm{out})$ will become an intermediate hypothesis for the body of the argument, thus restricting the range of possible values in, out may take.

This is the right method to express the situation where circuits can receive as inputs only a limited range of values, and interest in correctness proof is on these values, because they are the only admissible ones.

2.3.4 Comparison

An important and useful way to express a correctness argument is to compare two circuits that we claim to express the same function. Formally the argument takes the form:

$$\forall \mathrm{in}, \mathrm{out}.\mathrm{SPEC1}(\mathrm{in}, \mathrm{out}) \equiv \mathrm{SPEC2}(\tau(\mathrm{in}), \tau(\mathrm{out})) \tag{2.10}$$

where τ is a biiejctive function.

The meaning of this argument is simple: the first circuit is equivalent to second one, by a information preserving transformation, τ, that reinterprets inputs and outputs properly.

This technique is useful in two cases:

- when we know that a circuit is correct, but, for a revision of the project, it is substituted with a higher performance circuit, which we want to be substantially equivalent.

- when it is simpler to derive correctness for an elementar circuit, and there is a clear mapping between that one, and ours.

2.4 Verification versus Testing

We finish this chapter with a brief analisys of formal verification versus testing, showing that these two approaches are complementary and both useful.

Testing, or simulation, is the most used method to verify that a circuit meets its specification. The idea is very simple: in principle a circuit is correct if all its possible behaviors are, so if, for every input, the implementation gives the expected result, then the circuit is correct.

There are two obvious drawbacks in this approach: extensive testing is very time consuming; an error in the implementation force the correction of the project, and to redo a complete series of tests.

It is exactly the same situation for programming: debugging can point out errors, but, if it is not exhaustive, it cannot assure correctness.

On the other hand, testing is much more simple than verification: all we have to do is to simulate the circuit with a given input, and to look if output is the expected one.

Formal verification is, in some sense, the exact opposite with respect to simulation. Its result are easy to reuse when there are changes in the project; verification of each component is slow, but it assures total correctness.

Another important aspect in the comparison between simulation and formal verification is how much information is produced when an error is encountered. Simulation gives a little amount of information: an example of when things go wrong, and an idea of where the error takes place. Verification shows exactly why the error comes in, and gives, in an implicit way, the set of all examples that can generate the erroneous behavior.

So it seems that formal verification is much better than simulation as a general technique to assure quality in circuit design.

That is not so true: simulation has its place, where formal verification cannot be applied, or where its application can be very expensive.

Formal verification, in principle can go down till the gate level, assuming physical dimensions are acceptable, but when you have to deal with transistors, resistors, and other low-level components, the real ones, the only one that really makes the job, you must use simulation. Formal verification can be used to assure correctness of the project of a circuit, that is, of an abstraction of the real-world circuit, while simulation works only with the real-world circuit, ignoring the project.

This kind of simulation will be never thrown away by formal verification, because it does its tests on the final product, the one that has to work.

High level simulations are clearly unuseful, when you adopt formal verification, because the latter do the same job, in about the same time, with better result, and giving much more information on eventual problems.

Chapter 3

Gates and Basic Circuits

In this chapter we will work on two main topics: representation and verification of basic gates, and formal verification of combinational circuits.

3.1 Logical Gates

To represent logical gates, that are the simplest elements we deal with, we need a base, that means a set of gates we assume to be correctly implemented.

Then we have to develop, for every gate not in the base, a representation and a specificaton, and we must verify that the gate is correct.

We will collect all these proofs and definitions in a library, we will use as a support in the verification of more complex circuits.

3.1.1 Choosing a Base

There are two constraints when we choose a base:

- its gates must be enough powerful to represent all possible boolean operations, that means, every operation could be represented as the combination of some basic gates.
- its gates must be easily implemented in hardware.

These constraints give us an answer:

- TTL (Transistor-Transistor Logic) circuit family is based on a single basic gate: NAND.
- The NAND boolean operation is a base for boolean functions, i.e. every function of type $Boolean \longrightarrow Boolean$ can be represented as a proper composition of NAND operators, along with the basic boolean values.

So we choose as our base the set $\{\mathrm{NAND}\}$. Now we have to formally specify the meaning of the NAND operator. This is simple:

$$\forall a, b, out.\ nand2(a, b, out) \equiv (out \equiv \neg(a \wedge b))$$

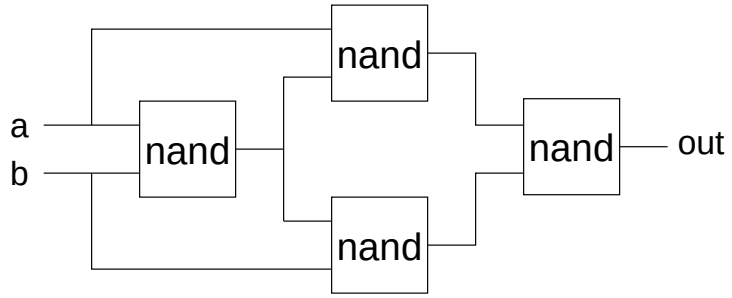


Figure 3.1: The binary XOR gate

Using the HOL syntax, this specification becomes:

```
let NAND2 = new_definition
            (‘NAND2‘,
             "!a b out. nand2 a b out = (out = ~(a /\ b))");;
```

Some comments on notation are important:

- NAND2 is the name of the definition, the one used in the application of inference rules.

- $nand2$ is the name of the binary operator that represents the logical NAND operation.

3.1.2 Verification of Gates

As we have seen in the preceding chapter, to verify a circuit we need a description, a specification and a correctness argument.

As an example, let take the binary XOR gate, in figure 3.1. Let's begin with the implementation: there are three internal wires, let say p, q, and r, and there are three I/O wires, a, b, out. Using the method described in the previous chapter, and black-boxing, we obtain the following description:

$$\begin{array}{lrll} \forall a, b, out. & xor2\_impl \equiv & & \\ & \exists p, q, r. & nand2(a, b, p) & \wedge \\ & & nand2(a, p, q) & \wedge \\ & & nand2(p, b, r) & \wedge \\ & & nand2(q, r, out) & \end{array}$$

Using the HOL syntax, this description becomes:

```
let XOR2_IMPL = new_definition
    (`XOR2_IMPL`,
     "!a b out. xor2_impl a b out =
                ?p q r. nand2 a b p /\
                        nand2 a p q /\
                        nand2 p b r /\
                        nand2 q r out");;
```

The name of this definition is XOR2_IMPL, while the name of the operator is xor2_impl. Our convention will be:

- The name of the definition of the implemention will be in uppercase and will end with _IMPL.
- The name of the definition of the specification will be in uppercase, and will be the same as the implementation's except that it will not end with _IMPL.
- The name of defined relations will be the same as the definition's one, but in lowercase.
- If we need to give a name to the proof, then it will be the same specification's one, but with an _PROOF at the end.

The rules on names of specification's parts is so because of the homomorphic property: when we will use the component in a more complex circuit, we will substitute its implementation with the specification, that is simpler.

So let's specify what a XOR gate has to do.

The rule is simple: $\mathrm{XOR}(a, b) = true$ iff $a \neq b$. Formally

$$\forall a, b, out.xor2(a, b, out) \equiv (out = \neg(a = b))$$

In the HOL syntax, this becomes:

```
let XOR2 = new_definition
    (`XOR2`,
     "! (a:bool) b out. xor2 a b out =
                (out = ~(a = b))");;
```

Now our correctness argument will be the equivalence between the implementation and specification. In HOL syntax this becomes:

```
set_goal([], "! a b out. xor2_impl a b out = xor2 a b out");;
```

Now we have to prove the goal. Let's examine it:

```
"! a b out. xor2_impl a b out = xor2 a b out"
```

The only thing we can do is to expand the definition of xor2_impl and xor2 by rewriting. The command (preceded by the prompt #) along with the resulting ouput is:

```
# e(REWRITE_TAC [XOR2_IMPL; XOR2]);;

"!a b out.
  (?p q r. nand2 a b p /\ nand2 a p q /\ nand2 p b r /\
           nand2 q r out) =
  (out = ~(a = b))"
```

Again the most obvious choice is the best: let's rewrite the goal using the definition of nand2:

```
# e(REWRITE_TAC [NAND2]);;

"!a b out.
  (?p q r.
    (p = ~(a /\ b)) /\
    (q = ~(a /\ p)) /\
    (r = ~(p /\ b)) /\
    (out = ~(q /\ r))) =
 (out = ~(a = b))"
```

Now let's throw away those unuseful universally quantified variables:

```
# e(REPEAT STRIP_TAC);;

"(?p q r.
     (p = ~(a /\ b)) /\
     (q = ~(a /\ p)) /\
     (r = ~(p /\ b)) /\
     (out = ~(q /\ r))) =
  (out = ~(a = b))"
```

To prove a goal of the form $A \equiv B$, the simplest thing is to prove $A \Rightarrow B$ and $B \Rightarrow A$. Because splitting the goal generates a bit of garbage, we get rid of it by stripping the result.

```
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

2 subgoals
"?p q r.
  (p = ~(a /\ b)) /\
  (q = ~(a /\ p)) /\
  (r = ~(p /\ b)) /\
  (out = ~(q /\ r))"
  1  ["out = ~(a = b)" ]

"out = ~(a = b)"
  4  ["p = ~(a /\ b)" ]
  3  ["q = ~(a /\ p)" ]
  2  ["r = ~(p /\ b)" ]
  1  ["out = ~(q /\ r)" ]
```

As you can see, what we called "garbage" is the set of hypothesis in the latter subgoal.

A goal of the form $\vdash \exists x.A \wedge B \Rightarrow C$ could be transformed, using natural calculus' inference rules into $\exists x.A \wedge B \vdash C$, and, replacing x with a fresh variable, we could obtain $A \wedge B \vdash C$; the last step is to notice that the conjunction on the left side of the $\vdash$ operator is equivalent to the pairing operation over hypothesis, so the final result is $A, B \vdash C$. But there is the STRIP_TAC, which does all the work.

The latter is the current subgoal. We could rewrite the goal with assumption 1, then with assumptions 2 and 3, and then with 4. Perhaps this rewriting will give us a simplified goal. It seems a good idea, let's try it:

```
# e(ASM_REWRITE_TAC []);;

"~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b)) = ~(a = b)"
  4  ["p = ~(a /\ b)" ]
  3  ["q = ~(a /\ p)" ]
  2  ["r = ~(p /\ b)" ]
  1  ["out = ~(q /\ r)" ]
```

It seems to be a complex goal, but, if you look closer, you will see that it is propositional, that is, there are no quantifiers. But propositional expression are decidable, and HOL gives us a decision procedure: it is called TAUT_TAC.

So, let's see if the goal is true, using the tautology checker:

```
# e(TAUT_TAC);;

goal proved
|- ~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b)) = ~(a = b)
.... |- out = ~(a = b)

Previous subproof:
"?p q r.
  (p = ~(a /\ b)) /\
  (q = ~(a /\ p)) /\
  (r = ~(p /\ b)) /\
  (out = ~(q /\ r))"
  1  ["out = ~(a = b)" ]
```

Now let's solve the other subgoal. The outermost functor is an $\exists$, and the best way to prove an existencial goal is to substitute the right term in the quantified variable. But, in our case, this is very simple: we read that $p = \neg(a \wedge b)$. Let's see what happens:

```
# e(EXISTS_TAC "~(a /\ b)");;

"?q r.
  (~(a /\ b) = ~(a /\ b)) /\
  (q = ~(a /\ ~(a /\ b))) /\
  (r = ~(~(a /\ b) /\ b)) /\
  (out = ~(q /\ r))"
  1  ["out = ~(a = b)" ]
```

We have again an existencial goal, and, for both the quantified variables, we have the right substitution. So let's do the job:

```
# e((EXISTS_TAC "~(a /\ ~(a /\ b))") THEN
    (EXISTS_TAC "~(~(a /\ b) /\ b)"));;

"(~(a /\ b) = ~(a /\ b)) /\
 (~(a /\ ~(a /\ b)) = ~(a /\ ~(a /\ b))) /\
 (~(~(a /\ b) /\ b) = ~(~(a /\ b) /\ b)) /\
 (out = ~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b)))"
  1  ["out = ~(a = b)" ]
```

On the first, second, and the third row there is a complex term, but it is trivial to prove that it holds: every row has the form $A = A$. On the fourth row the simplest thing to do is the replace out with $\neg(a = b)$, as assumption 1 says.

```
# e(ASM_REWRITE_TAC []);;

"~(a = b) = ~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b))"
  1  ["out = ~(a = b)" ]
```

It is another propositional goal: something for the tautology checker:

```
# e(TAUT_TAC);;

goal proved
|- ~(a = b) = ~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b))
. |- (~(a /\ b) = ~(a /\ b)) /\
     (~(a /\ ~(a /\ b)) = ~(a /\ ~(a /\ b))) /\
     (~(~(a /\ b) /\ b) = ~(~(a /\ b) /\ b)) /\
     (out = ~(~(a /\ ~(a /\ b)) /\ ~(~(a /\ b) /\ b)))
. |- ?q r.
      (~(a /\ b) = ~(a /\ b)) /\
      (q = ~(a /\ ~(a /\ b))) /\
      (r = ~(~(a /\ b) /\ b)) /\
      (out = ~(q /\ r))
. |- ?p q r.
      (p = ~(a /\ b)) /\
      (q = ~(a /\ p)) /\
      (r = ~(p /\ b)) /\
      (out = ~(q /\ r))
|- (?p q r.
     (p = ~(a /\ b)) /\
     (q = ~(a /\ p)) /\
     (r = ~(p /\ b)) /\
     (out = ~(q /\ r))) =
   (out = ~(a = b))
|- !a b out.
    (?p q r.
      (p = ~(a /\ b)) /\
      (q = ~(a /\ p)) /\
      (r = ~(p /\ b)) /\
      (out = ~(q /\ r))) =
    (out = ~(a = b))
|- !a b out.
    (?p q r. nand2 a b p /\ nand2 a p q /\ nand2 p b r /\
             nand2 q r out) =
    (out = ~(a = b))
```

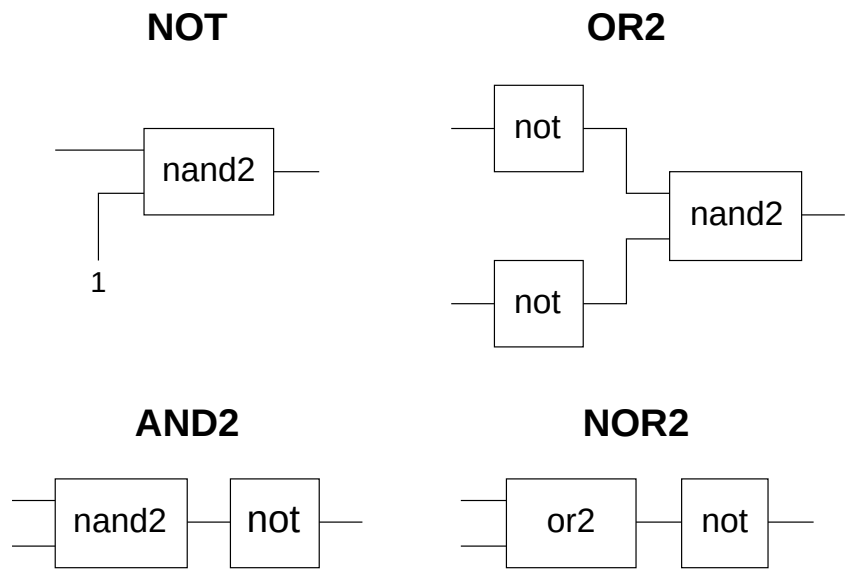


Figure 3.2: Implementation of binary gates

```
|- !a b out. xor2_impl a b out = xor2 a b out

Previous subproof:
goal proved
```

3.1.3 Developing a Library

What we have done with the XOR2 component, can be replicated for every gate.

We want to develop a library containing the definitionof every gate, along with correctness proof.

Let's plan our action; in order, what we need to do is:

- to decide what gates are needed
- to implement them in term of the NAND2 gate
- to specify what they are intended to do
- to prove their correctness
- to package all the previous definitions and proofs as a library for future use.

Let's begin! Obviously the NAND2 gate is needed, because it is our base. The simplest binary operation, such as and, or, not, are useful: let's call them AND2, OR2, NOT. Other useful gates are NOR2, and XOR2, which implement the binary operations nor and exclusive-or.

It will be useful to have a multiple input version of these gates. Because we do not know a priori the number of input lines that will be needed, then we have to implement those gates as extensible.

Without analyzing all details, refer to figure 3.2 to see how the binary version of the preceding gates could be implemented (the XOR2 gate we will use is described in the preceding subsection).

Specifications of the binary gates are summarized below. They are the obvious one: these circuits are so simple that there is no discussion on what they are intended to do.

$$\begin{aligned} not(a, out) &\equiv (out \equiv \neg a) \\ and2(a, b, out) &\equiv (out \equiv a \wedge b) \\ or2(a, b, out) &\equiv (out \equiv a \vee b) \\ nor2(a, b, out) &\equiv (out \equiv \neg(a \vee b)) \end{aligned}$$

In order to prove correctness of implementations of our gates, we need to state in HOL syntax implementations, specifications and correctness arguments.

Now we will develop those statements for every gate separately, along with the corresponding proof.

Let's begin with the NOT gate. Implementation and specification in HOL syntax have the form:

```
let NOT_IMPL = new_definition
    (‘NOT_IMPL‘,
     "! a out. not_impl a out = nand2 T a out");;

let NOT = new_definition
     (‘NOT‘,
      "! a out. not a out = (out = ~a)");;
```

To prove correctness argument, that is equivalence between specification and implementation, is simply a matter of rewriting:

```
set_goal([],"! a out. not a out = not_impl a out");;
e(REWRITE_TAC [NOT_IMPL; NOT; NAND2]);;
```

We can pack the proof in the following way:

```
let NOT_PROOF = prove_thm(‘NOT_PROOF‘,
        "! a out. not a out = not_impl a out",
        (REWRITE_TAC [NOT_IMPL; NOT; NAND2])
);;
```

The name of the theorem that state correctness argument is, according to our conventions, NOT_PROOF.

We can do the same thing for the OR2 gate:

```
let OR2_IMPL = new_definition
    (‘OR2_IMPL‘,
     "!a b out. or2_impl a b out =
        ? d e. (not a d /\ not b e /\ nand2 d e out)");;

let OR2 = new_definition
    (‘OR2‘,
     "!a b out. or2 a b out = (out = a \/ b)");;
```

To prove its correctness, we begin in the usual way. We set up correctness argument, then we rewrite our definitions, and we strip out the result, obtaining:

```
# set_goal([], "! a b out. or2_impl a b out = or2 a b out");;
# e(REWRITE_TAC [OR2_IMPL; OR2; NOT; NAND2]);;
# e(REPEAT STRIP_TAC);;

"(?d e. (d = ~a) /\ (e = ~b) /\
              (out = ~(d /\ e))) = (out = a \/ b)"
```

The principal functor of that goal is $=$, so we use EQ_TAC to split the goal, then we strip the result, obtaining:

```
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

2 subgoals
"?d e. (d = ~a) /\ (e = ~b) /\ (out = ~(d /\ e))"
  1  ["out = a \/ b" ]

"out = a \/ b"
  3  ["d = ~a" ]
  2  ["e = ~b" ]
  1  ["out = ~(d /\ e)" ]
```

The latter can be solved, simply by rewriting out and by propositional calculus:

```
# e(ASM_REWRITE_TAC []);;
# e(TAUT_TAC);;
```

The former has to obvious existential quantifiers to get rid of, with the result:

```
# e((EXISTS_TAC "~a") THEN (EXISTS_TAC "~b"));;

"(~a = ~a) /\ (~b = ~b) /\ (out = ~(~a /\ ~b))"
  1  ["out = a \/ b" ]
```

By rewriting out with the assumption and with propositional calculus, we are able to conclude the proof:

```
e(ASM_REWRITE_TAC []);;
e(TAUT_TAC);;
```

AND2 gate is quite similar, both in definitions and in proof development:

```
let AND2_IMPL = new_definition
    (‘AND2_IMPL‘,
     "! a b out. and2_impl a b out =
                 ?c. (nand2 a b c /\ not c out)");;

let AND2 = new_definition
    (‘AND2‘,
     "! a b out. and2 a b out =
                 (out = a /\ b)");;

set_goal([], "! a b out. and2_impl a b out = and2 a b out");;
e(REWRITE_TAC [AND2_IMPL; AND2; NOT; NAND2]);;
e(REPEAT STRIP_TAC);;
e(EQ_TAC THEN REPEAT STRIP_TAC);;
e(ASM_REWRITE_TAC []);;
e(EXISTS_TAC "~(a /\ b)");;
e(ASM_REWRITE_TAC []);;
```

The last binary gate we have decided to describe is NOR2 gate. Its circuit has the same structure of AND2's one. So definitions and corresponding correctness proof are very similar:

```
let NOR2_IMPL = new_definition
    (‘NOR2_IMPL‘,
     "!a b out. nor2_impl a b out =
        ?c. (or2 a b c /\ not c out)");;
```

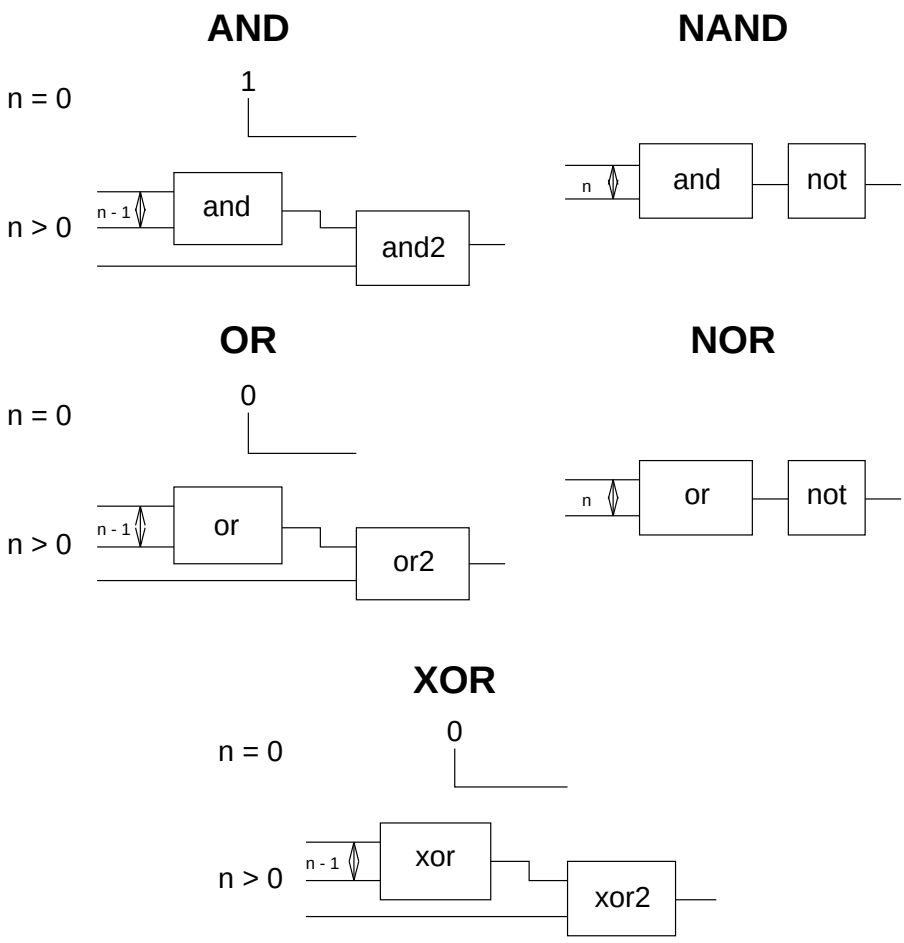


Figure 3.3: Implementation of generalized gates

```
let NOR2 = new_definition
    (‘NOR2‘,
     "! a b out. nor2 a b out =
                (out = ~(a \/ b))");;

set_goal([], "! a b out. nor2_impl a b out = nor2 a b out");;
e(REWRITE_TAC [NOR2_IMPL; NOR2; OR2; NOT]);;
e(REPEAT STRIP_TAC);;
e(EQ_TAC THEN REPEAT STRIP_TAC);;
e(ASM_REWRITE_TAC []);;
e(EXISTS_TAC "a \/ b");;
e(ASM_REWRITE_TAC []);;
```

Now we will analyze how to implement general gates, that is gates without

a fixed number of input lines.

Their implementations are shown in figure 3.3. Their specification are given in the table below:

$$
\begin{aligned}
and(x, out) &\equiv \begin{cases} out & x = [\,] \\ (\exists c.and(b, c) \wedge (out \equiv a \wedge c)) & x = [a|b] \end{cases} \\
or(x, out) &\equiv \begin{cases} \neg out & x = [\,] \\ (\exists c.or(b, c) \wedge (out \equiv a \vee c)) & x = [a|b] \end{cases} \\
nand(a, out) &\equiv (out \equiv and(a, \neg out)) \\
nor(a, out) &\equiv (out \equiv or(a, \neg out)) \\
xor(x, out) &\equiv \begin{cases} \neg out & x = [\,] \\ (\exists c.xor(b, c) \wedge (out \equiv \neg(a \equiv c))) & x = [a|b] \end{cases}
\end{aligned}
$$

So let's begin to describe to HOL the first in these circuits: general AND gate.

A note on naming conventions: for an extensible input gate we adopt the convention to append an _N after the name of gate in the implementation, but before any extension.

Here is the declaration (using the list recursive constructor) of implementation and of specification of AND gate:

```
let AND_N_IMPL = new_list_rec_definition
    (‘AND_N_IMPL‘,
     "(and_n_impl NIL out = out) /\
      (!a b out.  and_n_impl (CONS a b) out =
              (?c. and2 a c out /\ and_n_impl b c))");;

let AND = new_list_rec_definition
    (‘AND‘,
     "(and NIL out = out) /\
      (!a b. and (CONS a b) out =
             (?c. (out = (a /\ c)) /\
                  and b c))");;
```

The goal is the usual one:

```
# set_goal([], "!a out. and_n_impl a out = and a out");;

"!a out. and_n_impl a out = and a out"
```

Because a has type list and our definitions are recursive on lists, the best thing to do is to develop inductively the definitions along goal. So we need to produce a proof by induction on lists:

```
# e(LIST_INDUCT_TAC);;

2 subgoals
"and__impl(CONS h a)out = and(CONS h a)out"
  1  ["!out. and__impl a out = and a out" ]

"and__impl[]out = and[]out"
```

The second subgoal is resolved by rewriting with definitions when lists are equal to $[]$:

```
# e(REWRITE_TAC [AND_N_IMPL; AND]);;

goal proved
|- and_n_impl[]out = and[]out

Previous subproof:
"and_n_impl(CONS h a)out = and(CONS h a)out"
  1  ["!out. and_n_impl a out = and a out" ]
```

The remaining subgoal is similar to the former: we could rewrite it with inductive definitions, with definition of AND2, which will appear after the preceding rewriting, and with assumption. We obtain:

```
# e(ASM_REWRITE_TAC [AND_N_IMPL; AND; AND2]);;

goal proved
. |- and__impl(CONS h a)out = and(CONS h a)out
|- !a out. and__impl a out = and a out

Previous subproof:
goal proved
```

OR and XOR gates have a similar structure, and, because of this, a similar proving technique.

The OR case is here:

```
let OR_N_IMPL = new_list_rec_definition
    (‘OR_N_IMPL‘,
     "(or_n_impl NIL out = ~out) /\
      (!a b.  or_n_impl (CONS a b) out =
              (?c. or2 a c out /\ or_n_impl b c))");;
```

```
let OR = new_list_rec_definition
    (‘OR‘,
     "(or NIL out = ~out) /\
      (!a b. or (CONS a b) out =
             (?c. (out = (a \/ c)) /\ or b c))");;

set_goal([], "!a out. or_n_impl a out = or a out");;
e(LIST_INDUCT_TAC THEN (REPEAT STRIP_TAC));;
e(REWRITE_TAC [OR_N_IMPL; OR]);;
e(ASM_REWRITE_TAC [OR_N_IMPL; OR; OR2]);;
```

And here is the XOR case:

```
let XOR_N_IMPL = new_list_rec_definition
    (‘xXOR_N_IMPL‘,
     "(xor_n_impl NIL out = ~out) /\
      (!a b.  xor_n_impl (CONS a b) out =
              (?c. xor2 a c out /\ xor_n_impl b c))");;

let XOR = new_list_rec_definition
    (‘XOR‘,
     "(xor NIL out = ~out) /\
      (!a b. xor (CONS a b) out =
             (?c. (out = ~(a = c)) /\
          xor b c))");;

set_goal([], "!a out. xor_n_impl a out = xor a out");;
e(LIST_INDUCT_TAC THEN (REPEAT STRIP_TAC));;
e(REWRITE_TAC [XOR_N_IMPL; XOR]);;
e(ASM_REWRITE_TAC [XOR_N_IMPL; XOR; XOR2]);;
```

The definitions of NOR gate are:

```
let NOR_N_IMPL = new_definition
    (‘nor_n_impl‘,
     "!a out. nor_n_impl a out =
     (?c. or a c /\ not c out)");;

let NOR = new_definition
    (‘NOR‘,
     "!a out. nor a out = or a ~out");;
```

The proof is quite standard in its development: we setup an equivalence goal; we rewrite with definitions then we strip the result; we break the equivalence

with EQ_TAC, then, by rewriting with assumptions, and instantiating with the obvious terms existential quantifiers, the proof came out.

```
set_goal([], "!a out. nor_n_impl a out = nor a out");;
e(REWRITE_TAC [NOR_N_IMPL; NOR; NOT]);;
e(REPEAT STRIP_TAC);;
e(EQ_TAC THEN STRIP_TAC);;
e(ASM_REWRITE_TAC []);;
e(EXISTS_TAC "~out");;
e(ASM_REWRITE_TAC []);;
```

The correctness proof for NAND gate is exactly the same as for NOR one:

```
let NAND_N_IMPL = new_definition
    (`NAND_N_IMPL`,
     "!a out. nand_n_impl a out =
     (?c. and a c /\ not c out)");;

let NAND = new_definition
    (`NAND`,
     "!a out. nand a out = and a ~out");;

set_goal([], "!a out. nand_n_impl a out = nand a out");;
e(REWRITE_TAC [NAND_N_IMPL; NAND; NOT]);;
e(REPEAT STRIP_TAC);;
e(EQ_TAC THEN STRIP_TAC);;
e(ASM_REWRITE_TAC []);;
e(EXISTS_TAC "~out");;
e(ASM_REWRITE_TAC []);;
```

Now is time to collect all these result about gates, and to build a library which contains definitions of both implementation and specification of all gates, along with their correctness proof.

In order to achieve this result, we need:

- to pack all our proofs
- to define a file containing the code to generate the theory
- to define a file containing the loading code

Because we have already developed all our proofs along with their definitions, we begin defining the file which will generate our theory.

The first thing to do, is to load the taut library, because we need the TAUT_TAC. Then we need to declare a new theory; its name will be gates. Here are the commands:

```
load_library `taut`;;

new_theory `gates`;;
```

Then we must declare our first definition, i.e. NAND2 gate, our base:

```
let NAND2 = new_definition
    (`NAND2`,
     "!a b out. nand2 a b out =
                (out = ~(a /\ b))");;
```

The next thing to do is to define our binary gates, along with their proofs. We begin with NOT gate:

```
let NOT_IMPL = new_definition
    (`NOT_IMPL`,
     "! a out. not_impl a out =
                nand2 T a out");;

let NOT = new_definition
     (`NOT`,
      "! a out. not a out =
                (out = ~a)");;

let NOT_PROOF = prove_thm(`NOT_PROOF`,
        "! a out. not a out = not_impl a out",
        (REWRITE_TAC [NOT_IMPL; NOT; NAND2]));;
```

Now let's see how to declare AND2 gate:

```
let AND2_IMPL = new_definition
    (`AND2_IMPL`,
     "! a b out. and2_impl a b out =
                 ?c. (nand2 a b c /\ not c out)");;

let AND2 = new_definition
    (`AND2`,
     "! a b out. and2 a b out =
                 (out = a /\ b)");;
```

```
let AND2_PROOF = prove_thm(‘AND_PROOF‘,
        "! a b out. and2 a b out = and2_impl a b out",
        (REWRITE_TAC [AND2_IMPL; AND2; NOT; NAND2])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN REPEAT STRIP_TAC)
THEN    REPEAT (EXISTS_TAC "~(a /\ b)")
THEN    (ASM_REWRITE_TAC []));;
```

Note the way we have packaged the proof: our procedure is rewrite definitions, then strip everything, then split equality and strip the results, then apply on each goal, as much as you can, the existencial quantifier substitution, and then rewrite with assumptions. The REPEAT(EXISTS_TAC ...) statement is a trick: the REPEAT tactical apply its argument as many times as it can, and, in our case, that means exactly one time to the goal which begins with $\exists$, while, on the other subgoal, the EXISTS_TAC fails immediatly, so REPEAT apply such a tactic to that subgoal exactly zero times.

With these ideas we can quickly develop the rest of the file:

```
let OR2_IMPL = new_definition
    (‘OR2_IMPL‘,
     "!a b out. or2_impl a b out =
        ? d e. (not a d /\ not b e /\ nand2 d e out)");;

let OR2 = new_definition
    (‘OR2‘,
     "!a b out. or2 a b out =
                (out = a \/ b)");;

let OR2_PROOF = prove_thm(‘OR2_PROOF‘,
        "! a b out. or2_impl a b out = or2 a b out",
        (REWRITE_TAC [OR2_IMPL; OR2; NOT; NAND2])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN (REPEAT STRIP_TAC))
THEN    REPEAT (     (EXISTS_TAC "~a")
                THEN (EXISTS_TAC "~b"))
THEN    (ASM_REWRITE_TAC [] THEN TAUT_TAC));;


let NOR2_IMPL = new_definition
    (‘NOR2_IMPL‘,
     "!a b out. nor2_impl a b out =
        ?c. (or2 a b c /\ not c out)");;
```

```
let NOR2 = new_definition
    (‘NOR2‘,
     "! a b out. nor2 a b out = (out = ~(a \/ b))");;

let NOR2_PROOF = prove_thm(‘NOR2_PROOF‘,
        "! a b out. nor2_impl a b out = nor2 a b out",
        (REWRITE_TAC [NOR2_IMPL; NOR2; OR2; NOT])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN (REPEAT STRIP_TAC))
THEN    REPEAT (EXISTS_TAC "a \/ b")
THEN    (ASM_REWRITE_TAC []));;


let XOR2_IMPL = new_definition
    (‘XOR2_IMPL‘,
     "!a b out. xor2_impl a b out =
        ?p q r. nand2 a b p /\ nand2 a p q /\
                nand2 p b r /\ nand2 q r out");;

let XOR2 = new_definition
    (‘XOR2‘,
     "! (a:bool) b out. xor2 a b out = (out = ~(a = b))");;

let XOR2_PROOF = prove_thm(‘XOR2_PROOF‘,
        "! a b out. xor2_impl a b out = xor2 a b out",
        (REWRITE_TAC [XOR2_IMPL; XOR2; NAND2])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN (REPEAT STRIP_TAC))
THEN    REPEAT (     (EXISTS_TAC "~(a /\ b)")
                THEN (EXISTS_TAC "~(a /\ ~(a /\ b))")
                THEN (EXISTS_TAC "~(~(a /\ b) /\ b)"))
THEN    ((ASM_REWRITE_TAC []) THEN TAUT_TAC));;


let AND_N_IMPL = new_list_rec_definition
    (‘AND_N_IMPL‘,
     "(and__impl NIL out = out) /\
      (!a b out.  and__impl (CONS a b) out =
              (?c. and2 a c out /\ and__impl b c))");;

let AND_N = new_list_rec_definition
    (‘AND_N‘,
     "(and NIL out = out) /\
      (!a b. and (CONS a b) out =
             (?c. (out = (a /\ c)) /\ and b c))");;
```

```
let AND_N_PROOF = prove_thm(‘AND_N_PROOF‘,
        "!a out. and__impl a out = and a out",
        (LIST_INDUCT_TAC THEN (REPEAT STRIP_TAC))
THEN    (ASM_REWRITE_TAC [AND_N_IMPL; AND_N; AND2]));;


let OR_N_IMPL = new_list_rec_definition
    (‘OR_N_IMPL‘,
     "(or__impl NIL out = ~out) /\
      (!a b.  or__impl (CONS a b) out =
              (?c. or2 a c out /\ or__impl b c))");;

let OR_N = new_list_rec_definition
    (‘OR_N‘,
     "(or NIL out = ~out) /\
      (!a b. or (CONS a b) out =
             (?c. (out = (a \/ c)) /\ or b c))");;

let OR_N_PROOF = prove_thm(‘OR_N_PROOF‘,
        "!a out. or__impl a out = or a out",
        (LIST_INDUCT_TAC THEN (REPEAT STRIP_TAC))
THEN    (ASM_REWRITE_TAC [OR_N_IMPL; OR_N; OR2]));;


let NOR_N_IMPL = new_definition
    (‘NOR_N_IMPL‘,
     "!a out. nor__impl a out = (?c. or a c /\ not c out)");;

let NOR_N = new_definition
    (‘NOR_N‘,
     "!a out. nor a out = or a ~out");;

let NOR_N_PROOF = prove_thm(‘NOR_N‘,
        "!a out. nor__impl a out = nor a out",
        (REWRITE_TAC [NOR_N_IMPL; NOR_N; NOT])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN STRIP_TAC)
THEN    REPEAT (EXISTS_TAC "~out")
THEN    (ASM_REWRITE_TAC []));;


let NAND_N_IMPL = new_definition
    (‘NAND_N_IMPL‘,
     "!a out. nand__impl a out =
             (?c. and a c /\ not c out)");;
```

```
let NAND_N = new_definition
    (‘NAND_N‘,
     "!a out. nand a out = and a ~out");;

let NAND_N_PROOF = prove_thm(‘NAND_N_PROOF‘,
        "!a out. nand__impl a out = nand a out",
        (REWRITE_TAC [NAND_N_IMPL; NAND_N; NOT])
THEN    (REPEAT STRIP_TAC)
THEN    (EQ_TAC THEN STRIP_TAC)
THEN    REPEAT (EXISTS_TAC "~out")
THEN    (ASM_REWRITE_TAC []));;


let XOR_N_IMPL = new_list_rec_definition
    (‘XOR_N_IMPL‘,
     "(xor__impl NIL out = ~out) /\
      (!a b.  xor__impl (CONS a b) out =
              (?c. xor2 a c out /\ xor__impl b c))");;

let XOR_N = new_list_rec_definition
    (‘XOR_N‘,
     "(xor NIL out = ~out) /\
      (!a b. xor (CONS a b) out =
             (?c. (out = ~(a = c)) /\
                  xor b c))");;

let XOR_N_PROOF = prove_thm(‘XOR_N_PROOF‘,
        "!a out. xor__impl a out = xor a out",
        (LIST_INDUCT_TAC THEN (REPEAT STRIP_TAC))
THEN    (ASM_REWRITE_TAC [XOR_N_IMPL; XOR_N; XOR2]));;


close_theory();;
```

The previous file, when loaded by HOL, will generate a theory, *gates* containing all definitions and proofs we have developed so far.

What we need now, and it is the last thing to do, in order to complete our library, is to write a loading program, which loads our theory, enable the use of its symbols, and setup the environment properly.

What the loading file must do is:

- Setup the environment. In our case, this step is limited to verify that we are in draft mode. We require this mode because our theory is useful only when we design a new component, and, with HOL, designing means defining, and this can be done only in draft mode.

- Load the needed libraries. In our case, we have to load taut library, because we need to prove propositional theorems.
- We need to setup paths. Particulary we need to setup library path and help search path, because we want to write help files for elements of our theory (we will not do this here, but, if we write a general theory, is a good thing to provide help).
- We need to load our theory. In our case, we declare that our theory is an ancestor of the currently developed theory, and we declare that all our definitions and theorems have to be loaded on demand.

The result of such an effort is the following file, which name is `gates.ml`, and which resides in the `gates` directory, in the standard libraries search path, along with `gates.th`, generated by the previous file. The listing of `gates.ml` is:

```
if not (draft_mode()) then
  failwith `HOL must be in draft mode to load this library.`;;

load_library `taut`;;

let path = library_pathname() ^ `/gates/`
in  print_string `Updating search paths`; print_newline();
    set_search_path(union (search_path()) [path]);
    set_help_search_path (union [path ^ `help/entries/`]
                                (help_search_path ()));;

print_string `Declaring theory gates to be a new parent`;;
print_newline();;
let path = library_pathname() ^ `/gates/gates`
in  new_parent path;
    map (\x. autoload_theory(`definition`, path, fst x))
 (definitions path);
    map (\x. autoload_theory(`theorem`, path, fst x))
 (theorems path);;
```

3.1.4 Some Considerations

To end this section we want to make some considerations.

The work of defining gates, and proving their correctness is long, but not difficult. Almost all proofs have the same structure, that is rewriting the definitions, stripping everything, splitting equality, stripping again, and then substituting existential quantifiers with terms we find in the goal, and finally, rewriting with assumptions, eventually solving propositional tautologies. This schema is common to all examples we have seen since now.

To define a library is a long and tedious work, but is useful when we need to reuse our results in several contexts. We have seen the basic technique to begin the development of a complex library, but our example, though useful, is not difficult.

In the next section we will see how to extend our proof techniques to combinational circuits, and we will use results of this section to prove that more complex circuits are correct if basic gates are implemented in the proper way, but we know that this is possible, because we have proved it.

In the future we will not develop proofs in a such level of detail as we have done in this section, but we will point out the most important steps.

3.2 Combinational Circuits

Combinational circuits are defined having no state. From a practical point of view, this means that there are no memory elements, and the behaviour is functional, that is, the result depends only on inputs. Or, if you prefer, to the same input corresponds the same result every time.

These circuits have no clock, and time is not important in our treatment.

Techniques we will develop in this section are not very different from the one used to treat gates, but the power of formal methods begins to appear.

3.2.1 Representation

The basics to represent combinational circuits are the same as the one we have seen to represent gates.

An important concept is hierarchy of representations. From a teoretichal point of view, this is just an application of the homomorphic property. From a practical point of view, this means that, if we want to represent a complex circuit, we can divide it into a set of subcomponents, and we can represent them separately from the initial circuit, and, by componing them in the proper way, we can represent the initial circuit. This process can be iterated as needed.

We have two chances when we want to prove correctness of a hierarchy of components: we can work bottom-up, that is, we can prove correctness of the most elementar components, and then use them to prove correctness of more complex circuits, till to the top, which contains the initial circuit; we can work top-down, that is, we can prove correctness of each complex component assuming the simpler ones are correct, and then prove correctness of them.

Both choices have advantages and disadvantages. The bottom-up approach has the following characteristics:

- you build correct components from correct components
- this approach prevents that faults from different levels of abstraction can appear in the same component

while the top-down approach has the following characteristics:

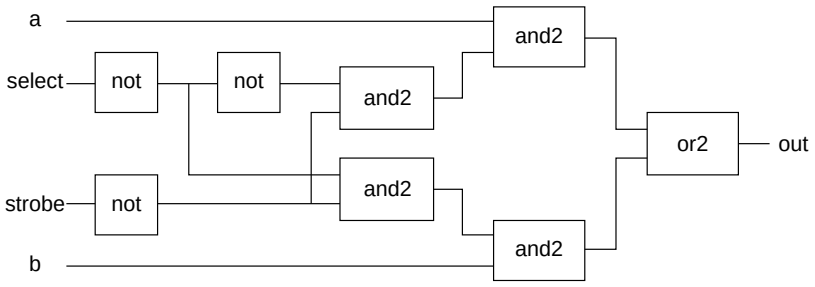


Figure 3.4: The SN74157: a 2 to 1 line data selector/multiplexer

- a proved correct components, with a faulty subcomponent preserve the validity of its initial project.
- major faults are located before minor ones

From these characteristics is clear that the bottom-up approach is preferred if formal verification takes place after, or concurrently, but separately from the project, while the top-down approach is better suited for situation where the project is verified during its production.

3.2.2 Example: A Simple Multiplexer

In this subsection we present the correctness proof of a simple multiplexer, you can see in figure 3.4. This circuit is interesting for two reasons: first, it is simple, but not too trivial to prove its correctness; second, this is a real circuit, the standard TTL SN74175.

Describing its implementation is very simple: we load *gates* library, then we use our standard method, and the result is here:

```
load_library `gates`;;

let SN74157_IMPL = new_definition
        (`SN74157_IMPL`,
         "! a b sel strobe out.
           sn74157_impl a b sel strobe out =
            ? t u v w x y z.    not sel x       /\
                                not x y         /\
                                not strobe z    /\
                                and2 y z v      /\
                                and2 x z w      /\
                                and2 a w u      /\
                                and2 b v t      /\
                                or2 u t out        ");;
```

This circuit is a multiplexer with a selection line and a strobe line. Strobe line is used to force output to false. We can express this fact with a statement like:

$$\mathit{if}\ \mathrm{strobe}\ \mathit{then}\ \mathrm{out} = \mathit{false}\ \mathit{else} \ldots$$

With our logic syntax, we can use the conditional operator:

$$\mathrm{strobe} \rightarrow \neg\mathrm{out} \mid \ldots$$

Selection line is used to make output equal to the first or to the second input line. If it is true then output is equal to b, otherwise it is equal to a, but the selection process take place if and only if strobe line is low.

We can formalize such a specification as:

```
let SN74157 = new_definition
        (‘SN74157‘,
         "! a b sel strobe out. sn74157 a b sel strobe out =
              strobe => ~out |
                        (sel => (out = b) | (out = a))");;
```

Our goal will state that all the possible behaviours of the circuit are correct. In other words, we state that the implementation implies the specification.

```
set_goal([],"! a b sel strobe out.
                sn74157_impl a b sel strobe out ==>
                                 sn74157 a b sel strobe out");;
```

To prove this goal, we try to use the standard technique: we begin rewriting definitions, and stripping the result.

```
# e(REWRITE_TAC [SN74157_IMPL; SN74157]);;
# e(REWRITE_TAC [NOT; AND2; OR2]);;
# e(REPEAT STRIP_TAC);;

"(strobe => ~out | (sel => (out = b) | (out = a)))"
  8  ["x = ~sel" ]
  7  ["y = ~x" ]
  6  ["z = ~strobe" ]
  5  ["v = y /\ z" ]
  4  ["w = x /\ z" ]
  3  ["u = a /\ w" ]
  2  ["t = b /\ v" ]
  1  ["out = u \/ t" ]
```

The outermost operator, i.e. the principal functor of this expression is the conditional operator. The meaning of the goal is: if $strobe$ is true then the output is forced to false, otherwise the output is equal to one of the inputs, depending on the selection line.

Let's split possibilities, expanding the conditional operator:

```
# e(COND_CASES_TAC);;

2 subgoals
"(sel => (out = b) | (out = a))"
  ...
  1  ["~strobe" ]

"~out"
  9  ["x = ~sel" ]
  8  ["y = ~x" ]
  7  ["z = ~strobe" ]
  6  ["v = y /\ z" ]
  5  ["w = x /\ z" ]
  4  ["u = a /\ w" ]
  3  ["t = b /\ v" ]
  2  ["out = u \/ t" ]
  1  ["strobe" ]
```

The former subgoal corresponds to $strobe = false$, the latter one, that is the current goal, corresponds to $strobe = true$.

Looking at assumptions, we note that the goal can be proved only by rewriting:

```
# e(ASM_REWRITE_TAC []);;

goal proved
......... |- ~out

Previous subproof:
"(sel => (out = b) | (out = a))"
  9  ["x = ~sel" ]
  8  ["y = ~x" ]
  7  ["z = ~strobe" ]
  6  ["v = y /\ z" ]
  5  ["w = x /\ z" ]
  4  ["u = a /\ w" ]
  3  ["t = b /\ v" ]
  2  ["out = u \/ t" ]
  1  ["~strobe" ]
```

Again we have to alternatives: *sel* can be *true* or *false*. Let's split again:

```
# e(COND_CASES_TAC);;

2 subgoals
"out = a"
  10 ["x = ~sel" ]
  9  ["y = ~x" ]
  8  ["z = ~strobe" ]
  7  ["v = y /\ z" ]
  6  ["w = x /\ z" ]
  5  ["u = a /\ w" ]
  4  ["t = b /\ v" ]
  3  ["out = u \/ t" ]
  2  ["~strobe" ]
  1  ["~sel" ]

"out = b"
  10 ["x = ~sel" ]
  9  ["y = ~x" ]
  8  ["z = ~strobe" ]
  7  ["v = y /\ z" ]
  6  ["w = x /\ z" ]
  5  ["u = a /\ w" ]
  4  ["t = b /\ v" ]
  3  ["out = u \/ t" ]
  2  ["~strobe" ]
  1  ["sel" ]
```

Again, if we look closer at assumptions, we see that both goals could be solved by rewriting.

```
# e(ASM_REWRITE_TAC []);;
# e(ASM_REWRITE_TAC []);;

...
Previous subproof:
goal proved
```

3.2.3 Example: A Complete Adder

To develop a complete adder we need an half adder. Let's use the one shown in figure 3.5. We want to prove its correctness, and then use this result to prove correctness of the complete adder, shown in figure 3.6.

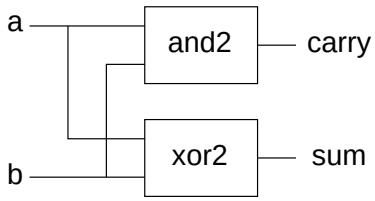


Figure 3.5: The half adder

We need two libraries in order to reason on half adder:

- the *gates* library, because our implementation uses basic gates.
- the *arith* library, because our specification will use an arithmetical argument.

So, the first thing to do in the development of this example is:

```
load_library `gates`;;
load_library `arith`;;
```

To simplify application of arithmetical theorems and definitions, we define a new tactic, which applies recursively arithmetical conversions through a goal. It will apply, descending the goal in a depth first fashion, the conversions defined by ARITH_CONV:

```
let DEPTH_ARITH_TAC = CONV_TAC (ONCE_DEPTH_CONV ARITH_CONV);;
```

We need a way to convert boolean values to integers, with the standard map: *false* is 0, and *true* is 1. The conversion function is:

```
let B2I = new_definition
        (`B2I`,
         "! a. b2i a = (a => SUC 0 | 0)");;
```

After these preliminaries, we are ready to approach the half adder. The first task is to define implementation:

```
let HALF_ADDER = new_definition
        (`HALF_ADDER`,
         "! a b c s. half_adder a b c s =
                        and2 a b c /\ xor2 a b s ");;
```

We diverged from our conventions on naming, because, in this case, implementation is simpler than specification, so we give the principal name, the one without extensions, to the predicate representing the circuit, and to its definition. We will add the _SPEC suffix to specification's definition, and _*spec* to its predicate.

Our specification have to say that the sum of a and b, is equal to *sum* plus $2 * carry$. Applying the conversion from booleans to integers, our definition becomes:

```
let HALF_ADDER_SPEC = new_definition
        (‘HALF_ADDER_SPEC‘,
         "! a b c s. half_adder_spec a b c s =
                       b2i a + b2i b = (2 * b2i c) + b2i s");;
```

Our proof begins setting the goal in the implicational form, and then rewrting definitions:

```
# set_goal([],"!a b c s. half_adder a b c s ==>
                           half_adder_spec a b c s");;
# e(REWRITE_TAC [HALF_ADDER_SPEC; HALF_ADDER;
                 AND2; XOR2; B2I]);;

"!a b c s.
  (c = a /\ b) /\ (s = ~(a = b)) ==>
  ((a => SUC 0 | 0) + (b => SUC 0 | 0) =
   (2 * (c => SUC 0 | 0)) + (s => SUC 0 | 0))"
```

The next step is to put numbers in canonical form, that is 0 or SUC of another number in canonical form, and then stripping the result:

```
# e(REWRITE_TAC [num_CONV "2"; num_CONV "1"]);;
# e(REPEAT STRIP_TAC);;

"(a => SUC 0 | 0) +
  (b => SUC 0 | 0) =
 ((SUC(SUC 0)) * (c => SUC 0 | 0)) +
  (s => SUC 0 | 0)"
  2  ["c = a /\ b" ]
  1  ["s = ~(a = b)" ]
```

Now, we can use assumptions to throw away c and s from the goal:

```
# e(ASM_REWRITE_TAC []);;

"(a => SUC 0 | 0) + (b => SUC 0 | 0) =
 ((SUC(SUC 0)) * ((a /\ b) => SUC 0 | 0)) +
  ((~(a = b)) => SUC 0 | 0)"
  2  ["c = a /\ b" ]
  1  ["s = ~(a = b)" ]
```

There are two possibilities for a: it can be *true* or *false*. Let's examine them with COND_CASES_TAC:

```
# e(COND_CASES_TAC);;

2 subgoals
"0 + (b => SUC 0 | 0) =
 ((SUC(SUC 0)) * ((F /\ b) => SUC 0 | 0)) +
  ((~(F = b)) => SUC 0 | 0)"
  3  ["c = a /\ b" ]
  2  ["s = ~(a = b)" ]
  1  ["~a" ]

"(SUC 0) + (b => SUC 0 | 0) =
 ((SUC(SUC 0)) * ((T /\ b) => SUC 0 | 0)) +
  ((~(T = b)) => SUC 0 | 0)"
  3  ["c = a /\ b" ]
  2  ["s = ~(a = b)" ]
  1  ["a" ]
```

The two subgoals are very similar: if we fix the value of b, then the entire expression becomes a pure arithmetic equality, one which could be decided by computing values of arithmetic subexpressions. So we will analyze both cases of the possible values of b, simplifing arithmetic expressions. Doing so on both subgoals suffice to prove both of them:

```
# e(COND_CASES_TAC THEN DEPTH_ARITH_TAC);;
# e(COND_CASES_TAC THEN DEPTH_ARITH_TAC);;

...
Previous subproof:
 Goal proved
```

The circuit implementing full adder is shown in figure 3.6. Because complete adder's implementation is simpler than its specification, we will give to implementation's predicate and definition's the principal name. Here follow definitions:

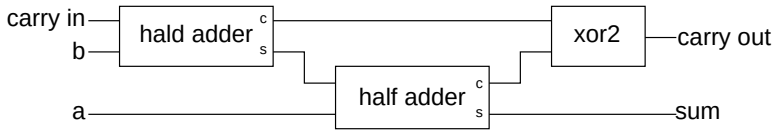


Figure 3.6: The complete adder

```
let FULL_ADDER = new_definition
        (‘FULL_ADDER‘,
         "! a b cin sum cout. full_adder a b cin sum cout =
                  ? x y z. half_adder b cin x y   /\
                           half_adder a y z sum   /\
                           xor2 z x cout");;
```

The specification says that the sum of the three inputs must be equal to the binary number whose least significant digit is sum and whose most significant digit is $carry\ out$. Formally:

```
let FULL_ADDER_SPEC = new_definition
        (‘FULL_ADDER_SPEC‘,
         "! a b cin sum cout.
                full_adder_spec a b cin sum cout =
                  ( b2i a + b2i b + b2i cin =
                    b2i sum + 2 * b2i cout)");;
```

The goal has the implicational form. We begin rewriting definitions, and putting numbers in the canonical form, and then stripping to simplify the goal:

```
# set_goal([],"! a b cin sum cout.
                full_adder a b cin sum cout ==>
                  full_adder_spec a b cin sum cout");;
# e(REWRITE_TAC [FULL_ADDER_SPEC; FULL_ADDER]);;
# e(REWRITE_TAC [HALF_ADDER; XOR2; AND2; B2I]);;
# e(REWRITE_TAC [num_CONV "2"; num_CONV "1"]);;
# e(REPEAT STRIP_TAC);;

"(a => SUC 0 | 0) + ((b => SUC 0 | 0) + (cin => SUC 0 | 0)) =
 (sum => SUC 0 | 0) + ((SUC(SUC 0)) * (cout => SUC 0 | 0))"
  5  ["x = b /\ cin" ]
  4  ["y = ~(b = cin)" ]
  3  ["z = a /\ y" ]
  2  ["sum = ~(a = y)" ]
  1  ["cout = ~(z = x)" ]
```

The value of a line is crucial when computing the sum, because it touches the half adder near the end of the circuit. We want to throw away the conditional on a line: the technique is the usual one, COND_CASE_TAC:

```
# e(COND_CASES_TAC);;

2 subgoals
"0 + ((b => SUC 0 | 0) + (cin => SUC 0 | 0)) =
 (sum => SUC 0 | 0) + ((SUC(SUC 0)) * (cout => SUC 0 | 0))"
  ...
  1  ["~a" ]

"(SUC 0) + ((b => SUC 0 | 0) + (cin => SUC 0 | 0)) =
 (sum => SUC 0 | 0) + ((SUC(SUC 0)) * (cout => SUC 0 | 0))"
  6  ["x = b /\ cin" ]
  5  ["y = ~(b = cin)" ]
  4  ["z = a /\ y" ]
  3  ["sum = ~(a = y)" ]
  2  ["cout = ~(z = x)" ]
  1  ["a" ]
```

Let's reason: if we analyze the two possibilities for b, then we analyze the two possibilities for cin, we have a goal that, rewritten by assumptions, will be purely arithmetical, so can be solved directly by our tactic. In HOL syntax this means:

```
# e(COND_CASES_TAC);;
# e(     COND_CASES_TAC
    THEN ASM_REWRITE_TAC []
    THEN DEPTH_ARITH_TAC);;
# e(     COND_CASES_TAC
    THEN ASM_REWRITE_TAC []
    THEN DEPTH_ARITH_TAC);;

goal proved
...
Previous subproof:
"0 + ((b => SUC 0 | 0) + (cin => SUC 0 | 0)) =
 (sum => SUC 0 | 0) + ((SUC(SUC 0)) * (cout => SUC 0 | 0))"
  6  ["x = b /\ cin" ]
  5  ["y = ~(b = cin)" ]
  4  ["z = a /\ y" ]
  3  ["sum = ~(a = y)" ]
  2  ["cout = ~(z = x)" ]
  1  ["~a" ]
```

The current goal, the last one, has exactly the same logical structure of the previous one, and, trying to apply the same reasoning, we can complete the proof:

```
# e(COND_CASES_TAC);;
# e(     COND_CASES_TAC
    THEN ASM_REWRITE_TAC []
    THEN DEPTH_ARITH_TAC);;
# e(     COND_CASES_TAC
    THEN ASM_REWRITE_TAC []
    THEN DEPTH_ARITH_TAC);;

...
Previous subproof:
goal proved
```

3.2.4 Developing a Proof

We want to end this chapter we some considerations on how to develop a correctness proof. If you look again at the examples we presented, you will notice that proofs are carried out always with the same pattern:

- Rewrite the goal with definitions, and, if needed, put things, like numbers, in a canonical form.
- Simplify the goal(s) using destructor rules for the principal functor.
- When a goal is simple enough, try to solve it by rewriting with assumptions.
- Now the resulting goal is a closed formula, so apply the appropriate decison procedure: TAUT_TAC for propositional tautologies, DEPTH_ARITH_TAC for arithmetic expressions, etc.

Let's examine each step from another point of view.

The first step, rewriting with definitions, is the way we use to transform a represented circuit, or specification, into a purely logical formula. A logical formula is a pure mathematical object, it has no relation with any real entity.

The second step, application of inference rules on a "remove the principal functor" basis, is the natural way to prove things in formal logic. Thanks to normalization theorem, every formula that doesn't contain an hard existential quantifier can be proved using the heuristic "destroy the principal functor". The way we describe circuits as logic formulae is set up so that, with our proof pattern, every existential quantifier can be solved only looking with sufficient attention to the goal itself.

The third step has to eliminate dependence of the goal from the assumptions, in order to prepare the fourth step.

The fourth step is based on the idea that a lot of mathematical domains, such as simple arithmetic, propositional calculus, and so on, can be decided with a completely automatic procedure.

In the next chapter we will study sequential circuits, but the basic schema to prove correctness will remain essentially the same.

Chapter 4

Clocked Sequential Circuits

In this chapter we speak about time in circuits with memory elements. We want to capture the notion of state, clock and memory cell.

4.1 Representing Time

Let's begin with the central notion of time. Standard circuits are synchronous, that means, they have a common *time generator*. It is the clock signal.

In this section we want to represent the clock. To achieve this result, we need to state some hypothesis on the nature of our time concept. Then we will begin to show methods to represent time in our circuit descriptions.

4.1.1 Hypothesis

Our first hypothesis is that time is a discrete entity. We state that our time is not the physical one, which is a continuous variable, but it is a discrete set of ordered observations of the continuous behaviour on a timed circuit.

The second hypothesis is that our time points are not arbitrary, but are those ones in which components have valid input or output signals, and the general state of the circuit is stable.

To meet these hypothesis is simple, because the clock signal is exactly the one which synchronizes every component.

We will not state that an observation take place when a clock cycle will finish, nor we state that there is an unique, global clock, but we state that exists an unique, global time, with our requirements, and that all clock signals are in relation with it in a simple, discrete way.

Our hypothesis means that the global time of a circuit can be represented as a natural number, where 0 means the begin of inizialitation time.

We also say that an instant A is immediately following another instant B if and only if $A = B + 1$. To be very formal we will use the Peano's successor function SUC which stands for $+1$.

4.1.2 Clock Abstraction

To represent the clock signal, we can follow two ways:

- we can represent the clock line in the usual way, and we develop a component which generates the clock signal
- we can choose not to represent the clock at all.

In both cases, the problem is time: in a sequential circuit, outputs depends on actual and preceding inputs, on the "history" of inputs, if you prefer.

The solution to this problem is quite simple, but very powerful: we can represent wires not as boolean valued variables, but as signal valued variables. And a signal is a function that produces the (boolean) value on the line, given a time instant. Formally:

$$Signals = Time \longrightarrow Booleans$$

Let's analyze the two possibilities we have to represent the clock signal.

We can represent lines as signal variables, and we can represent the clock as a device which toggles its output at every instant:

$$\text{clock out} = (\forall t.\text{out } t = \neg \text{out}(t+1) \wedge (\text{out } 0 = \text{true})$$

This means that at every instant the clock is high or low, so our time is half as long as a clock cycle.

On the other hand we can choose not the represent the clock. In this case, all lines are signal variables, and an instant is equal to the shortest time period that is significant in the circuit. Every component has a time-dependent behaviour and, when we have to prove properties of a circuit, we will consider all possible cases of the clock values in every instant.

The second possibility, in general, is simpler, so we will adopt it. But remember that exists the other solution, and try to understand if, in a particular case, it is better than to abstract the clock.

4.1.3 General Time Representation

To represent time on lines we defined the signal domain; to represent time in components, we need to define the properties time satisfies in order to represent it using HOL.

We want that the $Time$ domain has the following properties:

- It is totally ordered
- It is discrete
- It is not limited
- There exists an initial instant
- Given an instant, the next is fully determined

The most natural choice to represent time is by means of natural numbers:

- They are ordered, discrete and infinite
- The initial instant is called 0
- The instant next to x is $\mathrm{SUC}\ x$, that is $x + 1$

Sometimes is useful to adopt a slightly different convention: sometimes is better not to set the initial instant, but to code the initialization behaviour in the representation of circuits in the usual way.

4.2 Flip-Flops

In the previous section we have seen a bit of theory behind representation of sequential circuits. The most basic memory element is the flip-flop. It has a state of exactly one bit, and this state can be forced to a value, or simply can be read. Learning how to specify simple states (as the flip-flop's ones) is the goal of this section.

4.2.1 Basic Flip-Flops

We need to define a basic memory element, one that is so simple not to require a correctness proof, one that is so widely used that every electronic engineer knows how it works, one that is so powerful to let us build every other memory circuit.

There is an answer to these question: we need a flip-flop. But there are a lot of kinds of flip-flops. And we want to choose a minimal set.

As a practical consideration, is simpler to deal with a flip-flop with no invalid input configurations, so we discard S-R flip-flops. From another point of view, D flip-flops are the simplest ones, but it is difficult, and impractical to build all other kinds of flip-flops with a D type.

So we choose a J-K flip-flop as our base. An S-R flip-flop is exactly equivalent, with the Set line renamed to J, and the Reset line renamed to K.

The truth table of a J-K flip-flop is,

J	K	Q	out
0	0	X	X
0	1	X	0
1	0	X	1
1	1	X	$\neg$X

where Q is the current state, and out is the output and the next state.

We give four definitions for a J-K flip-flop:

- initialization at time 0 with value $false$
- initialization at time 0 with value $true$

- with a clear line
- with a preset line

Our naming convention will be:

1. The first letters will identify the flip-flop type: e.g. JK, SR.
2. If the flip-flop is resettable at any time, then we append a T to the basic name.
3. If there is only one possible inizialization value for the flip-flop, then we append it to the name
4. The name ends with a _FF.

This will be the naming schema for flip-flop components: all other naming rules will apply to the so generated names.

Now let's see definition of the choosed flip-flops in HOL:

```
let JK0_FF = new_definition
        (`JK0_FF`,
         "! j k out. jk0_ff j k out = ~out 0 /\
             ! t. ( j t /\  k t ==> (out (SUC t) = ~out t)) /\
                  (~j t /\ ~k t ==> (out (SUC t) =  out t)) /\
                  ( j t /\ ~k t ==>  out (SUC t)) /\
                  (~j t /\  k t ==> ~out (SUC t))");;

let JK1_FF = new_definition
        (`JK1_FF`,
         "! j k out. jk1_ff j k out =  out 0 /\
             ! t. ( j t /\  k t ==> (out (SUC t) = ~out t)) /\
                  (~j t /\ ~k t ==> (out (SUC t) =  out t)) /\
                  ( j t /\ ~k t ==>  out (SUC t)) /\
                  (~j t /\  k t ==> ~out (SUC t))");;

let JKT0_FF = new_definition
        (`JKT0_FF`,
         "! j k clear out. jkt0_ff j k clear out =
             ! (t:num). clear t => ~out(SUC t) |
                 (( j t /\  k t ==> (out (SUC t) = ~out t)) /\
                  (~j t /\ ~k t ==> (out (SUC t) =  out t)) /\
                  ( j t /\ ~k t ==>  out (SUC t)) /\
                  (~j t /\  k t ==> ~out (SUC t)))");;
```

```
let JKT1_FF = new_definition
        (‘JKT1_FF‘,
         "! j k preset out. jkt1_ff j k preset out =
             ! (t:num). preset t => out(SUC t) |
                 (( j t /\  k t ==> (out (SUC t) = ~out t)) /\
                  (~j t /\ ~k t ==> (out (SUC t) =  out t)) /\
                  ( j t /\ ~k t ==>  out (SUC t)) /\
                  (~j t /\  k t ==> ~out (SUC t)))");;
```

Exactly in the same way, we can define all other flip-flops. For example, we present the various specifications of the D flip-flop.

```
let D1_FF = new_definition
        (‘D1_FF‘,
         "! d out. d1_ff d out =
                (out 0 /\ (! t. out(SUC t) = d t))");;

let D0_FF = new_definition
        (‘D0_FF‘,
         "! d out. d0_ff d out =
                ((~ out 0) /\ (! t. out(SUC t) = d t))");;

let DT1_FF = new_definition
        (‘DT1_FF‘,
         "! d preset (out:num -> bool). dt1_ff d preset out =
               (! (t:num). preset t => out(SUC t) |
                                       out(SUC t) = d(t))");;

let DT0_FF = new_definition
        (‘DT0_FF‘,
         "! d clear (out:num -> bool). dt0_ff d clear out =
               (! (t:num). clear t => ~ out(SUC t) |
                                      out(SUC t) = d(t))");;
```

Now we want to prove that preinizialized D flip-flops are correct. We will present their implementations, and their correctness arguments.

But before these, we need to build some proving tools, because our proofs will be not elementar.

The very first thing we have to do, is to load the gates library.

```
load_library ‘gates‘;;
```

We want to reason on assumptions, that means, we want to deduce intermediate results from a given set of assumptions. To achieve this result in the simplest way, we need a method to name assumptions.

The goal is represented by an ordered pair, where the first element is a list of terms representing the assumptions. So a simple way to refer to a particular assumption is to give it a number, the one you see near the left border when the goal is printed. We write a function that, given a number, returns the corresponding assumption:

```
let assum n = (el n (fst(top_goal ())));;
```

We need such a function to develop two tactics: the former discharges an assumption, the latter discharges all assumption but one.

HOL can easily discharge one assumption by means of the UNDISCH_TAC, but it takes a term as an argument, and assumptions could be quite long, so we define UNDISCH_N_TAC that do the same work, but takes the assumption number as a parameter.

```
let UNDISCH_N_TAC n = UNDISCH_TAC (assum n);;
```

To discharge all assumptions except one, is tricky: we could perform a loop, discharging assumptions one by one, skipping the one which corresponds to the parameter.

This idea has a drawback: the user will see every step in the unmounting process. The solution is to build a tactic, which will do a UNDISCH_N_TAC for every assumption number except for the parameter, and, when the loop variable equals the parameter, then we produce the ALL_TAC, that is the identity tactic for THEN operator. The result is below:

```
let UNDISCH_BUT_TAC n = letref x = length (fst(top_goal ()))
                           and t = ALL_TAC
                        in while x > 0 do
                             x := (x = n =>
                                   t THEN ALL_TAC |
                                   t THEN (UNDISCH_N_TAC x)),
                                   x - 1 ; t;;
```

Representing the implementation is very simple: we follow the usual technique, paying attention that signals are functions from time to booleans. Here is the definition in HOL syntax of the two variants shown in figure 4.1

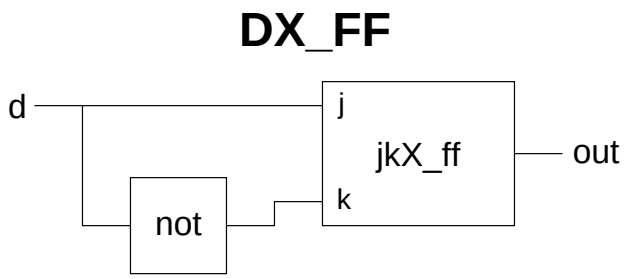


Figure 4.1: How to make a D flip-flop with a JK one

```
let D1_FF_IMPL = new_definition
        (‘D1_FF_IMPL‘,
         "! d out. d1_ff_impl d out =
                 (? x. !t. ((not (d t) (x t)) /\
                           (jk1_ff d x out)))");;

let D0_FF_IMPL = new_definition
        (‘D0_FF_IMPL‘,
         "! d out. d0_ff_impl d out =
                 (? x. !t. ((not (d t) (x t)) /\
                           (jk0_ff d x out)))");;
```

The correctness argument and the first steps are the usual ones:

```
# set_goal([],"! d out. d1_ff_impl d out = d1_ff d out");;
# e(REWRITE_TAC [D1_FF_IMPL; D1_FF; NOT; JK1_FF]);;
# e(REPEAT STRIP_TAC);;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

3 subgoals
"?x.
  !t.
   (x t = ~d t) /\
   out 0 /\
   (!t.
     (d t /\ x t ==> (out(SUC t) = ~out t)) /\
     (~d t /\ ~x t ==> (out(SUC t) = out t)) /\
     (d t /\ ~x t ==> out(SUC t)) /\
     (~d t /\ x t ==> ~out(SUC t)))"
  2  ["out 0" ]
  1  ["!t. out(SUC t) = d t" ]
```

```
"out(SUC t) = d t"
  1  ["!t.
        (x t = ~d t) /\
        out 0 /\
        (!t.
          (d t /\ x t ==> (out(SUC t) = ~out t)) /\
          (~d t /\ ~x t ==> (out(SUC t) = out t)) /\
          (d t /\ ~x t ==> out(SUC t)) /\
          (~d t /\ x t ==> ~out(SUC t)))" ]

"out 0"
  1  ["!t.
        (x t = ~d t) /\
        out 0 /\
        (!t.
          (d t /\ x t ==> (out(SUC t) = ~out t)) /\
          (~d t /\ ~x t ==> (out(SUC t) = out t)) /\
          (d t /\ ~x t ==> out(SUC t)) /\
          (~d t /\ x t ==> ~out(SUC t)))" ]
```

The current goal is easy: if we throw away the universal quantification in the assumption, and we take the second conjunct, then we have exactly the goal formula, and then, by rewriting with assumption, the proof is done. In HOL syntax:

```
# e(RULE_ASSUM_TAC (CONJUNCT1 o CONJUNCT2 o GSPEC));;
# e(ASM_REWRITE_TAC []);;
```

To solve the second subgoal, we have to split the assumption. First we need to move out conjunctions through quantifications, then we have to strip the result. Because we have no rule to strip an assumption, we discharge it, then we apply to the resulting goal the usual STRIP_TAC. The result is:

```
# e(RULE_ASSUM_TAC (CONV_RULE
                    (TOP_DEPTH_CONV FORALL_AND_CONV)));;
# e((UNDISCH_BUT_TAC 0) THEN (REPEAT STRIP_TAC));;

"out(SUC t) = d t"
  6  ["!t. x t = ~d t" ]
  5  ["!t. out 0" ]
  4  ["!t t. d t /\ x t ==> (out(SUC t) = ~out t)" ]
  3  ["!t t. ~d t /\ ~x t ==> (out(SUC t) = out t)" ]
  2  ["!t t. d t /\ ~x t ==> out(SUC t)" ]
  1  ["!t t. ~d t /\ x t ==> ~out(SUC t)" ]
```

To solve this goal, we specialize all universal quantifiers:

```
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 6);;
# e(UNDISCH_N_TAC 5);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;

"out 0 ==> (x t = ~d t) ==> (out(SUC t) = d t)"
  4  ["d t /\ x t ==> (out(SUC t) = ~out t)" ]
  3  ["~d t /\ ~x t ==> (out(SUC t) = out t)" ]
  2  ["d t /\ ~x t ==> out(SUC t)" ]
  1  ["~d t /\ x t ==> ~out(SUC t)" ]
```

Now the solution is very easy: the goal is propositional, as all the assumptions. We discharge them and call the tautology checher:

```
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
...
Previous subproof:
"?x. !t. (x t = ~d t) /\ out 0 /\
   (!t.
     (d t /\ x t ==> (out(SUC t) = ~out t)) /\
     (~d t /\ ~x t ==> (out(SUC t) = out t)) /\
     (d t /\ ~x t ==> out(SUC t)) /\
     (~d t /\ x t ==> ~out(SUC t)))"
  2  ["out 0" ]
  1  ["!t. out(SUC t) = d t" ]
```

To eliminate the existencial quantifier, we have to give a value for x. We read it in the formula: $\forall$t. x t = $\neg$d t.

```
# e(EXISTS_TAC "\(y:num). ~ d y" THEN BETA_TAC);;

"!t. (~d t = ~d t) /\ out 0 /\
  (!t'.
    (d t' /\ ~d t' ==> (out(SUC t') = ~out t')) /\
    (~d t' /\ ~~d t' ==> (out(SUC t') = out t')) /\
    (d t' /\ ~~d t' ==> out(SUC t')) /\
    (~d t' /\ ~d t' ==> ~out(SUC t')))"
  2  ["out 0" ]
  1  ["!t. out(SUC t) = d t" ]
```

We note that, eliminating the universal quantifier, we can simplify the goal by rewriting with assumptions:

```
# e(GEN_TAC);;
# e(ASM_REWRITE_TAC []);;

"!t'.
  (d t' /\ ~d t' ==> (d t' = ~out t')) /\
  (~d t' /\ d t' ==> (d t' = out t'))"
  2  ["out 0" ]
  1  ["!t. out(SUC t) = d t" ]
```

The goal is an universal quantification. If we eliminate it, we see a conjunction of two implications whose antecedents are always false. So we could eliminate the universal quantifier and then let the tautology checker do its work:

```
# e(GEN_TAC);;
# e(TAUT_TAC);;

goal proved
...
Previous subproof:
goal proved
```

In a similiar way we could prove the correctness of all other flip-flops implementations. But now you should be able to do this work by yourself, so try it as an exercise.

4.3 Sequential Circuits

Sequential circuits use flip-flops and registers as memory element, and as synchronizers. In this section, we will show through examples some general methods to develop correctness proofs for sequential circuits.

4.3.1 Correctness Proof Development

The UNDISCH_N_TAC and the UNDISCH_BUT_TAC we have shown in the previous section are generally useful, and we will adopt them in all our future examples.

Our approach is to prove things in the simplest, i.e. most direct, way. Previous proofs could be developed in a more concise way, but the result will be difficult to understand. We are interested in showing how to produce proofs, not in teaching how to use HOL to get short, efficent proofs.

There are two ways we can manipulate a proof state:

- we can reason backward, by applying a tactic to the goal formula. The meaning of a tactic is: we can prove from newly generated goals the old one by applying inference rule(s) used by the tactic.

- we can reason forward, by applying an inference rule to assumptions. The meaning of such an action is: we can safely substitute an assumption with a formula that could be derived from.

The HOL theorem prover works better backward, by tactic application. But we need to reason forward sometimes. To do so we use RULE_ASSUM_TAC, which applies an inference rule to assumptions: the tactic fails if the rule cannot be applied to all assumptions.

In general we want to apply a tactic only to a subset of all assumptions. So we developed UNDISCH_N_TAC and UNDISCH_BUT_TAC. Using them we can move assumptions into the goal formula. To get them back we can use STRIP_TAC. This way of manipulating assumptions is simple, direct, and well suited to take a quick approach to proofs, but it is also expensive in time and space, and resulting proofs are not compact, nor efficient.

Another important aspect of a complete proof process are conversions.

A conversion is an equation that says that something is equivalent to something else. By using CONV_TAC we can apply conversion to the goal formula, while using CONV_RULE we can manipulate assumptions.

Conversions are useful, but are superficial; sometimes we need to apply them to embedded terms, so we need to scan the whole term recursively, applying wherever possible the conversion.

For example, let us consider the following two tactics:

```
let DEPTH_ARITH_TAC = CONV_TAC (ONCE_DEPTH_CONV ARITH_CONV);;

let LET_TAC = (CONV_TAC (DEPTH_CONV let_CONV));;
```

The former applies arithmetic conversions, the ones used to resolve arithmetic expressions, scanning the term recursively, in a depth first fashion, but assuring that a subterm will never be converted twice.

The latter is used to rewrite let definitions inside a term. It scan the whole term, by replacing each variable, defined by let with the corresponding value. It works by scanning the term in a depth first fashion, allowing a subterm to be rewritten as many times as needed.

The difference in scanning is needed because arithmetic conversions can get the theorem prover looping, while let declarations are nested in such a way that the most internal can be rewritten safely, so the depth first application will always terminate.

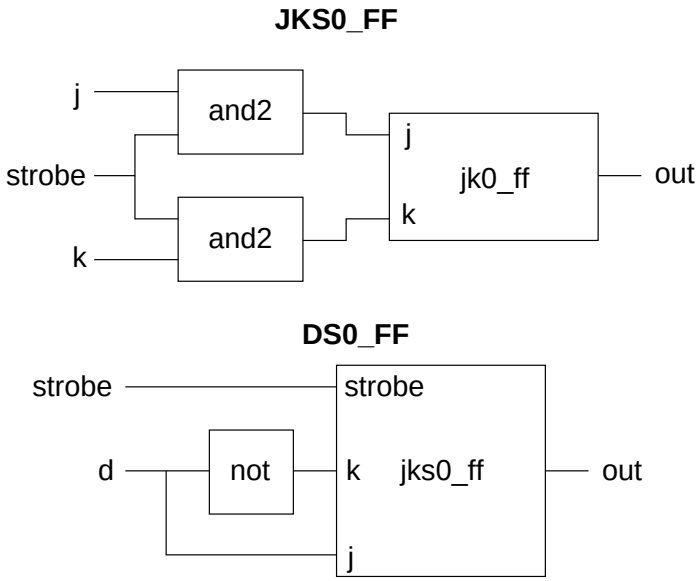


Figure 4.2: The complex flip-flop schema

4.3.2 A Complex Flip-Flop

Our first example is a D type flip-flop with a strobe line. Its implementation schema is shown in figure 4.2. Its truth table is the following:

strobe	d	state	out
0	X	Y	Y
1	0	Y	0
1	1	Y	1

The implementation of this circuit uses a variant of JK flip-flop with initial state equal to 0. In this variant, when strobe line is low, j and k line are forced to 0, thus the flip-flop output is its internal state.

Implementation and the corresponding specification, in HOL syntax, are:

```
let JKS0_FF_IMPL = new_definition
        (‘JKS0_FF_IMPL‘,
         "! j k strobe out. jks0_ff_impl j k strobe out =
                ? x y. !t.
                  jk0_ff x y out /\
                  and2 (j t) (strobe t) (x t) /\
                  and2 (k t) (strobe t) (y t)");;
```

```
let JKS0_FF = new_definition
        (‘JKS0_FF‘,
         "! j k s out. jks0_ff j k s out =
              ~out 0 /\ ! t.
             (~ s t                  ==>
                           (out (SUC t) = out t)) /\
             ((s t /\  j t /\  k t) ==>
                           (out(SUC t) = ~out t)) /\
             ((s t /\ ~j t /\ ~k t) ==>
                           (out(SUC t) =  out t)) /\
             ((s t /\  j t /\ ~k t) ==>
                            out(SUC t))           /\
             ((s t /\ ~j t /\  k t) ==>
                           ~out(SUC t))");;
```

We begin by proving that this flip-flop behaves as expected. The initial steps are the usual ones, i.e. rewriting w.r.t. definitions, stripping, and splitting equality.

But we also add a step where we move universal quantifier inside conjunctions. This simplifies our life when we need to work with assumptions.

The result is here: (we omit non-relevant output)

```
# set_goal([],"! j k s out. jks0_ff_impl j k s out =
                             jks0_ff j k s out");;
# e(REWRITE_TAC [JKS0_FF_IMPL; JKS0_FF; AND2; JK0_FF]);;
# e(REPEAT STRIP_TAC);;
# e(CONV_TAC (TOP_DEPTH_CONV FORALL_AND_CONV));;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

7 subgoals
...
"F"
  8  ["!t. ~out 0" ]
  7  ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  6  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  5  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  4  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  3  ["!t. x t = j t /\ s t" ]
  2  ["!t. y t = k t /\ s t" ]
  1  ["out 0" ]
```

This goal is trivial: we know that out 0 is false (assumption 8), and that it is true (assumption 1), so:

```
# e(UNDISCH_N_TAC 1);;
# e(ASM_REWRITE_TAC []);;

goal proved
Previous subproof:
6 subgoals
...
"out(SUC t) = out t"
  8  ["!t. ~out 0" ]
  7  ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  6  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  5  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  4  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  3  ["!t. x t = j t /\ s t" ]
  2  ["!t. y t = k t /\ s t" ]
  1  ["~s t" ]
```

This subgoal is a bit more difficult. We adopt the following technique:

1. discharge all non-quantified assumptions.

2. specialize assumptions to a common term.

3. repeat previous steps until you can.

4. discharge all assumptions: now, they have to be propositional.

5. use the tautology checker to solve the resulting goal

Application of this technique to our subgoal gives:

```
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
5 subgoals
...
```

```
"out(SUC t) = ~out t"
  10 ["!t. ~out 0" ]
  9  ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  8  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  7  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  6  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  5  ["!t. x t = j t /\ s t" ]
  4  ["!t. y t = k t /\ s t" ]
  3  ["s t" ]
  2  ["j t" ]
  1  ["k t" ]
```

Application of the same technique to the current subgoal gives:

```
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
4 subgoals
...
"out(SUC t) = out t"
  10 ["!t. ~out 0" ]
  9  ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  8  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  7  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  6  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  5  ["!t. x t = j t /\ s t" ]
  4  ["!t. y t = k t /\ s t" ]
  3  ["s t" ]
  2  ["~j t" ]
  1  ["~k t" ]
```

Again we can use the same technique: assumptions 1, 2 and 3 are propositional, so we discharge them; then we specialize universal quantifiers, and assumptions 4, 5 and 10 become propositional, so we discharge them; then we

specialize universal quantifiers again, and all remaining assumptions become propositional, so we discharge them, and the last step is to let the tautology checker prove the subgoal.

The HOL commands, and the corresponding result are the following:

```
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
3 subgoals
...
"out(SUC t)"
  10 ["!t. ~out 0" ]
  9  ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  8  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  7  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  6  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  5  ["!t. x t = j t /\ s t" ]
  4  ["!t. y t = k t /\ s t" ]
  3  ["s t" ]
  2  ["j t" ]
  1  ["~k t" ]
```

Using exactly the same reasoning, we solve also this subgoal:

```
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;
```

```
goal proved
Previous subproof:
2 subgoals
...
"F"
  11 ["!t. ~out 0" ]
  10 ["!t t. x t /\ y t ==> (out(SUC t) = ~out t)" ]
  9  ["!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)" ]
  8  ["!t t. x t /\ ~y t ==> out(SUC t)" ]
  7  ["!t t. ~x t /\ y t ==> ~out(SUC t)" ]
  6  ["!t. x t = j t /\ s t" ]
  5  ["!t. y t = k t /\ s t" ]
  4  ["s t" ]
  3  ["~j t" ]
  2  ["k t" ]
  1  ["out(SUC t)" ]
```

The solution is always the same:

```
# e(UNDISCH_N_TAC 4);;
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
"?x y.
  ((!t. ~out 0) /\
   (!t t. x t /\ y t ==> (out(SUC t) = ~out t)) /\
   (!t t. ~x t /\ ~y t ==> (out(SUC t) = out t)) /\
   (!t t. x t /\ ~y t ==> out(SUC t)) /\
   (!t t. ~x t /\ y t ==> ~out(SUC t))) /\
  (!t. x t = j t /\ s t) /\
  (!t. y t = k t /\ s t)"
  6  ["~out 0" ]
  5  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  4  ["!t. s t /\ j t /\ k t ==> (out(SUC t) = ~out t)" ]
```

```
  3  ["!t. s t /\ ~j t /\ ~k t ==> (out(SUC t) = out t)" ]
  2  ["!t. s t /\ j t /\ ~k t ==> out(SUC t)" ]
  1  ["!t. s t /\ ~j t /\ k t ==> ~out(SUC t)" ]
```

We eliminate the two existencially quantified variables substituting values we read in the last two lines of the goal formula. Then we simplify the goal, by resolving λ-forms, and rewriting with assumptions, and trivial simplifiers. The last step is stripping the goal as much as we can. The result is:

```
# e(EXISTS_TAC "(\(t:num).j t /\ s t)");;
# e(EXISTS_TAC "(\(t:num).k t /\ s t)");;
# e(BETA_TAC);;
# e(ASM_REWRITE_TAC []);;
# e(REPEAT STRIP_TAC);;

4 subgoals
...
"out(SUC t') = ~out t'"
  9  ["~out 0" ]
  8  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  7  ["!t. s t /\ j t /\ k t ==> (out(SUC t) = ~out t)" ]
  6  ["!t. s t /\ ~j t /\ ~k t ==> (out(SUC t) = out t)" ]
  5  ["!t. s t /\ j t /\ ~k t ==> out(SUC t)" ]
  4  ["!t. s t /\ ~j t /\ k t ==> ~out(SUC t)" ]
  3  ["j t'" ]
  2  ["s t'" ]
  1  ["k t'" ]
```

Looking closely at assumptions reveals that, after being specialized to t', assumption *7* implies the goal, and the three conjuncts that forms its antecedent, are exactly assumptions 1, 2 and 3.

The result, in HOL syntax, is:

```
# e(UNDISCH_BUT_TAC 7);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
2 subgoals
...
```

```
"out(SUC t') = out t'"
  8  ["~out 0" ]
  7  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  6  ["!t. s t /\ j t /\ k t ==> (out(SUC t) = ~out t)" ]
  5  ["!t. s t /\ ~j t /\ ~k t ==> (out(SUC t) = out t)" ]
  4  ["!t. s t /\ j t /\ ~k t ==> out(SUC t)" ]
  3  ["!t. s t /\ ~j t /\ k t ==> ~out(SUC t)" ]
  2  ["~(j t' /\ s t')" ]
  1  ["~(k t' /\ s t')" ]
```

Using the technique we developed before, we transform universally quantified assumptions into propositional terms, and then, we use the tautology checker to do the work:

```
# e(UNDISCH_N_TAC 8);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
2 subgoals
...
"out(SUC t')"
  9  ["~out 0" ]
  8  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  7  ["!t. s t /\ j t /\ k t ==> (out(SUC t) = ~out t)" ]
  6  ["!t. s t /\ ~j t /\ ~k t ==> (out(SUC t) = out t)" ]
  5  ["!t. s t /\ j t /\ ~k t ==> out(SUC t)" ]
  4  ["!t. s t /\ ~j t /\ k t ==> ~out(SUC t)" ]
  3  ["j t'" ]
  2  ["s t'" ]
  1  ["~(k t' /\ s t')" ]
```

Again, using the same technique, we are able to prove also this subgoal:

```
# e(UNDISCH_N_TAC 9);;
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
```

```
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
"F"
  10 ["~out 0" ]
  9  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  8  ["!t. s t /\ j t /\ k t ==> (out(SUC t) = ~out t)" ]
  7  ["!t. s t /\ ~j t /\ ~k t ==> (out(SUC t) = out t)" ]
  6  ["!t. s t /\ j t /\ ~k t ==> out(SUC t)" ]
  5  ["!t. s t /\ ~j t /\ k t ==> ~out(SUC t)" ]
  4  ["~(j t' /\ s t')" ]
  3  ["k t'" ]
  2  ["s t'" ]
  1  ["out(SUC t')" ]
```

Here we can reason as follows: from assumption 2 and 4, we infer that $j\ t'$ must be false, so, by assumptions 2, 3 and 5, we infer that $\mathrm{out}(\mathrm{SUC}\ t')$ is false, but assumption 1 says the contrary, so we are able to deduce falsity.

With HOL, we specialize assumption 5 to t', then we discharge everything and let the tautology checker resolve the last subgoal:

```
# e(UNDISCH_BUT_TAC 5);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
```

Now we can define the D type flip-flop with a strobe line.

Its implementation is described directly (see figure 4.2) using the already proved correct subcomponent, while its specification is derived directly from its truth table.

In HOL syntax, the resulting definitions are:

```
let DS0_FF_IMPL = new_definition
        (`DS0_FF_IMPL`,
         "! d strobe out. ds0_ff_impl d strobe out =
                        ? x. ! t.
                                not (d t) (x t) /\
                                jks0_ff d x strobe out");;
```

```
let DS0_FF = new_definition
        (‘DS0_FF‘,
         "! (d:num -> bool) s out. ds0_ff d s out =
            ~out 0 /\
            ! t. (~s t ==> (out (SUC t) = out t)) /\
                 ( s t ==> (out (SUC t) = d t))");;
```

To prove correctness of this flip-flop, we set an equivalence argument, and then we apply standard tacticts to the initial goal. The result follows:

```
set_goal([],"! d s out. ds0_ff_impl d s out =
                        ds0_ff d s out");;
# e(REWRITE_TAC [DS0_FF_IMPL; DS0_FF; NOT; JKS0_FF]);;
# e(REPEAT STRIP_TAC);;
# e(CONV_TAC (TOP_DEPTH_CONV FORALL_AND_CONV));;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

4 subgoals
...
"F"
  8  ["!t. x t = ~d t" ]
  7  ["!t. ~out 0" ]
  6  ["!t t. ~s t ==> (out(SUC t) = out t)" ]
  5  ["!t t. s t /\ d t /\ x t ==> (out(SUC t) = ~out t)" ]
  4  ["!t t. s t /\ ~d t /\ ~x t ==> (out(SUC t) = out t)" ]
  3  ["!t t. s t /\ d t /\ ~x t ==> out(SUC t)" ]
  2  ["!t t. s t /\ ~d t /\ x t ==> ~out(SUC t)" ]
  1  ["out 0" ]
```

Assumptions 1 and 7 are contradictory, so the subgoal is easy:

```
# e(UNDISCH_N_TAC 1);;
# e(ASM_REWRITE_TAC []);;

goal proved
Previous subproof:
3 subgoals
...
"out(SUC t) = out t"
  8  ["!t. x t = ~d t" ]
  7  ["!t. ~out 0" ]
  6  ["!t t. ~s t ==> (out(SUC t) = out t)" ]
  5  ["!t t. s t /\ d t /\ x t ==> (out(SUC t) = ~out t)" ]
```

```
  4  ["!t t. s t /\ ~d t /\ ~x t ==> (out(SUC t) = out t)" ]
  3  ["!t t. s t /\ d t /\ ~x t ==> out(SUC t)" ]
  2  ["!t t. s t /\ ~d t /\ x t ==> ~out(SUC t)" ]
  1  ["~s t" ]
```

We use the standard technique to bring universally quantified assumption in a propositional form:

```
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 6);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
2 subgoals
...
"out(SUC t) = d t"
  8  ["!t. x t = ~d t" ]
  7  ["!t. ~out 0" ]
  6  ["!t t. ~s t ==> (out(SUC t) = out t)" ]
  5  ["!t t. s t /\ d t /\ x t ==> (out(SUC t) = ~out t)" ]
  4  ["!t t. s t /\ ~d t /\ ~x t ==> (out(SUC t) = out t)" ]
  3  ["!t t. s t /\ d t /\ ~x t ==> out(SUC t)" ]
  2  ["!t t. s t /\ ~d t /\ x t ==> ~out(SUC t)" ]
  1  ["s t" ]
```

We apply the preceding technique to the current subgoal:

```
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_N_TAC 7);;
# e(UNDISCH_N_TAC 6);;
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
```

```
"?x.
  (!t. x t = ~d t) /\
  (!t. ~out 0) /\
  (!t t. ~s t ==> (out(SUC t) = out t)) /\
  (!t t. s t /\ d t /\ x t ==> (out(SUC t) = ~out t)) /\
  (!t t. s t /\ ~d t /\ ~x t ==> (out(SUC t) = out t)) /\
  (!t t. s t /\ d t /\ ~x t ==> out(SUC t)) /\
  (!t t. s t /\ ~d t /\ x t ==> ~out(SUC t))"
  3  ["~out 0" ]
  2  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  1  ["!t. s t ==> (out(SUC t) = d t)" ]
```

In the second line of goal formula we read the right substitution for the existentially quantified variable. After substituting, we simplify and strip the goal, obtaining:

```
# e(EXISTS_TAC "\(t:num). ~ d t");;
# e(BETA_TAC THEN ASM_REWRITE_TAC []);;
# e(REPEAT STRIP_TAC);;

4 subgoals
...
"out(SUC t') = ~out t'"
  6  ["~out 0" ]
  5  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  4  ["!t. s t ==> (out(SUC t) = d t)" ]
  3  ["s t'" ]
  2  ["d t'" ]
  1  ["~d t'" ]
```

Assumptions 1 and 2 are contradictory:

```
# e((UNDISCH_N_TAC 1) THEN (ASM_REWRITE_TAC []));;

goal proved
Previous subproof:
...
"out(SUC t') = out t'"
  6  ["~out 0" ]
  5  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  4  ["!t. s t ==> (out(SUC t) = d t)" ]
  3  ["s t'" ]
  2  ["~d t'" ]
  1  ["d t'" ]
```

Again assumption 1 and 2 are contradictory:

```
# e(UNDISCH_N_TAC 1);;
# e(ASM_REWRITE_TAC []);;

goal proved
Previous subproof:
...
"out(SUC t')"
  5  ["~out 0" ]
  4  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  3  ["!t. s t ==> (out(SUC t) = d t)" ]
  2  ["s t'" ]
  1  ["d t'" ]
```

We use the standard technique to transform the current subgoal into a propositional one:

```
# e(UNDISCH_N_TAC 5);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
"F"
  6  ["~out 0" ]
  5  ["!t. ~s t ==> (out(SUC t) = out t)" ]
  4  ["!t. s t ==> (out(SUC t) = d t)" ]
  3  ["s t'" ]
  2  ["~d t'" ]
  1  ["out(SUC t')" ]
```

And the last subgoal is easily solved with the standard technique:

```
# e(UNDISCH_N_TAC 6);;
# e(UNDISCH_N_TAC 3);;
# e(UNDISCH_N_TAC 2);;
# e(UNDISCH_N_TAC 1);;
# e(RULE_ASSUM_TAC (SPEC "t':num"));;
# e(UNDISCH_BUT_TAC 0);;
# e(TAUT_TAC);;
```

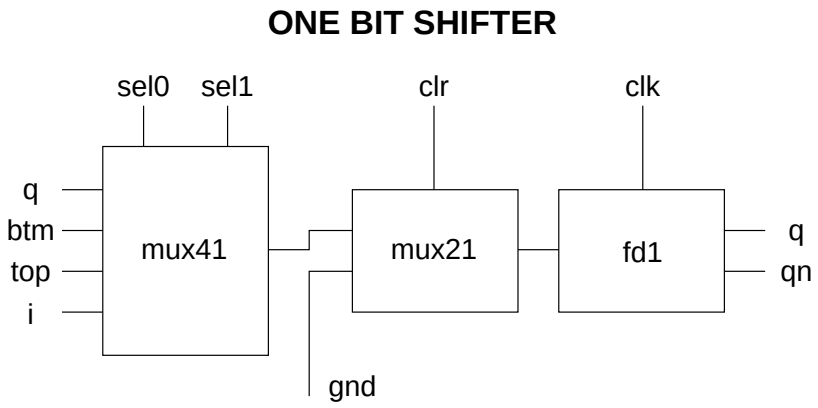


Figure 4.3: The simple shifter schema

4.3.3 A Simple Shifter

In this section we describe a simple 1-bit shifter, and we give it a correctness proof.

Our shifter is shown in figure 4.3. We need some components to define its implementation:

- a multiplexer 4 to 1, which selects one of four input lines, and gives its value to output, according to the value of the selector lines.
- a multiplexer 2 to 1, which selects one of two input lines, and gives its value to output, according to the value of selector line.
- a ground device, which means a line that is false every time.
- a D-type flip flop

All these items are correct, you can prove it by exercise, although we have already proved it for a similar 2 to 1 multiplexer, and for the D-type flip-flop.

Using HOL syntax, their definitions are the following:

```
let MUX41 = new_definition
        (‘MUX41‘,
         "mux41 sel1 sel2 i0 i1 i2 i3 (z:num -> bool) =
            ! (t:num). z t = (~sel1 t /\ ~sel2 t ==> i0 t)
                           /\ (~sel1 t /\  sel2 t ==> i1 t)
                           /\ ( sel1 t /\ ~sel2 t ==> i2 t)
                           /\ ( sel1 t /\  sel2 t ==> i3 t)");;
```

```
let MUX21 = new_definition
        (`MUX21`,
         "mux21 sel1 i0 i1 (z:num -> bool) =
            ! (t:num). z t =       (~sel1 t ==> i0 t)
                                /\ ( sel1 t ==> i1 t)");;

let GND = new_definition
        (`GND`,
         "gnd (t:num) = F");;

let FD1 = new_definition
        (`FD1`,
         " fd1 clk (i:num -> bool) q qn =
             ! t. (q  (SUC t) =  (clk t ==> i t) /\
                                  (~clk t ==> q t)) /\
                  (qn (SUC t) = ~q (SUC t))");;
```

Implementation of the shifter is straigthforward:

```
let SHIFT_IMPL = new_definition
        (`SHIFT_IMPL`,
         "shift_impl clk clr sel0 sel1 i top btm
                     (q:num -> bool) (qn:num->bool) =
            ? p1 p2.
                mux41 sel0 sel1 q btm top i p1 /\
                mux21 clr p1 gnd p2            /\
                fd1 clk p2 q qn");;
```

The shifter can receive four commands through selector lines:

- noop, which means don't change the actual value of output.
- shl, which means shift left the register, so it remembers btm input, and outputs its current state.
- shr, which means shift right, so it remembers top input, and outputs its current state.
- idt, identity, which forces the state to input i, and gives out the current state.

When clock line clk is high commands are received and processed, while when it is low, the output line is stable.

When the clr line is high, a reset occurs, that is, the next state is forced to false.

The shifter has two output state lines, q and qn, and their relation is that value on qn is always the complement of value on q.

The corresponding specification in HOL syntax, is the following:

```
let SHIFT_SPEC = new_definition
        (‘SHIFT_SPEC‘,
         "shift_spec clk clr sel0 sel1 i top btm
                     (q:num -> bool) (qn:num->bool) =
           ! (t:num).
                let noop = (~sel0 t /\ ~sel1 t) in
                let shl  = (~sel0 t /\  sel1 t) in
                let shr  = ( sel0 t /\ ~sel1 t) in
                let idt  = ( sel0 t /\  sel1 t) in
                let res  = (clk t ==>
                            ( clr t ==> F)                  /\
                            (~clr t /\ noop ==>   q t)      /\
                            (~clr t /\  shl ==> btm t)      /\
                            (~clr t /\  shr ==> top t)      /\
                            (~clr t /\  idt ==>   i t))
                           /\ (~clk t ==> q t) in
                ((q (SUC t) = res) /\
                 (qn (SUC t) = ~res))");;
```

The correctness argument is an equivalence between implementation and specification. First steps are the usual one: rewriting with definitions, expanding the let operator, stripping and splitting equality. Here is the answer from HOL:

```
# set_goal([],"! clk clr sel0 sel1 i top btm
               (q:num -> bool) (qn:num->bool).
               shift_impl clk clr sel0 sel1 i top btm q qn =
               shift_spec clk clr sel0 sel1 i top btm q qn");;
# e(REWRITE_TAC [SHIFT_IMPL; SHIFT_SPEC; MUX41;
                 MUX21; FD1; GND]);;
# e(LET_TAC);;
# e(REPEAT STRIP_TAC);;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

3 subgoals
...
"q(SUC t) =
 (clk t ==>
  ~clr t /\
  (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
  (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
```

```
 (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
 (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
(~clk t ==> q t)"
 3  ["!t.
       p1 t =
       (~sel0 t /\ ~sel1 t ==> q t) /\
       (~sel0 t /\ sel1 t ==> btm t) /\
       (sel0 t /\ ~sel1 t ==> top t) /\
       (sel0 t /\ sel1 t ==> i t)" ]
 2  ["!t. p2 t = (~clr t ==> p1 t) /\ ~clr t" ]
 1  ["!t.
       (q(SUC t) = (clk t ==> p2 t) /\ (~clk t ==> q t)) /\
       (qn(SUC t) = ~q(SUC t))" ]
```

The goal seems to be very complex, but if we look closely, it can be almost solved by rewriting $q(\mathrm{SUC}\ t)$ with assumptions. To do this, we need to remove universal quantifiers from assumptions, then rewrite with assumptions, and the result is a purely propositional goal, which can be solved by tautology checker:

```
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(ASM_REWRITE_TAC []);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
2 subgoals
...
"qn(SUC t) =
 ~((clk t ==>
    ~clr t /\
    (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
    (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
    (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
    (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
   (~clk t ==> q t))"
  3  ["!t.
        p1 t =
        (~sel0 t /\ ~sel1 t ==> q t) /\
        (~sel0 t /\ sel1 t ==> btm t) /\
        (sel0 t /\ ~sel1 t ==> top t) /\
        (sel0 t /\ sel1 t ==> i t)" ]
  2  ["!t. p2 t = (~clr t ==> p1 t) /\ ~clr t" ]
```

```
1  ["!t.
      (q(SUC t) = (clk t ==> p2 t) /\ (~clk t ==> q t)) /\
      (qn(SUC t) = ~q(SUC t))" ]
```

This subgoal is identical to the previous one, except it tries to prove correctness of the qn output line. The way to solve it is the same:

```
# e(RULE_ASSUM_TAC (SPEC "t:num"));;
# e(ASM_REWRITE_TAC []);;
# e(TAUT_TAC);;

goal proved
Previous subproof:
"?p1 p2.
  (!t.
    p1 t =
    (~sel0 t /\ ~sel1 t ==> q t) /\
    (~sel0 t /\ sel1 t ==> btm t) /\
    (sel0 t /\ ~sel1 t ==> top t) /\
    (sel0 t /\ sel1 t ==> i t)) /\
  (!t. p2 t = (~clr t ==> p1 t) /\ ~clr t) /\
  (!t.
    (q(SUC t) = (clk t ==> p2 t) /\ (~clk t ==> q t)) /\
    (qn(SUC t) = ~q(SUC t)))"
  1  ["!t.
       (q(SUC t) =
        (clk t ==>
         ~clr t /\
         (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
         (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
         (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
         (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
        (~clk t ==> q t)) /\
       (qn(SUC t) =
        ~((clk t ==>
           ~clr t /\
           (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
           (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
           (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
           (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
          (~clk t ==> q t)))" ]
```

From the goal formula we can read the right substitutions for p1 and p2. The resulting goal, after elimination of the two existencial quantifiers, is:

```
# e(EXISTS_TAC "\(t:num).
                        (~sel0 t /\ ~sel1 t ==> q t) /\
                        (~sel0 t /\  sel1 t ==> btm t) /\
                        ( sel0 t /\ ~sel1 t ==> top t) /\
                        ( sel0 t /\  sel1 t ==> i t)");;
# e(BETA_TAC);;
# e(EXISTS_TAC "\(t:num).
                        (~clr t ==>
                            (~sel0 t /\ ~sel1 t ==> q t)   /\
                            (~sel0 t /\  sel1 t ==> btm t) /\
                            ( sel0 t /\ ~sel1 t ==> top t) /\
                            ( sel0 t /\  sel1 t ==> i t))  /\
                         ~clr t");;
# e(BETA_TAC);;


"(!t.
   (~sel0 t /\ ~sel1 t ==> q t) /\
   (~sel0 t /\ sel1 t ==> btm t) /\
   (sel0 t /\ ~sel1 t ==> top t) /\
   (sel0 t /\ sel1 t ==> i t) =
   (~sel0 t /\ ~sel1 t ==> q t) /\
   (~sel0 t /\ sel1 t ==> btm t) /\
   (sel0 t /\ ~sel1 t ==> top t) /\
   (sel0 t /\ sel1 t ==> i t)) /\
 (!t.
   (~clr t ==>
    (~sel0 t /\ ~sel1 t ==> q t) /\
    (~sel0 t /\ sel1 t ==> btm t) /\
    (sel0 t /\ ~sel1 t ==> top t) /\
    (sel0 t /\ sel1 t ==> i t)) /\
   ~clr t =
   (~clr t ==>
    (~sel0 t /\ ~sel1 t ==> q t) /\
    (~sel0 t /\ sel1 t ==> btm t) /\
    (sel0 t /\ ~sel1 t ==> top t) /\
    (sel0 t /\ sel1 t ==> i t)) /\
   ~clr t) /\
 (!t.
   (q(SUC t) =
    (clk t ==>
     (~clr t ==>
      (~sel0 t /\ ~sel1 t ==> q t) /\
      (~sel0 t /\ sel1 t ==> btm t) /\
      (sel0 t /\ ~sel1 t ==> top t) /\
```

```
    (sel0 t /\ sel1 t ==> i t)) /\
   ~clr t) /\
  (~clk t ==> q t)) /\
 (qn(SUC t) = ~q(SUC t)))"
1  ["!t.
     (q(SUC t) =
      (clk t ==>
       ~clr t /\
       (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
       (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
       (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
       (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
      (~clk t ==> q t)) /\
     (qn(SUC t) =
      ~((clk t ==>
         ~clr t /\
         (~clr t /\ ~sel0 t /\ ~sel1 t ==> q t) /\
         (~clr t /\ ~sel0 t /\ sel1 t ==> btm t) /\
         (~clr t /\ sel0 t /\ ~sel1 t ==> top t) /\
         (~clr t /\ sel0 t /\ sel1 t ==> i t)) /\
        (~clk t ==> q t)))" ]
```

Goal formula has the form:

$$(\forall t.X = X) \wedge (\forall t.Y = Y) \wedge (\forall t.Z\ t)$$

By rewriting with assumptions we can simplify the goal formula which becomes:

$$(\forall t.Z'\ t)$$

where Z' is Z rewritten.

Then we eliminate the universal quantifier using the STRIP_TAC and we will have a propositional formula.

The last, now obvious, step is to solve it by using the tautology checker.

```
# e(ASM_REWRITE_TAC []);;
# e(REPEAT STRIP_TAC);;
# e(TAUT_TAC);;

goal proved
```

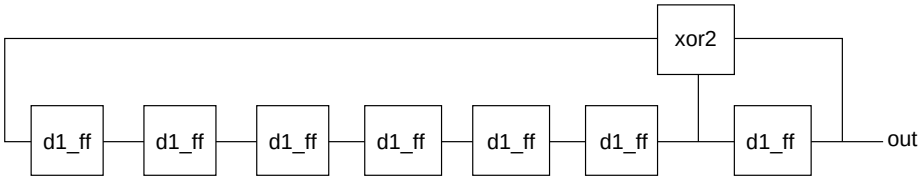


Figure 4.4: The sequential scrambler

4.3.4 A Scrambler

In this section we describe a more complex circuit: a scrambler. A scrambler is a circuit used to produce deterministic *garbage*, that will be combined with data sent on a trasmission line. We want such an effect because there are digital detectors of the state of line, that, if detect a long number of 0 or 1, think that the line is broken (either cut, or shorted).

In this example we will not specify the circuit with a behavioural model, but with another circuit. In fact we know that there is a correct circuit, but it is sequential, that is, it produces only one bit per clock cycle, while we want to prove correctness of an equivalent circuit which produces eight bits per clock cycle.

Our strategy will be:

- preliminary subcomponents specification
- definition of the two circuit implementation
- equivalence of circuits' behaviours by a function that parallelizes and shift in time the output of the sequential version.

From figure 4.5, we need a D-type flip-flop, and we note that it is always coupled with a XOR2 gate, so we define another component which is the combination of the flip-flop with the gate:

```
let FF = new_definition
        (`FF`,
         "ff in out = ~out 0 /\ (!t. out (SUC t) = in t)");;

let XFF = new_definition
        (`XFF`,
         "xff a b out = ?x. (!t. xor2 (a t) (b t) (x t)) /\
                                  ff x out");;
```

With these definitions is simple to produce the implementation of the two circuits. We use the implementation of the sequential version as our specification, and the parallel version of the scrambler will act as our real implementation:

```
let SCRAMBLER_SPEC = new_definition
        (‘SCRAMBLER_SPEC‘,
         "scrambler_spec w = ? w1 w2 w3 w4 w5 w6 w7.
                ff w1  w /\
                ff w2 w1 /\
                ff w3 w2 /\
                ff w4 w3 /\
                ff w5 w4 /\
                ff w6 w5 /\
                ff w7 w6 /\
                (!t. xor2 (w1 t) (w t) (w7 t))");;

let SCRAMBLER_IMPL = new_definition
        (‘SCRAMBLER_IMPL‘,
         "scrambler_impl s7 s6 s5 s4 s3 s2 s1 s0 =
                xff s5 s6 s7 /\
                xff s4 s5 s6 /\
                xff s3 s4 s5 /\
                xff s2 s3 s4 /\
                xff s1 s2 s3 /\
                xff s0 s1 s2 /\
                xff s5 s7 s1 /\
                (!t. xor2 (s6 t) (s7 t) (s0 t))");;
```

Now we need to define the transformation function, that manage the different timing of the two circuits, in such a way to render their outputs equals (we hope).

The 0 line of the parallel version at time 0 must have the same value of output line of the sequential circuit at time 0; the 1 line of the parallel version at time 0 must have the same value of output line of the sequential version at time 1, and so on; the 0 line must have at time 1 the same value as output line at time 8. The formula that converts time from the parallel version to the sequential version must be:

$$(parallel\ output\ n)t = sequential\ output\ (t * 8 + n)$$

So we can define the time mapping function, along with some useful conversion to manage the first ten natural numbers:

```
let TIME_MAP = new_definition
        (‘TIME_MAP‘,
         "time t n = (t * 8) + n");;
```

```
let num_convs = [num_CONV "1";num_CONV "2";num_CONV "3";
                 num_CONV "4";num_CONV "5";num_CONV "6";
                 num_CONV "7";num_CONV "8";num_CONV "9"];;
```

We set the correctness argument in a different way from previous examples: we say that if output of the sequential circuit is w and output of the parallel circuit is the array s then, for all time instants t, the elements in array s are in the $time$ relation with w.

```
# set_goal([],"! s w. scrambler_spec w /\
                    scrambler_impl (s 0) (s 1) (s 2) (s 3)
                                   (s 4) (s 5) (s 6) (s 7) ==>
                (! t.   ((s 0) t = w (time t 0)) /\
                        ((s 1) t = w (time t 1)) /\
                        ((s 2) t = w (time t 2)) /\
                        ((s 3) t = w (time t 3)) /\
                        ((s 4) t = w (time t 4)) /\
                        ((s 5) t = w (time t 5)) /\
                        ((s 6) t = w (time t 6)) /\
                        ((s 7) t = w (time t 7)))");;
```

The first step in the proof is rewriting with definitions; then we strip goal formula until its principal functor will be the $\forall t$ universal quantifier. Then we reason by induction on time:

```
# e(REWRITE_TAC [SCRAMBLER_SPEC; SCRAMBLER_IMPL;
                 FF; XFF; XOR2; TIME_MAP]);;
# e((REPEAT GEN_TAC) THEN STRIP_TAC);;
# e(INDUCT_TAC);;

2 subgoals
"(s 0 0 = w((0 * 8) + 0)) /\
 (s 1 0 = w((0 * 8) + 1)) /\
 (s 2 0 = w((0 * 8) + 2)) /\
 (s 3 0 = w((0 * 8) + 3)) /\
 (s 4 0 = w((0 * 8) + 4)) /\
 (s 5 0 = w((0 * 8) + 5)) /\
 (s 6 0 = w((0 * 8) + 6)) /\
 (s 7 0 = w((0 * 8) + 7))"
  37 ["~w 0" ]
  36 ["!t. w(SUC t) = w1 t" ]
```

```
 35 ["~w1 0" ]
 34 ["!t. w1(SUC t) = w2 t" ]
 33 ["~w2 0" ]
 32 ["!t. w2(SUC t) = w3 t" ]
 31 ["~w3 0" ]
 30 ["!t. w3(SUC t) = w4 t" ]
 29 ["~w4 0" ]
 28 ["!t. w4(SUC t) = w5 t" ]
 27 ["~w5 0" ]
 26 ["!t. w5(SUC t) = w6 t" ]
 25 ["~w6 0" ]
 24 ["!t. w6(SUC t) = w7 t" ]
 23 ["!t. w7 t = ~(w1 t = w t)" ]
 22 ["!t. x t = ~(s 2 t = s 1 t)" ]
 21 ["~s 0 0" ]
 20 ["!t. s 0(SUC t) = x t" ]
 19 ["!t. x' t = ~(s 3 t = s 2 t)" ]
 18 ["~s 1 0" ]
 17 ["!t. s 1(SUC t) = x' t" ]
 16 ["!t. x'' t = ~(s 4 t = s 3 t)" ]
 15 ["~s 2 0" ]
 14 ["!t. s 2(SUC t) = x'' t" ]
 13 ["!t. x''' t = ~(s 5 t = s 4 t)" ]
 12 ["~s 3 0" ]
 11 ["!t. s 3(SUC t) = x''' t" ]
 10 ["!t. x'''' t = ~(s 6 t = s 5 t)" ]
 9  ["~s 4 0" ]
 8  ["!t. s 4(SUC t) = x'''' t" ]
 7  ["!t. x''''' t = ~(s 7 t = s 6 t)" ]
 6  ["~s 5 0" ]
 5  ["!t. s 5(SUC t) = x''''' t" ]
 4  ["!t. x'''''' t = ~(s 2 t = s 0 t)" ]
 3  ["~s 6 0" ]
 2  ["!t. s 6(SUC t) = x'''''' t" ]
 1  ["!t. s 7 t = ~(s 1 t = s 0 t)" ]
```

The goal seems to be very complex, but, using assumptions in the proper way, and performing the arithmetic calculations, it can be easily proved, infact:

```
# e(ASM_REWRITE_TAC (num_convs @ [MULT; ADD_CLAUSES]));;

goal proved
```

```
Previous subproof:
"(s 0(SUC t) = w(((SUC t) * 8) + 0)) /\
 (s 1(SUC t) = w(((SUC t) * 8) + 1)) /\
 (s 2(SUC t) = w(((SUC t) * 8) + 2)) /\
 (s 3(SUC t) = w(((SUC t) * 8) + 3)) /\
 (s 4(SUC t) = w(((SUC t) * 8) + 4)) /\
 (s 5(SUC t) = w(((SUC t) * 8) + 5)) /\
 (s 6(SUC t) = w(((SUC t) * 8) + 6)) /\
 (s 7(SUC t) = w(((SUC t) * 8) + 7))"
  38 ["~w 0" ]
  37 ["!t. w(SUC t) = w1 t" ]
  36 ["~w1 0" ]
  35 ["!t. w1(SUC t) = w2 t" ]
  34 ["~w2 0" ]
  33 ["!t. w2(SUC t) = w3 t" ]
  32 ["~w3 0" ]
  31 ["!t. w3(SUC t) = w4 t" ]
  30 ["~w4 0" ]
  29 ["!t. w4(SUC t) = w5 t" ]
  28 ["~w5 0" ]
  27 ["!t. w5(SUC t) = w6 t" ]
  26 ["~w6 0" ]
  25 ["!t. w6(SUC t) = w7 t" ]
  24 ["!t. w7 t = ~(w1 t = w t)" ]
  23 ["!t. x t = ~(s 2 t = s 1 t)" ]
  22 ["~s 0 0" ]
  21 ["!t. s 0(SUC t) = x t" ]
  20 ["!t. x' t = ~(s 3 t = s 2 t)" ]
  19 ["~s 1 0" ]
  18 ["!t. s 1(SUC t) = x' t" ]
  17 ["!t. x'' t = ~(s 4 t = s 3 t)" ]
  16 ["~s 2 0" ]
  15 ["!t. s 2(SUC t) = x'' t" ]
  14 ["!t. x''' t = ~(s 5 t = s 4 t)" ]
  13 ["~s 3 0" ]
  12 ["!t. s 3(SUC t) = x''' t" ]
  11 ["!t. x'''' t = ~(s 6 t = s 5 t)" ]
  10 ["~s 4 0" ]
  9  ["!t. s 4(SUC t) = x'''' t" ]
  8  ["!t. x''''' t = ~(s 7 t = s 6 t)" ]
  7  ["~s 5 0" ]
  6  ["!t. s 5(SUC t) = x''''' t" ]
  5  ["!t. x'''''' t = ~(s 2 t = s 0 t)" ]
  4  ["~s 6 0" ]
```

```
 3  ["!t. s 6(SUC t) = x'''''' t" ]
 2  ["!t. s 7 t = ~(s 1 t = s 0 t)" ]
 1  ["(s 0 t = w((t * 8) + 0)) /\
      (s 1 t = w((t * 8) + 1)) /\
      (s 2 t = w((t * 8) + 2)) /\
      (s 3 t = w((t * 8) + 3)) /\
      (s 4 t = w((t * 8) + 4)) /\
      (s 5 t = w((t * 8) + 5)) /\
      (s 6 t = w((t * 8) + 6)) /\
      (s 7 t = w((t * 8) + 7))" ]
```

As for the base case, the winning move is to do arithmetic calculations, and rewriting with assumptions. The goal reduces to:

```
# e(ASM_REWRITE_TAC (num_convs @ [MULT; ADD_CLAUSES]));;
"~(w2(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))) =
   w(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))) =
 ~(~(w2(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))) =
     w1(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))) =
  ~(w1(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))) =
     w(t * (SUC(SUC(SUC(SUC(SUC(SUC(SUC(SUC 0)))))))))"
 38 ["~w 0" ]
     ...
  1  [...]
```

But this is a simple tautology! So we prove the final step with the tautology checker:

```
# e(TAUT_TAC);;

goal proved
```

We end this chapter with a brief consideration: you have seen that simple examples admit complex proofs, while complex examples admit simple proofs. This fact is not always true, but, in general, is much more difficult to prove correctness of "obviously" right things than of complex ones.

The main trick is to have long goals, but with a simple structure, so it is easy to analyze them.

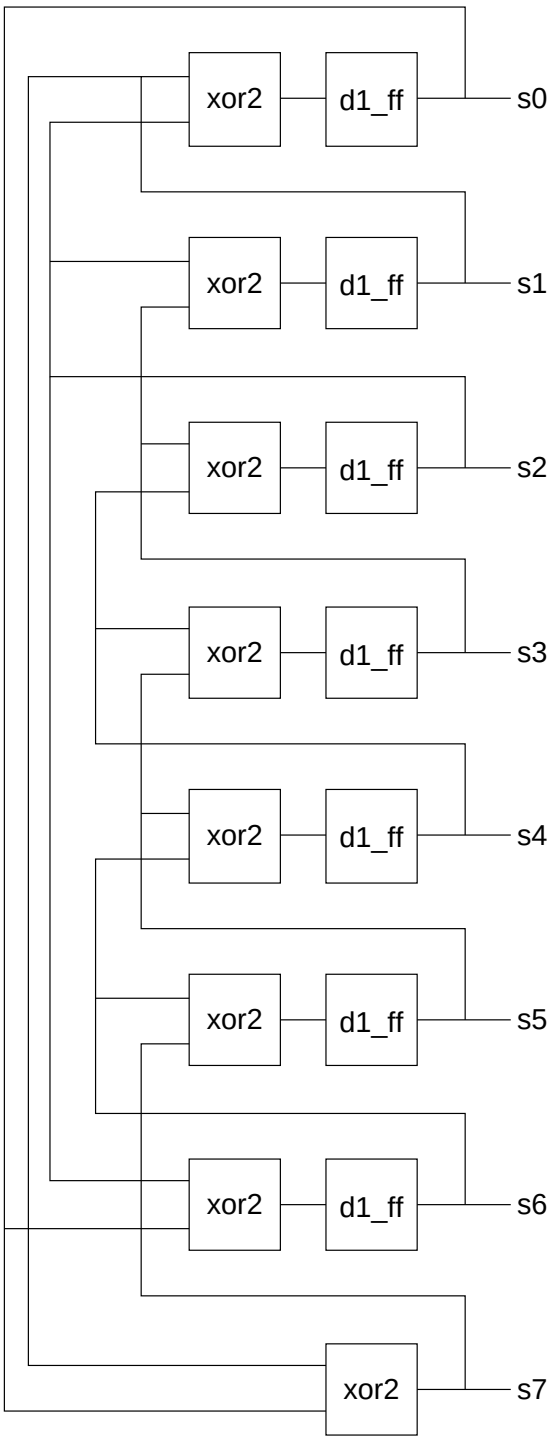


Figure 4.5: The parallel scrambler

Chapter 5

Finite State Machines

In this chapter we define in a precise, mathematical way what are Finite State Machines, and we derive some of their properties. We will use these results to prove correctness of circuits where a finite state machine is given as a specification.

5.1 Theory of Finite State Machines

In this section we develop basics for theory of finite state machines. We introduce their formal definition, and we prove a theorem which says how to reduce a machine to a standard format.

5.1.1 Definitions

There are two definitions of finite state machines; Moore machines and Mealy machines. They can be proved to be equivalent, that is, for every machine of one type, there is a machine of the other type, that computes the same function. We are not interested in proving these properties, but we are interested in proving that there is a standard format for both types, and that this format can be used to prove correctness of circuits.

Mealy finite state machines are defined as follows:

Def: 1 *Let* K, Σ *and* O *be finite sets of symbols, and let* δ *and* λ *be functions such that:*

$$\begin{aligned} \delta &: K \times \Sigma \longrightarrow K \\ \lambda &: K \times \Sigma \longrightarrow O \end{aligned}$$

then a Mealy finite state machine is defined as:

$$FSM = \langle K, \Sigma, O, \delta, \lambda \rangle$$

The corresponding definition for Moore machines is:

Def: 2 *Let* K, Σ *and* O *be finite sets of symbols, and let* δ *and* λ *be functions such that:*

$$\begin{aligned} \delta &: K \times \Sigma \longrightarrow K \\ \lambda &: K \longrightarrow O \end{aligned}$$

then a Moore finite state machine is defined as:

$$FSM = \langle K, \Sigma, O, \delta, \lambda \rangle$$

δ function is called state transition function, or, in short, transition function. λ function is called output function.

Given an initial state k_0 and an input string in of symbols from Σ, a computation is defined by means of the $\delta*$ and $\lambda*$ functions.

The former computes the state after n computation steps, with $n < L$, where L is the length of in. The latter gives the output after n steps, with $n < L$. Their formal definitions are given by recursion on natural numbers:

$$\begin{aligned} \delta^* \; in \; 0 &= k_0 \\ \delta^* \; in \; (n+1) &= \delta(\delta^* \; in \; n)(in \; (n+1)) \end{aligned}$$

$$\lambda^* \; in \; n = \lambda(\delta^* \; in \; n)(in \; (n+1))$$

The definition of λ^* for a Moore machine is simply:

$$\lambda^* \; in \; n = \lambda(\delta^* \; in \; n)$$

We say that two finite state machines are equivalent if, given a fixed initial state for both of them, and the same input string, they produce the same output string.

We can relax the previous definition by allowing a bijective map between input alphabets and output alphabets.

5.1.2 Boolean Reducibility

We want to prove that exists a *normal* form for a given finite state machine. This means that, given any machine, we can find another machine, with a restricted form, that is equivalent to the first one.

To reach such a result we need to define *Boolean Finite State Machines.*

Def: 3 *A Finite State Machine (either a Mealy or a Moore one), is said to be boolean if the following conditions holds:*

$$\begin{aligned} K &= \{0,1\}^\alpha \\ \Sigma &= \{0,1\}^\beta \\ O &= \{0,1\}^\gamma \end{aligned}$$

where $\{0,1\}^n$ *is the set of all strings of boolean elements with length* n*, and* α*,* β *and* γ *are natural numbers.*

We want to prove that exists a boolean finite state machine which is equivalent to a given a finite state machine.

Theorem: 1 *Let* $A = \langle K, \Sigma, O, \delta, \lambda \rangle$ *be a generic Mealy finite state machine, then exists a Mealy boolean finite state machine* $B = \langle K', \Sigma', O', \delta', \lambda' \rangle$*, such that* A *and* B *are equivalent.*

Proof

Let define dimensions of alphabets for B:

$$\begin{aligned} \alpha &= \min\{n \mid 2^n \geq |K|\} \\ \beta &= \min\{n \mid 2^n \geq |\Sigma|\} \\ \gamma &= \min\{n \mid 2^n \geq |O|\} \end{aligned}$$

With these data we define alphabets of B:

$$\begin{aligned} K' &= \{0,1\}^\alpha \\ \Sigma' &= \{0,1\}^\beta \\ O' &= \{0,1\}^\gamma \end{aligned}$$

Let's take three injective functions:

$$\begin{aligned} \tau &: K \longrightarrow K' \\ \sigma &: \Sigma \longrightarrow \Sigma' \\ \omega &: O \longrightarrow O' \end{aligned}$$

Let $\overrightarrow{0}$ be the boolean string composed by all 0s.

We impose the condition that, if k_0 is the initial state of A then:

$$\tau\ k_0 = \overrightarrow{0}$$

We define the transition function of B as:

$$\delta'\ a\ b = \begin{cases} \tau(\delta\ x\ y) & \text{if } \exists x\ y.(a = \tau\ x \wedge b = \sigma\ y) \\ a & \text{otherwise} \end{cases}$$

The output function of B is:

$$\lambda'\ a\ b = \begin{cases} \omega(\lambda\ x\ y) & \text{if } \exists x\ y.(a = \tau\ x \wedge b = \sigma\ y) \\ \overrightarrow{0} & \text{otherwise} \end{cases}$$

A is equivalent to B iff:

$$\begin{aligned} \forall k\ i. \quad & \omega(\lambda\ k\ i) = \lambda'(\tau\ k)(\sigma\ i) \quad \wedge \\ & \tau(\delta\ k\ i) = \delta'(\tau\ k)(\sigma\ i) \end{aligned}$$

But this is obviously true, because this are the definitions of δ' and λ'.

End Of Proof.

A symmetric theorem holds for Moore machines.

Now we can focus our attention to boolean finite state machines.

A final note: the previous theorem is constructive, i.e. it describes the effective way to construct an equivalent boolean machine, given a generic finite state machine.

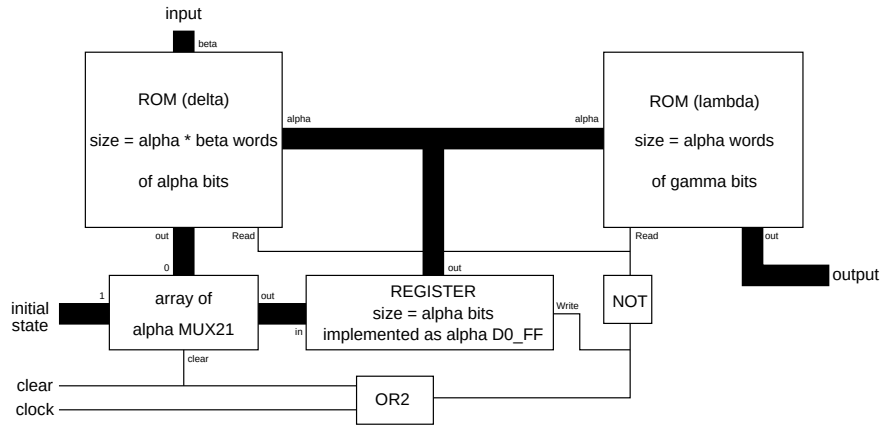


Figure 5.1: Clocked hardware implementation of a Moore boolean machine

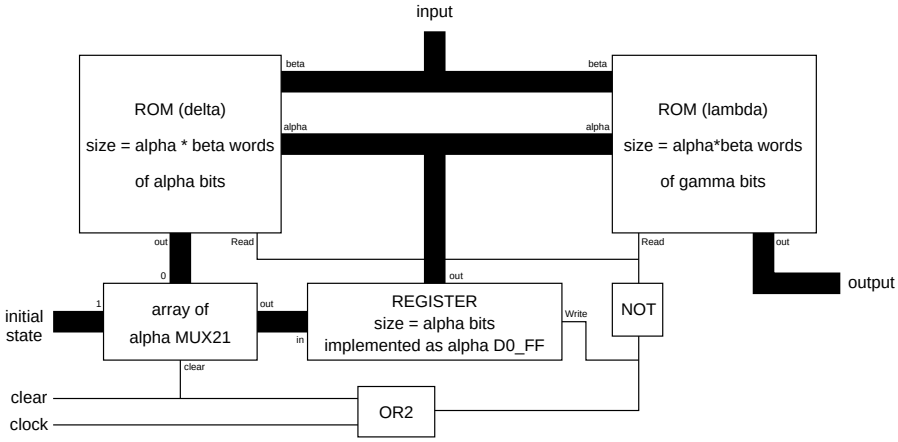


Figure 5.2: Clocked hardware implementation of a Mealy boolean machine

5.2 Canonical Implementation

In this section we show that boolean finite state machines could be implemented as hardware circuits. We show that the implementation we give is correct, and we call it *canonical*.

5.2.1 Definitions

We want to prove correctness of the circuit we have choose as the standard way to represent a finite state machine.

We have four ways to represent finite state machines: two for every type, i.e. Moore's or Mealy's. Because of the structure of circuits and because definitions of the finite state machines are very similar, we will present only proofs for the

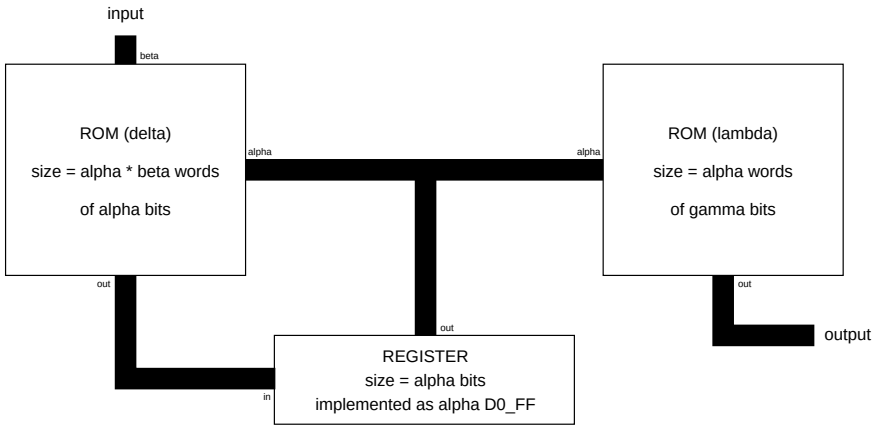


Figure 5.3: Simple hardware implementation of a Moore boolean machine

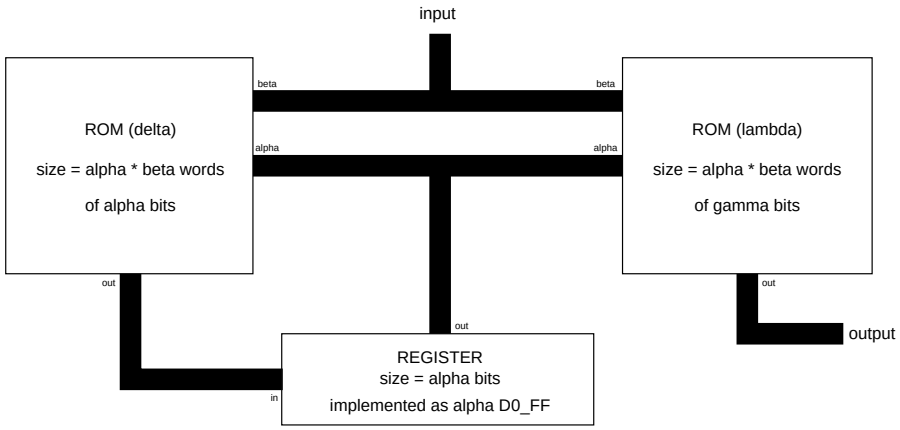


Figure 5.4: Simple hardware implementation of a Mealy boolean machine

Moore version, and you can try to prove correctness for the Mealy version (it is a bit simpler).

We are not interested in the clocked circuit, because it is simple, although long, to prove its correctness, given the simple implementation.

Now we will develop a set of definitions to represent the simple implementation of a Moore finite state machine; then we give specification for a Moore machine, and we develop the correctness argument.

The circuit we need to represent is shown in figure 5.3. It contains three type of components:

- A register which contains α bits. It works like a D-type flip-flop, and its initial value is $\overrightarrow{0}$.
- A ROM with an input bus of α bits. Its output has width γ.

- A ROM with two input buses: one has width α bits, while the other has width β bits. Its output has width α bits.

Sizes names are the same ones we used in the previous section to name cardinality of boolean finite machine's alphabets.

In our circuit, the ROM marked as $delta$ has to compute the δ function, while the ROM marked as $lambda$ computes the λ function.

Let's begin to specify to HOL how to represent our components.

The very first thing to define is the initial state of the boolean machine. Here we need a decision: what HOL type to give to boolean states. Because we assumed states to be array of bits, with a fixed but unknown length, there is no simple way to tell to HOL this fact. We choose to relax a bit the type: we choose that a generic state is a list of bits.

Representation now is simple: for every machine, the simplest "state" it can be is the null list of booleans:

```
let ZEROVEC = new_definition
        (`ZEROVEC`,
         "zero = (NIL:(bool)list)");;
```

The register is like a D-type flip-flop: it delays its input of exactly one instant . We know that the register must contain $\overrightarrow{0}$ as the initial value:

```
let REG = new_definition
        (`REG`,
         "reg in out = (out 0 = zero)
                     /\ (!t. out (SUC t) = in t)");;
```

The ROMs are of two kinds: with one address line or two address lines. We call ROM1 and ROM2 the different kinds.

To represent a ROM, we need to know how it was programmed. A ROM is a read-only memory, where every address contains a word that is *function* of the address. We can define a function to compute values in the ROM, using the standard techniques to derive boolean functions from their truth table, that, in this case, corresponds to the content of the ROM.

Because a ROM produces values of a function whose truth table is stored in, we could specify a ROM in the following way:

```
let ROM1 = new_definition
        (`ROM1`,
         "rom1 f (in:num -> (bool)list)
                 (out:num -> (bool)list) =
                (!t. out t = f (in t))");;
```

```
let ROM2 = new_definition
        (‘ROM2‘,
         "rom2 f (in1:num -> (bool)list)
                 (in2:num -> (bool)list)
                 (out:num -> (bool)list) =
                (!t. out t = f (in1 t) (in2 t))");;
```

Now to define implementation of the boolean Moore machine is easy:

```
let MOORE_IMPL = new_definition
        (‘MOORE_IMPL‘,
         "moore_impl delta lambda in out =
                (? h k.  rom2 delta h in k
                      /\ rom1 lambda h out
                      /\ reg k h)");;
```

To define a specification, we need to solve a problem: how to represent δ^* and λ^* functions. The latter is simple to represent, but the former no. To solve this problem, we define a function which takes only one argument, the time, and produces a function, which, given a δ and an input, produces the resulting state at the given time.

We call this higher order function $\mathrm{fold\_states}$, and it is defined by induction over natural numbers:

$$\begin{aligned} \mathrm{fold\_states}\ 0 &= \lambda\ \mathrm{delta}\ \mathrm{in}.\ \overrightarrow{0} \\ \mathrm{fold\_states}\ (n+1) &= \lambda\ \mathrm{delta}\ \mathrm{in}.\ \mathrm{delta}(\mathrm{fold\_states}\ n\ \mathrm{delta}\ \mathrm{in})(\mathrm{in}\ n) \end{aligned}$$

Using the HOL syntax, this becomes:

```
let FOLD_STATES = new_prim_rec_definition
        (‘FOLD_STATES‘,
         "(fold_states 0 =
                (\(x:(bool)list -> ((bool)list -> (bool)list))
                            (y:num -> (bool)list).
                               zero)) /\
          (fold_states (SUC n) =
                (\(x:(bool)list -> ((bool)list -> (bool)list))
                  (y:num -> (bool)list).
                        x (fold_states n x y) (y n)))");;
```

The specification becomes, remembering the definition of λ^*:

```
let MOORE_SPEC = new_definition
        (‘MOORE_SPEC‘,
         "moore_spec delta lambda in (out:num -> (bool)list) =
             (!t. out t = lambda (fold_states t delta in))");;
```

5.2.2 Correctness of the Canonical Implementation

With previous definitions, the proof we are searching for, is a normal hardware verification.

First steps are the usual ones, as the correctness argument:

```
# set_goal([],"!delta lambda in out.
                    moore_spec delta lambda in out =
                    moore_impl delta lambda in out");;
# e(REWRITE_TAC [MOORE_IMPL; MOORE_SPEC]);;
# e(REWRITE_TAC [ROM2; ROM1; REG]);;
# e(REPEAT STRIP_TAC);;
# e(EQ_TAC THEN STRIP_TAC);;

2 subgoals
...
"?h k.
  (!t. k t = delta(h t)(in t)) /\
  (!t. out t = lambda(h t)) /\
  (h 0 = zero) /\
  (!t. h(SUC t) = k t)"
  1  ["!t. out t = lambda(fold_states t delta in)" ]
```

The right substitution for `h` is obtained comparing assumption with the third line of the goal formula. The right substitution for `k` is in the first line of the goal formula. The result in HOL syntax is:

```
# e(EXISTS_TAC "\t. fold_states t delta in");;
# e(EXISTS_TAC "\t. delta (fold_states t delta in) (in t)");;
# e(BETA_TAC);;

"(!t.
   delta(fold_states t delta in)(in t) =
   delta(fold_states t delta in)(in t)) /\
 (!t. out t = lambda(fold_states t delta in)) /\
 (fold_states 0 delta in = zero) /\
 (!t. fold_states(SUC t)delta in =
      delta(fold_states t delta in)(in t))"
  1  ["!t. out t = lambda(fold_states t delta in)" ]
```

Now, the first conjunct in the goal formula is trivial, and the third conjunct could be simplified using the definition of fold_states. Using HOL, the steps to do are:

```
# e(ASM_REWRITE_TAC [FOLD_STATES]);;
# e(BETA_TAC);;
# e(ASM_REWRITE_TAC []);;

goal proved
...
Previous subproof:
"!t. out t = lambda(fold_states t delta in)"
  4  ["!t. k t = delta(h t)(in t)" ]
  3  ["!t. out t = lambda(h t)" ]
  2  ["h 0 = zero" ]
  1  ["!t. h(SUC t) = k t" ]
```

The only thing we can do in order to solve this goal, is to rewrite by assumptions:

```
# e(ASM_REWRITE_TAC []);;

"!t. lambda(h t) = lambda(fold_states t delta in)"
  4  ["!t. k t = delta(h t)(in t)" ]
  3  ["!t. out t = lambda(h t)" ]
  2  ["h 0 = zero" ]
  1  ["!t. h(SUC t) = k t" ]
```

Now, the goal is obviously true, because we are able to prove, from assumptions, that h t = fold_states t delta in. We need this lemma, in order to prove our main goal.

We construct in INTERMEDIATE_GOAL the goal, taking the assumption from the main goal, and setting a different goal formula. To prove the goal we work by induction on t; the base case is trivial since fold_states t delta in is equal to zero, and, by the second assumption, this holds also for h 0. The induction case, is equally trivial, because assumptions 1 and 4, produces the right form to apply to the inductive assumption.

The lemma could be stated and proved in the following way:

```
# let INTERMEDIATE_GOAL =
        (fst(top_goal()),"!t. h t = fold_states t delta in");;
```

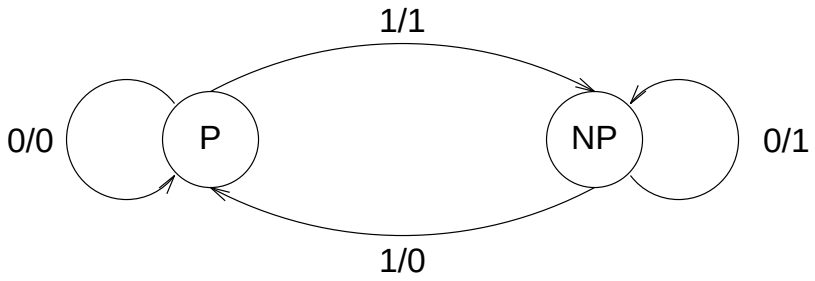


Figure 5.5: The Finite State Machine for the parity checker

```
# let INTERMEDIATE_PROOF =
        TAC_PROOF(INTERMEDIATE_GOAL,
                INDUCT_TAC THEN
                (ASM_REWRITE_TAC [FOLD_STATES]) THEN
                BETA_TAC THEN
                (ASM_REWRITE_TAC []));;

INTERMEDIATE_PROOF =
    h 0 = zero, !t. h(SUC t) = k t, !t. k t = delta(h t)(in t)
   |- !t. h t = fold_states t delta in
```

Now the last step of the main goal is a simple rewriting with our lemma:

```
# e(ASM_REWRITE_TAC [INTERMEDIATE_PROOF]);;

goal proved
```

5.3 Proving Correctness of Circuits

Now we have the right way to prove circuit correctness: we know that a generic finite state machine is equivalent to a boolean finite state machine; we know that a finite state machine is equivalent to a canonical circuit, which implements its functionality. Then, to prove that a given circuit is correct, when its specification is a finite state machine, we need to produce the canonical circuit, and to compare, with the aid of formal methods, the two circuits.

5.3.1 Example: Parity Checker

Our first example is a simple parity checker: in figure 5.5 you see the finite state machine that forms the specification, while in figure 5.6, you see the circuit we claim to implement correctly such a specification.

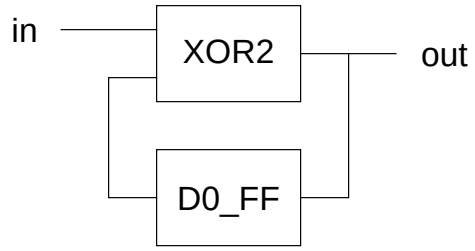


Figure 5.6: The circuit that implements the parity checker

Informally, out must be true iff the n-bit word in input contains an odd number of 1.

We use the standard definition D0_FF for the D- type flip-flop. The corresponding definition for the circuit we want to prove correct is:

```
let PARITY_IMPL = new_definition
        (‘PARITY_IMPL‘,
         "parity_impl in out =
          (? x.    d0_ff out x
                /\ (!(t:num). xor2 (in t) (x t) (out t)))");;
```

To define the canonical circuit, we need to state the actual definitions for REG, ZEROVEC and ROM2. The size of REG is 1 bit, because we have two possible states. Input and output alphabets are already expressed as boolean sets, so, the corresponding definitions for components in the canonical circuit are:

```
let ZEROVEC = new_definition
        (‘ZEROVEC‘,
         "zero = F");;

let REG = new_definition
        (‘REG‘,
         "reg in out = (out 0 = zero)
                    /\ (!t. out (SUC t) = in t)");;

let ROM2 = new_definition
        (‘ROM2‘,
          "rom2 f (in1:num -> bool)
                  (in2:num -> bool)
                  (out:num -> bool) =
                (!t. out t = f (in1 t) (in2 t))");;
```

To define the canonical circuit, we need the functions delta and lambda.

From figure 5.5, we write the formal definition of this finite state machine:

$$\langle \{P, NP\}, \{0,1\}, \{0,1\}, \delta, \lambda \rangle$$

where:

$$\delta\ x\ y = \begin{cases} P & \text{if } (x = P \wedge y = 0) \text{ or } (x = NP \wedge y = 1) \\ NP & \text{otherwise} \end{cases}$$

$$\lambda\ x\ y = \begin{cases} 0 & \text{if } (x = P \wedge y = 0) \text{ or } (x = NP \wedge y = 1) \\ 1 & \text{otherwise} \end{cases}$$

The corresponding HOL definitions, remembering that the intended initial state is P, which corresponds to $\overrightarrow{0}$, are:

```
let DELTA = new_definition
        (‘DELTA‘,
         "delta = (\x y. (x /\ ~y) \/ (~x /\ y))");;

let LAMBDA = new_definition
        (‘LAMBDA‘,
         "lambda = (\x y. (x /\ ~y) \/ (~x /\ y))");;

let PARITY_CANON_IMPL = new_definition
        (‘PARITY_CANON_IMPL‘,
         "parity_canon_impl in out =
          (? h k.   rom2 delta h in k
                 /\ rom2 lambda h in out
                 /\ reg k h)");;
```

The correctness argument is an equivalence between implementation descriptions of the two circuits. The goal and the first steps are the usual ones:

```
# set_goal([],"! in out. parity_impl in out =
                         parity_canon_impl in out");;
# e(REWRITE_TAC [PARITY_IMPL; PARITY_CANON_IMPL]);;
# e(REWRITE_TAC [DO_FF; XOR2; ROM2; REG]);;
# e(REWRITE_TAC [DELTA; LAMBDA; ZEROVEC]);;
# e(BETA_TAC);;
# e(REPEAT STRIP_TAC);;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

2 subgoals
...
```

```
"?h k.
  (!t. k t = h t /\ ~in t \/ ~h t /\ in t) /\
  (!t. out t = h t /\ ~in t \/ ~h t /\ in t) /\
  ~h 0 /\
  (!t. h(SUC t) = k t)"
  3  ["~x 0" ]
  2  ["!t. x(SUC t) = out t" ]
  1  ["!t. out t = ~(in t = x t)" ]
```

Comparing assumptions with the goal formula gives us the witness for the variable h, while k is defined in the body of the goal formula. So the steps to approach this goal are:

```
# e(EXISTS_TAC "x:num -> bool");;
# e(EXISTS_TAC "\(t:num). x t /\ ~in t \/ ~x t /\ in t");;
# e(BETA_TAC);;
# e(ASM_REWRITE_TAC []);;

"!t. ~(in t = x t) = x t /\ ~in t \/ ~x t /\ in t"
  3  ["~x 0" ]
  2  ["!t. x(SUC t) = out t" ]
  1  ["!t. out t = ~(in t = x t)" ]
```

But this goal is trivial: it is a tautology, apart the external universal quantifier, so we proceed with:

```
# e(GEN_TAC THEN TAUT_TAC);;

goal proved
...
"?x. (~x 0 /\ (!t. x(SUC t) = out t)) /\
     (!t. out t = ~(in t = x t))"
  4  ["!t. k t = h t /\ ~in t \/ ~h t /\ in t" ]
  3  ["!t. out t = h t /\ ~in t \/ ~h t /\ in t" ]
  2  ["~h 0" ]
  1  ["!t. h(SUC t) = k t" ]
```

This goal is the reverse of the preceding one: the witness for x is h, and the result of rewriting with assumption will be a tautology, so:

```
# e(EXISTS_TAC "h:num -> bool");;
# e(ASM_REWRITE_TAC []);;
```

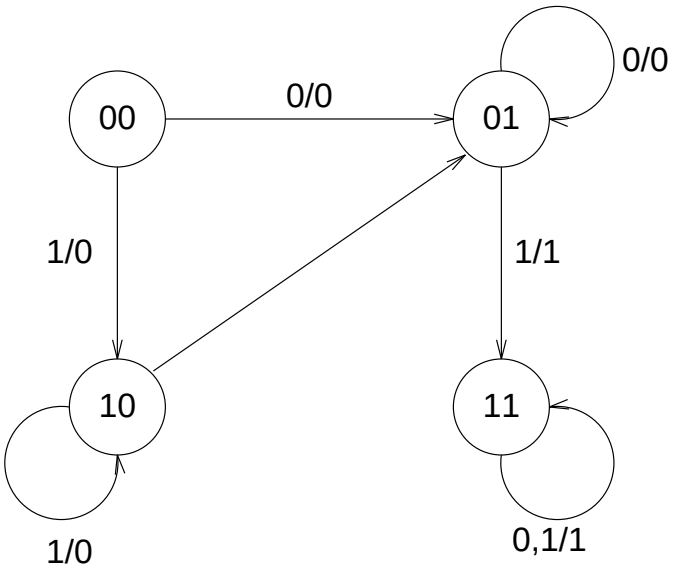


Figure 5.7: Finite State Machine for the illegal sequence detector

```
# e(GEN_TAC THEN TAUT_TAC);;

goal proved
```

5.3.2 Example: Illegal Sequence Detector

Our second example is a bit more complex. We want a circuit which, scanning sequentially a bit stream on input, gives 1 on output iff it detects the illegal sequence 01.

The finite state machine that specify such a property is shown in figure 5.7, while the implementing circuit is shown in figure 5.8.

The description of implementation is straightforward:

```
let SEQ_IMPL = new_definition
        (`SEQ_IMPL`,
         "seq_impl in out =
                (? a b c d e f g h i.
                           d0_ff i a
                        /\ d0_ff h b
                        /\ (!t. not (a t) (c t))
                        /\ (!t. and2 (b t) (c t) (d t))
                        /\ (!t. and2 (in t) (d t) (e t))
                        /\ (!t. and2 (a t) (b t) (f t))
```

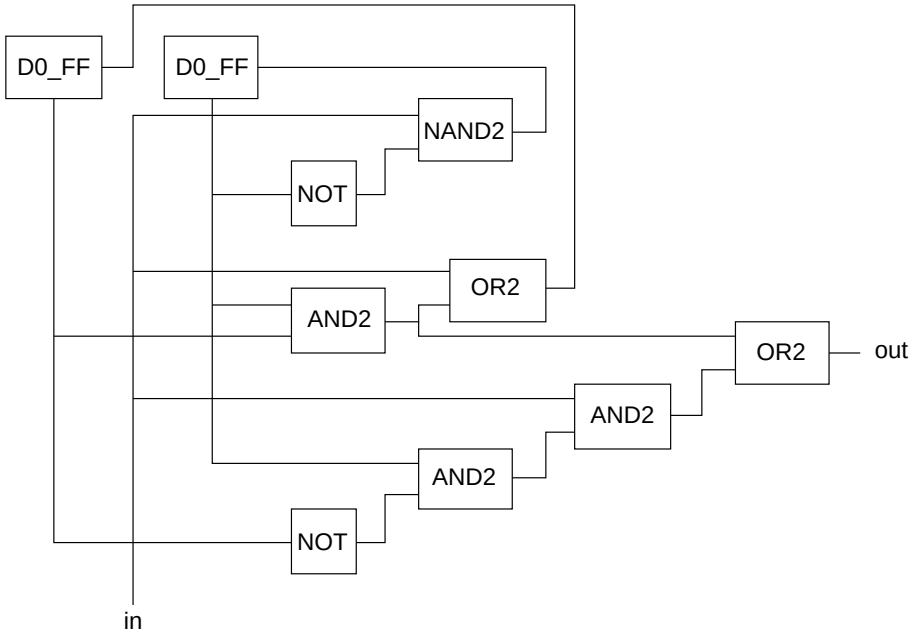


Figure 5.8: The circuit that implements the illegal sequence detector

```
                        /\ (!t. or2 (f t) (e t) (out t))
                        /\ (!t. or2 (in t) (f t) (i t))
                        /\ (!t. not (b t) (g t))
                        /\ (!t. nand2 (in t) (g t) (h t)))");;
```

As in the previous example, we define the register and the ROM according to our needs. In our particular case, we need a two bit synchronous register, and the simplest way to implement it is by means of two D-type flip-flops. In short, here are definitions:

```
let REG = new_definition
        (`REG`,
         "reg (k1,k2) (h1,h2) =
                   d0_ff k1 h1
                 /\ d0_ff k2 h2");;

let ROM2 = new_definition
        (`ROM2`,
          "rom2 f (in1:num -> *)
                  (in2:num -> **)
                  (out:num -> ***) =
                 (!t. out t = f (in1 t) (in2 t))");;
```

Now we need to define the state transition function and the output function. δ function must produce a state. We choose to represent a state as a pair of bits. From the figure 5.7, we obtain the following definitions of these functions:

```
let SEQ_DELTA = new_definition
        (‘SEQ_DELTA‘,
         "seq_delta (a1,a2) in =
                ((in \/ (~in /\ a1 /\ a2)),
                 (~((in /\ ~a1 /\ ~a2) \/
                    (in /\ a1 /\ ~a2))))");;

let SEQ_LAMBDA = new_definition
        (‘SEQ_LAMBDA‘,
         "seq_lambda (a1,a2) in =
                   (~in /\ a1 /\ a2)
                 \/ (in /\ ~a1 /\ a2)
                 \/ (in /\ a1 /\ a2)");;
```

And the corresponding canonical circuit will be:

```
let SEQ_CANON_IMPL = new_definition
        (‘SEQ_CANON_IMPL‘,
         "seq_canon_impl in out =
          (? h k.   rom2 seq_delta h in k
                 /\ rom2 seq_lambda h in out
                 /\ reg ((\t. FST (k t)),(\t. SND (k t)))
                        ((\t. FST (h t)),(\t. SND (h t))))");;
```

The correctness argument is an equivalence between the two circuit descriptions. The first proof steps are the usual ones, and lead to:

```
# set_goal([],"! in out. seq_impl in out =
                         seq_canon_impl in out");;
# e(REWRITE_TAC [SEQ_IMPL; SEQ_CANON_IMPL]);;
# e(REWRITE_TAC [DO_FF; NOT; AND2; OR2; NAND2; ROM2; REG]);;
# e(BETA_TAC);;
# e(REPEAT STRIP_TAC);;
# e(EQ_TAC THEN REPEAT STRIP_TAC);;

2 subgoals
...
```

```
"?h k.
  (!t. k t = seq_delta(h t)(in t)) /\
  (!t. out t = seq_lambda(h t)(in t)) /\
  (~FST(h 0) /\ (!t. FST(h(SUC t)) = FST(k t))) /\
  ~SND(h 0) /\
  (!t. SND(h(SUC t)) = SND(k t))"
  12 ["~a 0" ]
  11 ["!t. a(SUC t) = i t" ]
  10 ["~b 0" ]
  9  ["!t. b(SUC t) = h t" ]
  8  ["!t. c t = ~a t" ]
  7  ["!t. d t = b t /\ c t" ]
  6  ["!t. e t = in t /\ d t" ]
  5  ["!t. f t = a t /\ b t" ]
  4  ["!t. out t = f t \/ e t" ]
  3  ["!t. i t = in t \/ f t" ]
  2  ["!t. g t = ~b t" ]
  1  ["!t. h t = ~(in t /\ g t)" ]
```

From the circuit we get the witness for h. Because h represents the next state line in the canonical circuit, and a and b have the same purpouse in our circuit, then we combine them to instantiate h. The witness for k is in the second line of the goal formula. Then we rewrite using definitions of state transition function and of output function. The result follows:

```
# e(EXISTS_TAC "\t.((a:num -> bool) t, (b:num -> bool) t)");;
# e(BETA_TAC);;
# e(EXISTS_TAC "\(t:num). seq_delta(a t,b t)(in t)");;
# e(BETA_TAC);;
# e(ASM_REWRITE_TAC [SEQ_LAMBDA; SEQ_DELTA]);;

"(!t.
   a t /\ b t \/ in t /\ b t /\ ~a t =
   ~in t /\ a t /\ b t \/
    in t /\ ~a t /\ b t \/
    in t /\ a t /\ b t) /\
 (!t. in t \/ a t /\ b t = in t \/ ~in t /\ a t /\ b t) /\
 (!t. ~(in t /\ ~b t) =
      ~(in t /\ ~a t /\ ~b t \/ in t /\ a t /\ ~b t))"
  12 ["~a 0" ]
  11 ["!t. a(SUC t) = i t" ]
  10 ["~b 0" ]
  9  ["!t. b(SUC t) = h t" ]
```

```
  8  ["!t. c t = ~a t" ]
  7  ["!t. d t = b t /\ c t" ]
  6  ["!t. e t = in t /\ d t" ]
  5  ["!t. f t = a t /\ b t" ]
  4  ["!t. out t = f t \/ e t" ]
  3  ["!t. i t = in t \/ f t" ]
  2  ["!t. g t = ~b t" ]
  1  ["!t. h t = ~(in t /\ g t)" ]
```

This goal is almost propositional and could be solved using the tautology checkers' conversion.

```
# e(CONV_TAC (ONCE_DEPTH_CONV TAUT_CONV));;
# e(ASM_REWRITE_TAC []);;

goal proved
...
Previous subproof:
"?a b c d e f g h i.
  (~a 0 /\ (!t. a(SUC t) = i t)) /\
  (~b 0 /\ (!t. b(SUC t) = h t)) /\
  (!t. c t = ~a t) /\
  (!t. d t = b t /\ c t) /\
  (!t. e t = in t /\ d t) /\
  (!t. f t = a t /\ b t) /\
  (!t. out t = f t \/ e t) /\
  (!t. i t = in t \/ f t) /\
  (!t. g t = ~b t) /\
  (!t. h t = ~(in t /\ g t))"
  6  ["!t. k t = seq_delta(h t)(in t)" ]
  5  ["!t. out t = seq_lambda(h t)(in t)" ]
  4  ["~FST(h 0)" ]
  3  ["!t. FST(h(SUC t)) = FST(k t)" ]
  2  ["~SND(h 0)" ]
  1  ["!t. SND(h(SUC t)) = SND(k t)" ]
```

The witnesses for a and b are the two components of the state on h. We know this fact comparing the two circuits. All other witnesses are present in the goal formula, so we could eliminate them using EXISTS_TAC enough times.

```
# e(EXISTS_TAC "\t. FST ((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t. SND ((h:num -> bool # bool) t)");;
```

```
# e(EXISTS_TAC "\t.~FST ((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t. SND ((h:num -> bool # bool) t) /\
                  ~FST((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t. in t /\ SND ((h:num -> bool # bool) t) /\
                  ~FST ((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t. FST ((h:num -> bool # bool) t) /\
                  SND ((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t.~SND ((h:num -> bool # bool) t)");;
# e(EXISTS_TAC "\t. ~(in t /\
                      ~SND ((h:num -> bool # bool) t))");;
# e(EXISTS_TAC "\t. in t \/ FST ((h:num -> bool # bool) t)
                       /\ SND ((h:num -> bool # bool) t)");;
# e(BETA_TAC);;

"(~FST(h 0) /\
   (!t. FST(h(SUC t)) = in t \/ FST(h t) /\ SND(h t))) /\
 (~SND(h 0) /\
   (!t. SND(h(SUC t)) = ~(in t /\ ~SND(h t)))) /\
 (!t. ~FST(h t) = ~FST(h t)) /\
 (!t. SND(h t) /\ ~FST(h t) = SND(h t) /\ ~FST(h t)) /\
 (!t. in t /\ SND(h t) /\ ~FST(h t) =
      in t /\ SND(h t) /\ ~FST(h t)) /\
 (!t. FST(h t) /\ SND(h t) = FST(h t) /\ SND(h t)) /\
 (!t. out t = FST(h t) /\ SND(h t) \/
              in t /\ SND(h t) /\ ~FST(h t)) /\
 (!t. in t \/ FST(h t) /\ SND(h t) =
      in t \/ FST(h t) /\ SND(h t)) /\
 (!t. ~SND(h t) = ~SND(h t)) /\
 (!t. ~(in t /\ ~SND(h t)) = ~(in t /\ ~SND(h t)))"
  6  ["!t. k t = seq_delta(h t)(in t)" ]
  5  ["!t. out t = seq_lambda(h t)(in t)" ]
  4  ["~FST(h 0)" ]
  3  ["!t. FST(h(SUC t)) = FST(k t)" ]
  2  ["~SND(h 0)" ]
  1  ["!t. SND(h(SUC t)) = SND(k t)" ]
```

The next step will be rewriting with assumptions, but there is problem. We chose to represent states as pair of booleans. In particular, we have assumptions 5 and 6 which depend on the value of $h\ t$. This value is a pair, and our definitions of seq_delta and seq_lambda are given with pairs. But HOL is not able to expand definitions because $h\ t$ has not the form of a pair.

The axiom $PAIR$ states that, if x has type $pair$ then x is equivalent to $FST(x), SND(x)$. We need to create a type specific, reversed version of this rewrite rule. We can do it in the following way:

```
# let BOOLPAIR = INST_TYPE [(":bool",":*");
                            (":bool",":**")] PAIR;;
# let H_AS_PAIR = SYM (SPEC "(h:num -> bool # bool) t"
                       BOOLPAIR);;

BOOLPAIR = |- !x. FST x,SND x = x
H_AS_PAIR = |- h t = FST(h t),SND(h t)
```

Now let's rewrite with assumptions to simplify the goal, and then with our lemma, but only once, because it produces cycles in the HOL rewriting algorithm. The result follows:

```
# e(ASM_REWRITE_TAC []);;
# e(ONCE_REWRITE_TAC [H_AS_PAIR]);;

"(!t.
   FST(seq_delta(FST(h t),SND(h t))(in t)) =
   in t \/ FST(FST(h t),SND(h t)) /\
           SND(FST(h t),SND(h t))) /\
 (!t.
   SND(seq_delta(FST(h t),SND(h t))(in t)) =
   ~(in t /\ ~SND(FST(h t),SND(h t)))) /\
 (!t.
   seq_lambda(FST(h t),SND(h t))(in t) =
   FST(FST(h t),SND(h t)) /\ SND(FST(h t),SND(h t)) \/
   in t /\ SND(FST(h t),SND(h t)) /\ ~FST(FST(h t),SND(h t)))"
  6  ["!t. k t = seq_delta(h t)(in t)" ]
  5  ["!t. out t = seq_lambda(h t)(in t)" ]
  4  ["~FST(h 0)" ]
  3  ["!t. FST(h(SUC t)) = FST(k t)" ]
  2  ["~SND(h 0)" ]
  1  ["!t. SND(h(SUC t)) = SND(k t)" ]
```

Now we can rewrite with definitions of transition and output functions. The result will be an almost propositional formula, and we know how to solve it:

```
# e(REWRITE_TAC [SEQ_LAMBDA; SEQ_DELTA]);;
# e(CONV_TAC (ONCE_DEPTH_CONV TAUT_CONV));;
# e(ASM_REWRITE_TAC []);;

goal proved
```

Chapter 6

Overview of Microprocessors Verification

In this chapter we will examine how to verify microprocessors. We will not show you a complete example, nor we will give you an established methodology, because there are few examples, and the topic is largely a research field. Instead we will try to give you some basic ideas and we will point out major problems. Using formal methods on microprocessors is an exciting task, but it is a very difficult one.

6.1 Different Levels of Description

Microprocessors are complex objects. In previous chapters we described simple circuits: their behaviour and their structure could be directly related. The only exception we found was with finite state machines.

From an abstract point of view, when we have a finite state state machine that specify the behaviour of a circuit, we have two description levels: the canonical circuit and the real one. Proving correctness of those circuits implies two steps: proving that the canonical circuit correctly maps the given specification (i.e. the finite state machine), and proving that the real circuit is equivalent to the canonical one, which act as a specification.

With microprocessors the situation is similar, but much more complex. Let's look at figure 6.1.

We have two basic entities: a circuit, and a specification of the microprocessor's behaviour.

The circuit itself is nothing special: wires, gates, registers, ..., in a word, the usual components.

The specification describes what the microprocessor has to do when it processes an instruction, when it receives an interrupt or when it accesses memory or devices. It says things that an assembler programmer can understand because they are what he can observe,

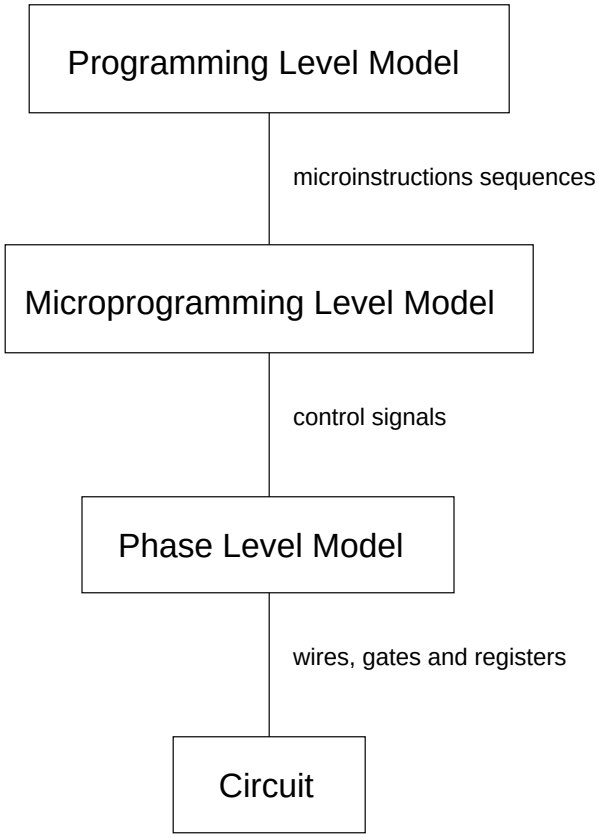


Figure 6.1: Description Levels

We can abstract this view saying that a microprocessor is an abstract machine which implements instructions' and interrupts' semantics.

Internally, we can think what implements that abstract machine (we call it *Programming Level Model*) is another abstract machine, whose instructions are the microinstructions. We have reached the microprogramming level model.

If we design a machine which is composed by components like the ALU, the memory interface, ..., and the control unit, which say what component to activate and when, we are designing another model, the phase level model.

Each component in the phase level model is a circuit, so we have reached the target.

Let's take a look at each level-to-level transition:

- **From the original specification to the programming level model**.
  We have to prove that this level satisfies semantics of instructions (i.e. ADD performs a sum), we have to prove that, when an interrupt occurs, the right sequence of actions is taken (for example, switch from user mode to supervisor mode), we have to prove that memory or devices are accessed or access the microprocessor when they have to do so.

- **From the programming level to the microprogramming level**.
  We have to prove that sequences of microinstructions correctly implement machine language instructions and interrupts, and we have to prove that memory and devices communicate in the right way with the microcode interpreter.

- **From the microprogramming level to the phase level**.
  The third step is to prove that the microcode interpreter is correctly implemented by the control unit, and that every component performs the right operations in the right way.

- **From the phase level to the real circuit**.
  The final step is to prove that the real circuit contains the correct implementations of the various components at the phase level, and that signals between components are correctly mapped into hardware signals through appropriate wires.

With these guidelines in mind, we can approach the problem in a more technical way, showing how to describe objects at various level, and how to relate them.

6.2 The Programming Level Model

In this section we present a way to describe the programming level model. This model can be viewed as an interpreter for manipulating a set of variables which corresponds to the externally visible state of a microprocessor.

In this section, and in the next ones, we will use as examples parts of the specification of the TAMARACK-3 microprocessor (see [37]).

6.2.1 Basic Datatypes and Primitive Operations

To specify the programming level model we need only two datatypes: boolean values and natural numbers. They are predefined in the HOL system as the $:\mathrm{bool}$ and $:\mathrm{num}$ types.

To describe the model we need different datatypes. Data bus has a fixed size, address bus has another fixed size, and memory is a finite resource. Experience tells that is convenient not to fix sizes at this description level. So we define the following generic types:

$:*wordn$	full size machine words
$:*address$	memory addresses
$:*memory$	memory states

We could assume an instruction is represented exactly by one word. But this is not always true. For example, in the TAMARACK-3 microprocessor an instruction occupies one word, but only 3 bits are used to represent an

instruction itself, while others are used to represent the operand. So we define another datatype which represents the instruction opcode, without operands:

$$: *wordop$$

To deal with these datatypes we need functions to convert values from one to the other.

The relation between words and integers is carried on by the following functions:

$$\begin{aligned} wordn &: num \longrightarrow *wordn \\ valn &: *wordn \longrightarrow num \end{aligned}$$

To extract opcodes and to simplify their treatment we define:

$$\begin{aligned} opcode &: *wordn \longrightarrow *wordop \\ valop &: *wordop \longrightarrow num \end{aligned}$$

Memory, from microprocessor's point of view, is accessible only through the following functions:

$$\begin{aligned} fetch &: (*memory \times *address) \longrightarrow *wordn \\ store &: (*memory \times *address \times *wordn) \longrightarrow *memory \end{aligned}$$

In most microprocessors we must define multiple fetch and store functions, because they can access memory using multiples or fractions of a $*wordn$.

The last function converts a $*wordn$ into an $*address$.

$$address: *wordn \longrightarrow *address$$

In general, microprocessors encode addresses in multiples of $*wordn$. We defined the address function thinking to TAMARACK-3 example, but in a MC68020 the address function will be:

$$address: (*wordn \times *wordn) \longrightarrow *address$$

With these functions we can define a map between values in the specification and values in the programming level model.

6.2.2 Describing the Machine State

From a programmer's point of view a microprocessor's state is fully described by memory, registers (accumulators, for example), the program counter and interrupt line's values.

In the case of TAMARACK-3, an address fits into a $*wordn$, so the program counter is represented by a single word. The microprocessor has two registers: an accumulator (acc) and a return address register (rtn). There is a single interrupt acknowledging line, $iack$. So a microprocessor state could be described by values of the following variables:

$$\begin{aligned} mem &: *memory \\ pc &: *wordn \\ acc &: *wordn \\ rtn &: *wordn \\ iack &: bool \end{aligned}$$

In a more complex microprocessor we will have a similar description, but with more variables, and, perhaps, different types. For example, the state register of a MC68000 could be defined as a variable:

$$\text{CCR}: (\text{bool} \times \text{bool} \times \text{bool} \times \text{bool} \times \text{bool})$$

and it will be convenient to define the following abbreviations:

$$\begin{array}{lcl} \text{CCR\_carry} & = & \text{first}(\text{CCR}) \\ \text{CCR\_overflow} & = & \text{second}(\text{CCR}) \\ \text{CCR\_zero} & = & \text{third}(\text{CCR}) \\ \text{CCR\_negative} & = & \text{fourth}(\text{CCR}) \\ \text{CCR\_extend} & = & \text{fifth}(\text{CCR}) \end{array}$$

6.2.3 Semantics of the Instruction Set

Every instruction of the microprocessor is uniquely associated with an opcode. In the Programming Level Model, we give to every instruction an appropriate semantics. It describes what a single instruction do in the abstract machine.

To do so, we need some definitions, and we have to solve some problems. The first problem is to specify in HOL what are elementary operations we will use to describe behaviour of Programming Level Model. The second problem is how to specify what are the assumptions we make on the general model.

Let's begin to work on the first problem: we need to specify the parameters from which our representation depends. We may define a *representation type*. In the TAMARACK-3 example, we can express this type as:

```
let PLM_rep_ty =
    "(*wordn -> bool)                          #   % iszero  %
     (*wordn -> *wordn)                        #   % inc     %
     (*wordn # *wordn -> *wordn)               #   % add     %
     (*wordn # *wordn -> *wordn)               #   % sub     %
     (num -> *wordn)                           #   % wordn   %
     (*wordn -> num)                           #   % valn    %
     (*wordn -> *wordop)                       #   % opcode  %
     (*wordop -> num)                          #   % valop   %
     (*wordn -> *address)                      #   % address %
     (*memory # *address -> *wordn)            #   % fetch   %
     (*memory # *address # *wordn -> *wordn)";;  % store   %
```

obviously we can define basic functions as projection functions of the representation type.

For example:

```
let inc = new_definition
          (‘inc‘,
           "inc (rep:^PLM_rep_ty) = FST(SND rep)");;

let wordn = new_definition
            (‘wordn‘,
             "wordn (rep:^PLM_rep_ty) =
                    FST(SND(SND(SND(SND(SND rep)))))’’);;
```

Let's examine the representation type: it says that the way our microprocessor encodes integers is expressed by $valn$ and $wordn$ fields; the way an instruction is represented in a machine word is defined by the $opcode$, $valop$ and $address$ fields; the memory interface is defined by the particular instance of the $fetch$ and $store$ fields.

In the same way, the basic functions performed by ALU are defined by values from the first four fields.

The second problem is what kind of axioms have to limit the way we could instantiate the various fields in PLM_rep_ty.

Lower descriptions levels of the microprocessor will limit $iszero$, inc, add and sub. Number coding is not a problem, because nothing limit it in the Programming Level Model. The same consideration holds for memory addresses. But there is a very precise limit to opcodes representation; it must respect the following axiom:

$$\forall(x{:}*wordop).valop\ x < number\ of\ instructions$$

Using HOL syntax:

```
let Valop_CASES_AX = new_definition
    (‘Valop_CASES_AX‘,
     "Valop_CASES_AX (rep:^PLM_rep_ty) =
                 ! w. ((valop rep) w) < instruction_number");;
```

In the TAMARACK-3 example, we add the following definition, after a brief look to table 6.2:

```
let inst_number_AX = new_definition
    (‘inst_number_AX‘,
     "instruction_number = 8");;
```

To define semantics for every instruction is useful to remember that the current instruction is:

$$inst = fetch(mem, (address\ pc))$$

Instruction	Opcode	Effect
JZR	0	jump if zero
JMP	1	jump
ADD	2	add accumulator
SUB	3	subtract accumulator
LDA	4	load accumulator
STA	5	store accumulator
RFI	6	return from interrupt
NOP	7	no operation

Figure 6.2: TAMARACK-3 Instruction Set

while the addressed operand is

$$operand = \mathrm{fetch}(mem, (address\ inst))$$

TAMARACK-3 has only absolute addressing, so the definition of operand is unique.

In general we could have something like:

$operand\_abs =$ $\mathrm{fetch}\ (mem, (address\ inst))$ absolute

$operand\_ind =$ $\mathrm{fetch}\ (mem, (address$ $\mathrm{fetch}\ (mem, (address\ inst))))$ indirect

$operand\_dX =$ $\mathrm{fetch}\ (mem, (address$ $(\mathrm{add}\ X\ (addrtowordn$ $(address\ inst)))))$ indexed by register X

$operand\_imm =$ $addrtowordn(address\ inst)$ immediate

where $addrtowordn$ converts an address into the numerical representation used by the microprocessor.

With these elements we are able to define semantics of instructions. Here are some examples:

```
let ADD_SEM = new_definition
    (‘ADD_SEM‘,
     "ADD_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
        let inst    = (fetch rep)(mem, (address rep) pc) in
        let operand = (fetch rep)(mem, (address rep) inst) in
            (mem, (inc rep) pc,
             (add rep)(acc, operand), rtn, iack)");;
```

```
let JZR_SEM = new_definition
    (`JZR_SEM`,
     "JZR_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
        let inst   = (fetch rep)(mem, (address rep) pc) in
        let nextpc = ((iszero rep) acc) => inst |
                                           ((inc rep) pc) in
            (mem, nextpc, acc, rtn, iack)");;

let RFI_SEM = new_definition
    (`RFI_SEM`,
     "RFI_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
            (mem, rtn, acc, rtn, F)");;
```

6.2.4 Hardware Interrupts

In the normal flow of program execution, instructions are sequentially executed, according to their semantics.

Interrupts are events which modify this sequential behaviour. At this level of description, internal interrupts, the so called *traps*, are not interesting, they are part of semantics of instructions. But external interrupts must be modeled.

In the TAMARACK-3, we have only one kind of interrupt: it is a single level, non-vectored, non-maskable hardware interrupt. It is processed after the instruction the microprocessor is processing is completed.

To model this kind of interrupt with HOL, we need the following definition:

```
let IRQ_SEM = new_definition
    (`IRQ_SEM`,
     "IRQ_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
            (mem, ((wordn rep) 0), acc, pc, T)");;
```

Its meaning is: when an interrupt is processed, the program counter is set to 0, the rtn register takes the value of the program counter pc, and the iack line line is set to true.

The microprocessor will process the interrupt handler, whose code begins at address 0 in the memory, and, when it finishes, it can return to the interrupt point using the RFI instruction. The iack line high prevents interrupts during this process.

We can follow this schema also to model more general interrupts.

Vectored interrupts are modeled by changing the pc register to the content of the vector:

```
let IRQ_SEM = new_definition
    (‘IRQ_SEM‘,
     "IRQ_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
            (mem, (fetch rep)(mem, irq_vec), acc, pc, T)");;
```

where irq_vec is the address of the vector for IRQ interrupt.

Masking is implemented in the general instruction flow, we will describe in the next section.

Multiple levels of interrupts are a bit more difficult. If we assume that rtn is an address that points to the top of the interrupt stack, we can describe a multiple level IRQ interrupt in the following way:

```
let IRQ_SEM = new_definition
    (‘IRQ_SEM‘,
     "IRQ_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
         ((store rep) (mem, (address rep)((inc rep) rtn), pc),
          (wordn rep) 0, acc, (inc rep) rtn, T)");;
```

and we have to modify the RFI semantics:

```
let RFI_SEM = new_definition
    (`RFI_SEM`,
     "RFI_SEM (rep :^PLM_rep_ty)
              (mem :*memory,
               pc  :*wordn,
               acc :*wordn,
               rtn :*wordn,
               iack:bool) =
         (mem, (fetch rep)(mem, (address rep) rtn),
          acc, (sub rep)(rtn, (wordn rep) 1),
          (rtn = (wordn rep) 0))");;
```

The meaning is: when an IRQ occurs, we store the current value of pc in the memory location whose address is in $\mathrm{rtn}+1$, the new stack pointer is $\mathrm{rtn}+1$, and we tell to external hardware that an interrupt is being processed, setting iack to true. When an RFI instruction occurs, we pop the pc value from the top of the stack, we decrement rtn, and, if all interrupts are processed, we set iack to false.

6.2.5 Memory Interface and General Behaviour

Now we have described all pieces that a low-level programmer can see. Our microprocessor's programming level model seems to be fully implemented. But this is not true. We need to specify how these pieces interact and we have to describe how the interface between memory and CPU works.

Let's begin with the general behaviour. It is defined by saying that the next state is a function of the current one:

```
let TamarackBehaviour = new_definition
    (`TamarackBehaviour`,
     "TamarackBehaviour (rep:^PLM_rep_ty)
                        (ireq, mem, pc, acc, rtn, iack) =
        ! u:time
          (mem (u + 1), pc (u + 1), acc (u + 1),
           rtn (u + 1), iack (u + 1)) =
          NextState rep (ireq u, mem u, pc u, acc u,
                         rtn u, iack u)");;
```

The *NextState* function computes a new microprocessor state, given the current one. It simply analyzes conditions that generate interrupts, and, if the case, generate them, otherwise get the opcode of the instruction pointed by pc, and executes it.

In the TAMARACK-3 example, *NextState* is defined as:

```
let NextState = new_definition
    (‘NextState‘,
     "NextState (rep:^PLM_rep_ty)
                (ireq, mem, pc, acc, rtn, iack) =
      let opc = OpcVal rep (mem, pc) in
        (ireq /\ ~iack) => IRQ_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = JZR_OPC) => JZR_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = JMP_OPC) => JMP_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = ADD_OPC) => ADD_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = SUB_OPC) => SUB_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = LDA_OPC) => LDA_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = STA_OPC) => STA_SEM rep (mem,pc,acc,rtn,iack) |
        (opc = RFI_OPC) => RFI_SEM rep (mem,pc,acc,rtn,iack) |
                           NOP_SEM rep (mem,pc,acc,rtn,iack)");;
```

We want to develop two considerations:

- Programming Level Model hides effective time, because every instruction seems to be executed in one time unit, while different instructions could take a different number of clock cycles to complete their operations.
- Interrupts are revealed and actioned in NextState. Here we could specify different policies to manage interrupts, such as masking.

To complete the Programming Level Model description, we need to specify a memory model, that is, we have to describe the behaviour of the memory device. It is used to relate the implementation of lower levels with the model we have developed till now. TAMARACK-3 implements a very simple memory access schema: access is synchronous, and it is specified using microprocessor's lines as variables. The HOL form for memory specification is:

```
let SynMemory = new_definition
    (‘SynMemory‘,
     "SynMemory (rep:^PLM_rep_ty)
                (w, addr, dataout, mem, datain) =
      ! t:time.
        (~(w t) ==>
                (datain t = (fetch rep)(mem t, addr t))) /\
        (mem (t + 1) =
           (w t => (store rep)(mem t, addr t, dataout t) |
                    mem t))");;
```

The meaning of this definition is: for every clock cycle, if the microprocessor does not request a write operation, then its $datain$ bus is set to the value of the $addr$ memory cell, and, if a write operation is requested, then the value on $dataout$ bus is stored in the $addr$ location.

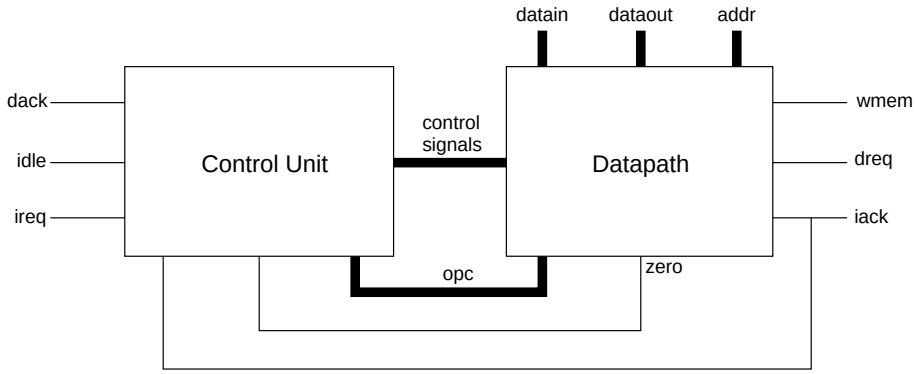


Figure 6.3: Structural Model of TAMARACK-3

We will not describe more complex memory models because they are very difficult. Our approach is not perfect: some things, and the memory model is one of them, are really difficult to describe, and there are no general guidelines to help you.

6.3 The Microprogramming Level Model

In this model, a programming level instruction is interpreted by executing a sequence of microinstructions. This level is more structured than the previous one: we will distinguish a part (the Control Unit) which generates and executes microinstructions, from another part (the Datapath) which really performs operations.

The communication between them takes place in the form of control signals.

At this level some internal registers become visible, and the microprocessor's state becomes more complex.

6.3.1 Structural Model

At the microprogramming level model, a microprocessor is divided into two main parts: a control unit and a datapath.

The datapath is a component which manipulates data, while the control unit is a component which decides what to do.

To do its work, the control unit needs to know, at appropriate time, what instruction is to be processed (opc), the state of some internal flags ($zero$), some synchronization signals from the outside world ($dack$, $idle$, $iack$ and $ireq$).

Using these inputs it generates microinstructions that tell the microprocessor what to do: it executes its part of each microinstruction and sends the rest ($control\ signals$) to the datapath.

The datapath activates some of its components in response to a set of control signals generated by the control unit.

As we will see later, the datapath contains the system bus, ALU, memory interface and all registers. Because at this level (and in lower ones) is difficult to generalize, we will focus only on the TAMARACK-3 example.

6.3.2 States

At the programming level, the microprocessor's state was described by five variables: mem, pc, acc, rtn and iack.

To deal with interrupts we added another variable, ireq, which signals a pending request. More variables were added when we described the memory interface.

Now we have two components: each of them has its own inputs, outputs and internal state.

So there is not a *global* state, but each component has its own and here we want to describe the variables which hold values that are part of.

Inputs of the control unit are shown in figure 6.3. They are:

$$\begin{aligned} dack &: bool \\ idle &: bool \\ ireq &: bool \\ iack &: bool \\ zero &: bool \\ opc &: *wordop \end{aligned}$$

Its output is part of a microinstruction: it could be defined as a triple where the first element specify what datapath register to read from, the second element specify the datapath register to write to, and the third element specify the operation to perform. We will describe later the exact format of a microinstruction, because we need to introduce variables that define the datapath.

The datapath takes its input from the control unit and from datain, the input side of the microprocessor's databus. The variable datain:$*wordn$ is used to denote its input. The outputs the datapath generates, are:

$$\begin{aligned} wmem &: bool \\ dataout &: *wordn \\ dreq &: bool \\ addr &: *address \\ iack &: bool \\ zero &: bool \\ opc &: *wordop \end{aligned}$$

Internal registers of the datapath are the ones of the programming level model (pc, acc, rtn and mem), plus the following:

$$\begin{aligned} mar &: *wordn \\ ir &: *wordn \\ arg &: *wordn \\ buf &: *wordn \end{aligned}$$

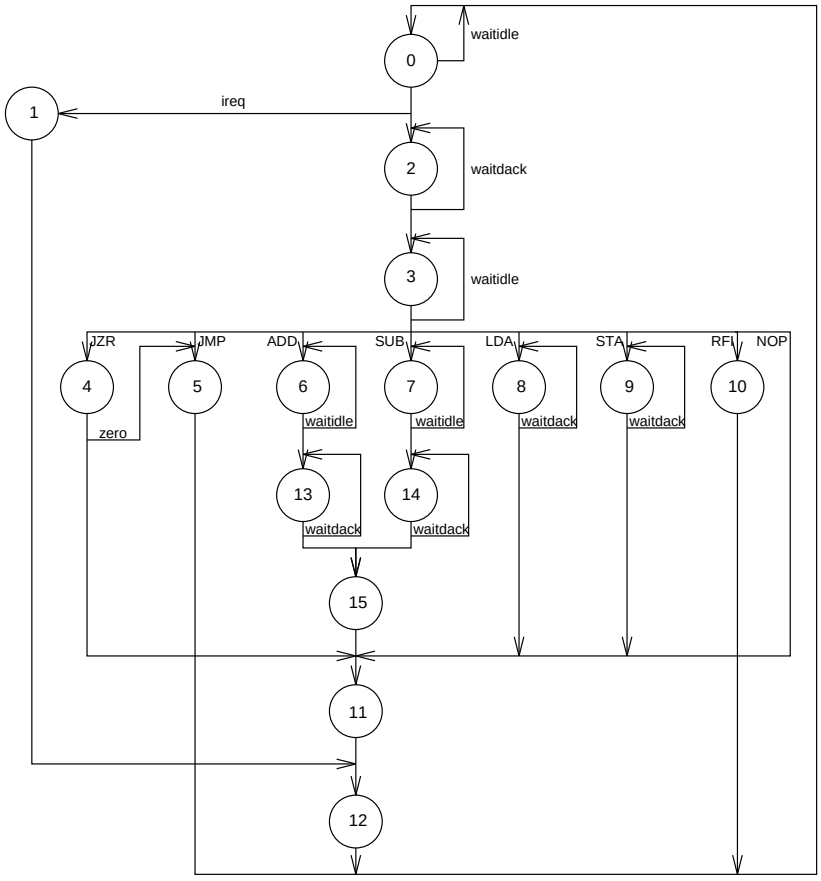


Figure 6.4: Control Unit Finish State Machine Flow Diagram

Memory interface is connected to microprocessor through the variables that are part of the states of control unit and datapath. We omitted time in the definition of state variables because we assume that the control unit and the datapath are functions which take as input the current state, and generate as output the next one; we are not specifying time because, at this level, our description is totally functional.

6.3.3 The Control Unit

The control unit can be described as a finite state machine. It is a Moore machine, i.e., outputs are associated with states, not with inputs and states. Figure 6.4 shows the diagram of the finite state machine.

State 0 is the initial state; when an interrupt occurs, there is a transition to state 1; if an instruction is ready to be fetched, there is a transition to state 2. States 2 and 3 are responsible for the fetch instruction phase. States 4 to 10 plus 13, 14 and 15 process a programming level instruction. Here to process means to generate microcode.

State 11 add 1 to the program counter, while state 12 set the program counter to the new value.

Sometimes a state is maintained until some synchronization signal arrives. In the diagram these conditions are represented as loops. The label $waitdack$ stands for the condition $\neg \mathrm{dack}$, and the label $waitidle$ stands for $\neg(\mathrm{idle} \vee \neg \mathrm{dack})$.

In state 3 the next state in chosen according to the opc input.

Every state generates exactly one microinstruction and it is immediately divided into two parts: the first one is given to datapath, and it specifies what to do; the second part is executed by control unit itself, and it specifies what is the next state of the finite state machine, according to the current input.

6.3.4 Microinstructions

At this level of description, a microinstruction is a tuple. More precisely, a microinstruction is composed by four fields:

- datapath instructions
- loop condition
- how to compute the next state of the control unit
- control unit addresses

Let's begin describing the possible values for the loop condition field:

$$\begin{aligned} waitidle &: \text{repeat current state if } \neg(\mathrm{idle} \vee \neg \mathrm{dack}) \\ waitdack &: \text{repeat current state if } \neg \mathrm{dack} \\ continue &: \text{do not repeat current state} \end{aligned}$$

When a repeat condition is not satisfied (i.e. the current state has not to be repeated), the third and the fourth fields are used to compute the next state of the control unit.

More precisely, the fourth element is a pair of state numbers. The meaning of the third field, along with its possible values, is (where $\langle a, b \rangle$ is the value of the fourth field):

$$\begin{aligned} jump &: \text{go to state } a \\ jireq &: \text{if ireq then go to state } a \text{ else go to state } b \\ jzero &: \text{if zero then go to state } a \text{ else go to state } b \\ jopcode &: \text{go to state } a + \mathrm{opc} \end{aligned}$$

The control unit operates as follows:

- when a state is entered, the corresponding microinstruction is generated.
- the control unit processes the second field of the microinstruction, and, concurrently, it sends the first field to the datapath.

- if the loop condition is not satisfied (i.e. if no loop occurs), then the control unit uses the third and the fourth fields to make a transition to a new state.

The first field of a microinstruction says to datapath what to do. It is a triple $\langle r, w, x \rangle$ where the first element specifies the register to read from, the second element specifies the register to write to, and the x field specifies the ALU operation to perform.

The possible values for the first field are:

$$rpc \quad rmem \quad rir \quad racc \quad rrtn \quad rbuf \quad none$$

The possible values for the write field are:

$$wmar \quad wrtm \quad wir \quad wpc \quad warg \quad wacc \quad wmem \quad none$$

The possible values for the third field are:

$$zero \quad inc \quad add \quad sub \quad none$$

We said that every state of the control unit's finite state machine generates a microinstruction. Here is the complete listing:

state	microinstruction
0	$\langle\langle rpc, wmar, none\rangle, waitidle, jireq, \langle 1, 2\rangle\rangle$
1	$\langle\langle rpc, wrtn, zero\rangle, continue, jump, \langle 12, 0\rangle\rangle$
2	$\langle\langle rmem, wir, none\rangle, waitdack, jump, \langle 3, 0\rangle\rangle$
3	$\langle\langle rir, wmar, none\rangle, waitidle, jopcode, \langle 4, 0\rangle\rangle$
4	$\langle\langle none, none, none\rangle, continue, jzero, \langle 5, 11\rangle\rangle$
5	$\langle\langle rir, wpc, none\rangle, continue, jump, \langle 0, 0\rangle\rangle$
6	$\langle\langle racc, warg, none\rangle, waitidle, jump, \langle 13, 0\rangle\rangle$
7	$\langle\langle racc, warg, none\rangle, waitidle, jump, \langle 14, 0\rangle\rangle$
8	$\langle\langle rmem, wacc, none\rangle, waitdack, jump, \langle 11, 0\rangle\rangle$
9	$\langle\langle racc, wmem, none\rangle, waitdack, jump, \langle 11, 0\rangle\rangle$
10	$\langle\langle rrtn, wpc, none\rangle, continue, jump, \langle 0, 0\rangle\rangle$
11	$\langle\langle rpc, none, inc\rangle, continue, jump, \langle 12, 0\rangle\rangle$
12	$\langle\langle rbuf, wpc, none\rangle, continue, jump, \langle 0, 0\rangle\rangle$
13	$\langle\langle rmem, none, add\rangle, waitdack, jump, \langle 15, 0\rangle\rangle$
14	$\langle\langle rmem, none, sub\rangle, waitdack, jump, \langle 15, 0\rangle\rangle$
15	$\langle\langle rbuf, wacc, none\rangle, continue, jump, \langle 11, 0\rangle\rangle$

Now we have a complete description of the control unit: its inputs, outputs, states and transitions.

We have also a complete description of states and interface (i.e. I/O) of datapath, but we need to describe its operations.

6.3.5 The Datapath

To describe how datapath performs operations that the control unit requires, we begin defining how outputs are generated.

The line $wmem$ is true iff the second field in the control signal input has the $wmem$ value.

The line $dreq$ is true iff $wmem$ or $rmem$ are present in the control signal input.

The $addr$ variable is equal to $address\ mar$.

The $iack$ line is true if $wrtn$ is present in the control signal, is false if $rrtn$ is present, and maintains its current value otherwise.

The line $zero$ is true iff acc is zero.

The variable opc is equal to $opcode\ ir$.

To complete the description of the datapath, we need to introduce some syntactic conventions: let $\langle r, w, x \rangle$ be the control signal. We define the following:

$$read\_reg = \begin{cases} \perp & \text{if } r = none \\ \alpha & \text{if } r = r\alpha \end{cases}$$

$$write\_reg = \begin{cases} \perp & \text{if } w = none \\ \beta & \text{if } w = w\beta \end{cases}$$

For example, when the control unit is in state 0, the control signal is $\langle rpc, wmar, none \rangle$; here $read\_reg = pc$ and $write\_reg = mar$.

To describe how internal registers are manipulated and what values are produced on the $dataout$ output bus, we need to define how control signals are interpreted. Let $\langle r, w, x \rangle$ be a control signal:

- if $read\_reg \neq \perp$ and $write\_reg \neq \perp$ then, in the new datapath state, $write\_reg := read\_reg$.
- if $read\_reg = mem$ then, in the new state, $write\_reg := datain$.
- if $write\_reg = mem$ then, in the new state, $dataout := read\_reg$.
- if $x = zero$ then, in the next state, $buf := wordn\ 0$.
- if $x = add$ or $x = sub$ then, in the next state, $buf := (x)(arg, read\_reg)$.
- if $x = inc$ then, in the next state, $buf := inc\ read\_reg$.
- what is not specified to be changed by the preceding rules, is left unchanged.

This completes our description of datapath, and of the microprogramming level model.

We could introduce formal descriptions, but they are pretty long, difficult to understand, and very tied to the TAMARACK-3 example. If you are interested in, refer to the TAMARACK-3 documentation.

6.4 The Phase Level Interpreter

This description level is very similar to the microprogramming level. But there is an important difference: while the microprogramming description is abstract,

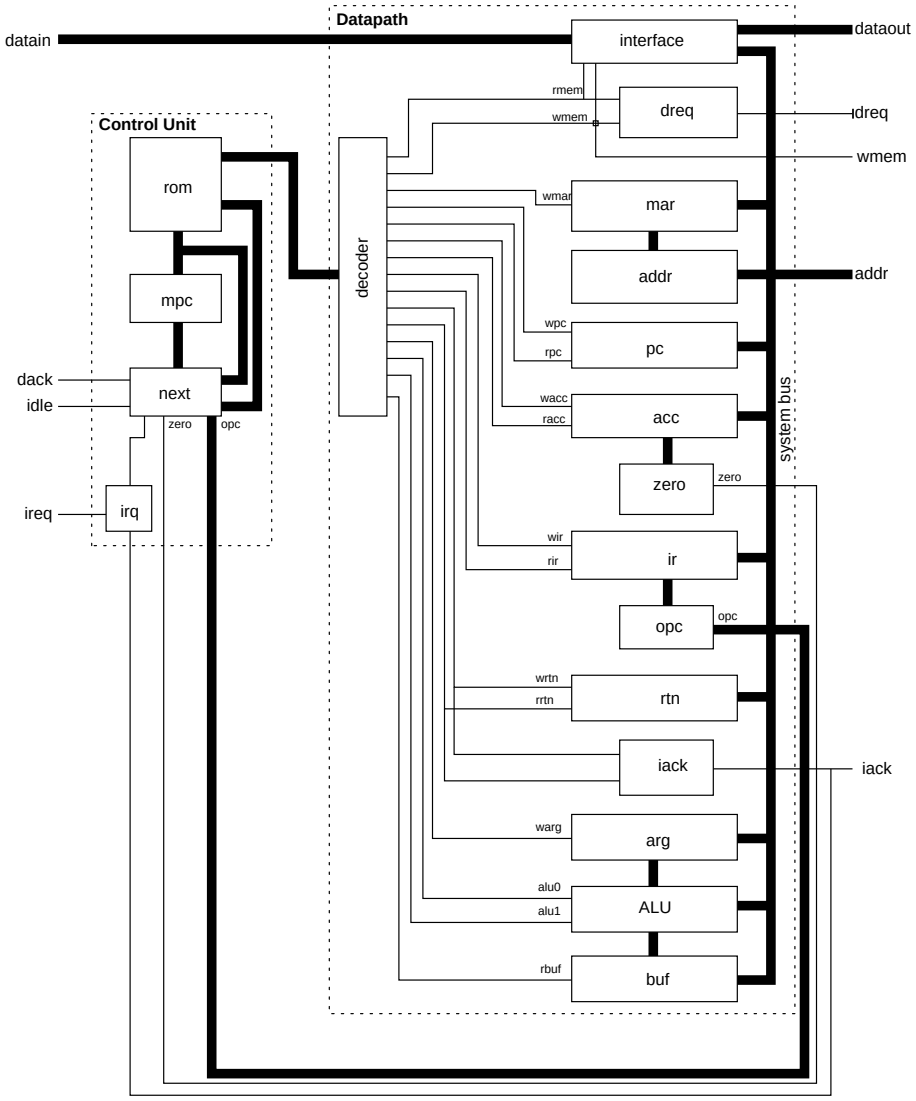


Figure 6.5: Register-Transfer Level Architecture

here the description is concrete. In the preceding section, we presented a functional model of internals of our example processor. Here we will map this description to actual signals, to lines and hardware components.

The plan of this section is:

- translating abstract microinstructions into a set of hardware signals.
- describing how the datapath is implemented.
- describing how the control unit is implemented.
- putting all together to present an implementation of TAMARACK-3 microprocessor.

To do this work we need an overview of the internal architecture of the TAMARACK-3 at the register-transfer level. You can see it in figure 6.5.

We could get a more concrete description level, which corresponds tightly to real hardware by exploding the content of each box in the figure, but this is a lengthy process, and, if you read with attention so far, you are able to do it.

6.4.1 Compiling Microcode

In the microprogramming level model, microinstructions have been represented as four fields records. Each field had its own internal structure and its own datatypes.

Here we need to represent microinstructions as tuples of bits. So we write a compiler from abstract to concrete microinstructions.

From figure 6.5, we see that fifteen signals are generated by the decoder to components of the datapath.

A microinstruction's first field could be compiled into a tuple of 15 bits, where every bit represents a value on those control lines.

Here is the procedure:

```
first_field_compiler (tok1, tok2, tok3) =
      ((tok2 = ‘wmem‘),
       (tok1 = ‘rmem‘),
       (tok2 = ‘wmar‘),
       (tok2 =  ‘wpc‘),
       (tok1 =  ‘rpc‘),
       (tok2 = ‘wacc‘),
       (tok1 = ‘racc‘),
       (tok2 =  ‘wir‘),
       (tok1 =  ‘rir‘),
       (tok2 = ‘wrtn‘),
       (tok1 = ‘rrtn‘),
       (tok2 = ‘warg‘),
       ((tok3 = ‘inc‘) \/ (tok3 = ‘add‘)),
       ((tok3 = ‘inc‘) \/ (tok3 = ‘sub‘)),
       (tok1 = ‘rbuf‘));;
```

In the preceding definition, we coded the ALU operation in the following way:

op	alu0	alu1
zero	0	0
sub	0	1
add	1	0
inc	1	1

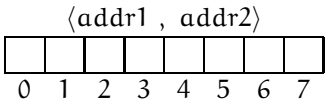
To compile the second field we can use the following table:

value	bit0	bit1
continue	0	0
waitidle	1	0
waitdack	1	1

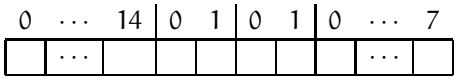
We need other two bits to compile the third field, because it can take four values:

value	bit0	bit1
jopcode	0	0
jzero	0	1
jireq	1	0
jump	1	1

The last field is a pair of state numbers; we have sixteen states, so we need four bits per address:



Assembling all these representations, we obtain a compiled microinstruction:



The decoder in figure 6.5 is a null component: we represent microinstructions in the ROM of the control unit with their compiled version, that is, as 27-bits words. Then we split lines into two sets: 15 (the first field) go to the decoder, while the other 12 go to the next component. So the work the decoder has to do is simply to give the right wire to the right datapath's component.

Now is clear how microinstructions are represented, and it is also clear the interface between control unit and datapath.

6.4.2 The System Bus

The system bus of TAMARACK-3 is used to transfer data between various devices in the datapath.

When we want to model this bus, we need to consider a problem: two (or more) devices must not write onto the bus at the same time. But there is no centralized control, so we need to ensure this property on the whole design of the microprocessor.

The goal of this subsection is simple: we want to define in a formal way that two devices cannot write data onto the bus at the same time. When we will (try to) prove that the microprocessor is formally correct, then we will prove that this property holds in the whole design.

When a datapah device writes onto the system bus, it performs a *read* operation, and this fact is controlled by the *read* line of that device. For example,

when $rmem$ is true, the $interface$ is trying to write a value onto the system bus.

What we need is a predicate that is true iff at most one read line is true. The following definition do the work:

```
Bus (rmem, rpc, racc, rir, rrtn, rbuf, bus) =
 (bus t =
  (rmem t => ~( rpc t \/ racc t \/  rir t \/ rrtn t \/ rbuf t)
  | rpc t => ~(racc t \/  rir t \/ rrtn t \/ rbuf t)
  |racc t => ~( rir t \/ rrtn t \/ rbuf t)
  | rir t => ~(rrtn t \/ rbuf t)
  |rrtn t => ~rbuf t
  |          true))
```

6.4.3 ALU

The ALU may perform four operations: inc, add, sub and $zero$. They are selected using control signals $alu0$ and $alu1$, which encode the operation request as specified in section 6.4.1.

At the next description level we could find the real hardware implementation of ALU. Now we specify only what it must do when it receives a control signal.

Because ALU is a component, we describe it by means of a predicate:

```
ALU (alu0, alu1, inp1, inp2, out) = !t:time.
  out t =
    ( ((alu0, alu1) = ( true, true )) => inc (inp2 t)
    | ((alu0, alu1) = ( true, false)) => add (inp1 t, inp2 t)
    | ((alu0, alu1) = (false, true )) => sub (inp1 t, inp2 t)
    |                                    wordn 0)
```

6.4.4 The Interface

The interface device is responsible of the communication between microprocessor and memory. It has four inputs and one output: the inputs are $rmem$, $wmem$, bus, $datain$ and the output is $dataout$.

The behaviour of the interface component is the following:

- if $rmem$ is true then bus is set to $datain$.
- if $wmem$ is true then $dataout$ is set to bus.
- otherwise $dataout$ is set to 0.

We can formalize the interface as:

```
Interface (wmem, rmem, bus, datain, dataout) = !t:time.
    (rmem t ==> (bus t = datain t)) /\
    (dataout t = (wmem t => bus t | wordn 0))
```

To complete memory interface's formalization, we need to specify the dreq device: it is true iff an I/O to memory is requested. So its behaviour is described simply as (remembering definitions of gates):

```
! t:time . OR2 (rmem t, wmem t, dreq t)
```

6.4.5 Other Signals and Devices

To complete description of datapath devices, we need to describe how registers are implemented and how the remaining outputs are produced.

All registers in the datapath are instances of a common definition: mar, pc, acc, ir, rtn, arg and buf share a common behaviour.

They have as I/O lines (possibly not connected):

- read and write
- an input (it can be the bus)
- an output (that is the variable containing the state)
- an auxiliary output to the bus

The behaviour of such a register is described by the following predicate:

```
Register (write, read, input, bus, output) = ! t:time .
    (read t ==> (bus t = output t)) /\
    (output (t + 1) = (write t => input t | output t))
```

If a read operation is requested then the state is put onto the bus; if a write operation is requested, then the new state will be the current input; if no operation is requested then the register maintains its state.

There are other small devices to describe in order to complete the datapath: addr, zero, opc and iack.

The addr device converts the context of mar register from : *wordn to : *address type. So its behaviour is easily described by the predicate:

```
Addr (in, out) = ! t:time .
    out t = address (in t)
```

The opc device is a converter too: it takes a : *wordn from the ir register and produces a : *wordop.

```
Opc (in, out) = ! t:time .
    out t = opcode (in t)
```

The zero device produces true iff its input is 0:

```
Zero (in, out) = ! t:time .
    out t = (in t = wordn 0)
```

The iack line is produced by looking values of wrtn and rrtn. When wrtn is true, iack is set to true; when rrtn is true, iack is set to false; when both are false, iack holds its current value.

Remembering the flip-flops specifications, the iack device is described by:

```
JK0_FF (wrtn, rrtn, iack)
```

6.4.6 Datapath Implementation

Now we have all components described in the standard formal way. Applying composition, we obtain a predicate which describes the entire datapath:

```
Datapath (rmem, wmem, wmar,  wpc,  rpc,
          wacc, racc,  wir,  rir, wrtn,
          rrtn, warg, alu0, alu1, rbuf),
         datain, dataout, dreq, outwmem,
         addr, opc, zero, iack) =
  ? bus, mar, pc, acc, ir, rtn, arg, alu, buf .
    Bus(rmem, rpc, racc, rir, rrtn, rbuf, bus)  /\
    Interface(wmem, rmem, bus, datain, dataout) /\
    (! t:time . OR2(rmem t, wmem t, dreq t))    /\
    (outwmem = wmem)                            /\
    Register(wmar, false, bus, bus, mar)        /\
    Addr(mar, addr)                             /\
    Register(wpc, rpc, bus, bus, pc)            /\
    Register(wacc, racc, bus, bus, acc)         /\
    Zero(acc, zero)                             /\
    Register(wir, rir, bus, bus, ir)            /\
    Opc(ir, opc)                                /\
    Register(wrtn, rrtn, bus, bus, rtn)         /\
```

```
JK0_FF(wrtn, rrtn, iack)                    /\
Register(warg, false, bus, bus, arg)        /\
ALU(alu0, alu1, arg, bus, alu)              /\
Register(true, rbuf, alu, bus, buf)
```

6.4.7 Control Unit Implementation

The control unit, as you can see from figure 6.5, is composed by three main subdevices: rom, mpc and $next$.

Let's begin with the rom device. It is like a table, where each line contains a microinstruction, coded as described in subsection 6.4.1, and the index is the corresponding state of finite state machine which the control unit implements.

We assume that we have an uninterpreted type, : $*fsmstate$, and two functions to deal with it:

$$\begin{array}{lcl} fsmstatetonum & : & *fsmstate \longrightarrow num \\ numtofsmstate & : & num \longrightarrow *fsmstate \end{array}$$

with the axiom

$$\forall n.n \leq 15 \Rightarrow fsmstatetonum(numtofsmstate\ n) = n$$

Now, remembering correspondence between states and microinstructions, we can define the predicate that describes the rom device:

```
Rom (in,
     wmem, rmem, wmar,  wpc,  rpc,
     wacc, racc,  wir,  rir, wrtn,
     rrtn, warg, alu0, alu1, rbuf,
     wait0, wait1, jump0, jump1,
     addr10, addr11, addr12, addr13,
     addr20, addr21, addr22, addr23) = ! t: time .
  let out = (wmem t, rmem t, wmar t,  wpc t,  rpc t,
             wacc t, racc t,  wir t,  rir t, wrtn t,
             rrtn t, warg t, alu0 t, alu1 t, rbuf t,
             wait0 t, wait1 t, jump0 t, jump1 t,
             addr10 t, addr11 t, addr12 t, addr13 t,
             addr20 t, addr21 t, addr22 t, addr23 t,) in
  let n = fsmstatetonum (in t) in out =
    ( n =  0 => microcompile(( `rpc`, `wmar`, `none`),
                             `waitidle`, `jireq`, ( 1, 2))
    | n =  1 => microcompile(( `rpc`, `wrtn`, `zero`),
                             `continue`,  `jump`, (12, 0))
    | n =  2 => microcompile((`rmem`,  `wir`, `none`),
                             `waitdack`,  `jump`, ( 3, 0))
```

```
    | n =  3 => microcompile(( 'rir', 'wmar', 'none'),
                             'waitidle','jopcode',( 4, 0))
    | n =  4 => microcompile(('none', 'none', 'none'),
                             'continue', 'jzero', ( 5,11))
    | n =  5 => microcompile(( 'rir',  'wpc', 'none'),
                             'continue',  'jump', ( 0, 0))
    | n =  6 => microcompile(('racc', 'warg', 'none'),
                             'waitidle',  'jump', (13, 0))
    | n =  7 => microcompile(('racc', 'warg', 'none'),
                             'waitidle',  'jump', (14, 0))
    | n =  8 => microcompile(('rmem', 'wacc', 'none'),
                             'waitdack',  'jump', (11, 0))
    | n =  9 => microcompile(('racc', 'wmem', 'none'),
                             'waitdack',  'jump', (11, 0))
    | n = 10 => microcompile(('rrtn',  'wpc', 'none'),
                             'continue',  'jump', ( 0, 0))
    | n = 11 => microcompile(( 'rpc', 'none',  'inc'),
                             'continue',  'jump', (12, 0))
    | n = 12 => microcompile(('rbuf',  'wpc', 'none'),
                             'continue',  'jump', ( 0, 0))
    | n = 13 => microcompile(('rmem', 'none',  'add'),
                             'waitdack',  'jump', (15, 0))
    | n = 14 => microcompile(('rmem', 'none',  'sub'),
                             'waitdack',  'jump', (15, 0))
    |           microcompile(('rbuf', 'wacc', 'none'),
                             'continue',  'jump', (11, 0)))
```

In the definition, *microcompile* is the function that compiles an abstract microinstruction into a 27-bit tuple.

The *mpc* subdevice is a register which holds the current finite state machine's state number. It is defined by:

```
Mpc (in, out) = ! t:time .
  (out (t + 1) = in t) /\
  (out 0 = numtofsmstate  0)
```

The *next* subdevice calculates the next state for the finite state machine. If there is a wait condition, it outputs the current state until the condition is satisfied; then, following the jump instruction, it produces the right next state.

Formally:

```
Next (dack, idle, ireq,
      zero, opc, wait0, wait1, jump0, jump1,
      addr10, addr11, addr12, addr13,
```

```
     addr20, addr21, addr22, addr23,
     mpc, next) = ! t:time .
  let waitcond =
    (((wait0 t, wait1 t) = (true, true) /\ ~dack t) \/
     ((wait0 t, wait1 t) = (true,false) /\
       ~(idle t \/ ~dack t))) in
    next t =
      ( waitcond => mpc t
      | ((jump0 t, jump1 t) = ( true,  true)) =>
          (addr10 t, addr11 t, addr12 t, addr13 t)
      | ((jump0 t, jump1 t) = ( true, false)) =>
          (ireq t =>
            (addr10 t, addr11 t, addr12 t, addr13 t)
          | (addr20 t, addr21 t, addr22 t, addr23 t))
      | ((jump0 t, jump1 t) = ( true,  true)) =>
          (zero t =>
            (addr10 t, addr11 t, addr12 t, addr13 t)
          | (addr20 t, addr21 t, addr22 t, addr23 t))
      | numtofsmstate (valop(opc) +
          fsmstatetonum (addr10 t, addr11 t,
                         addr12 t, addr13 t)))
```

To complete the control unit subdevices, we need to define irq, Its output must be equal to ireq if iack is false, but false otherwise. Formally:

```
Irq (ireq, iack, out) = ! t:time .
  out t = ~iack t /\ ireq t
```

Now all parts of the control unit are defined. The description of the control unit itself is obtained combining all subdevices:

```
ControlUnit (dack, idle, ireq, iack,
             zero, opc,
             rmem, wmem, wmar,  wpc,  rpc,
             wacc, racc,  wir,  rir, wrtn,
             rrtn, warg, alu0, alu1, rbuf) =
  ? irq, wait0, wait1, jump0, jump1,
    addr10, addr11, addr12, addr13,
    addr20, addr21, addr22, addr23,
    mpc, next .
    Next (dack, idle, irq, zero, opc,
```

```
        wait0, wait1, jump0, jump1,
        addr10, addr11, addr12, addr13,
        addr20, addr21, addr22, addr23,
        mpc, next) /\
   Irq  (ireq, iack, irq) /\
   Mpc  (next, mpc) /\
   Rom  (mpc,
        wmem, rmem, wmar,  wpc,  rpc,
        wacc, racc,  wir,  rir, wrtn,
        rrtn, warg, alu0, alu1, rbuf,
        wait0, wait1, jump0, jump1,
        addr10, addr11, addr12, addr13,
        addr20, addr21, addr22, addr23)
```

6.4.8 Microprocessor Implementation

Now is time to define the whole microprocessor, using descriptions of the control unit and of the datapath.

The microprocessor is defined by the following predicate:

```
Tamarack (datain, dataout,
         dack, idle, ireq,
         dreq, outwmem, iack,
         addr) =
  ? wmem, rmem, wmar,  wpc,  rpc,
    wacc, racc,  wir,  rir, wrtn,
    rrtn, warg, alu0, alu1, rbuf,
    opc, zero .
    Datapath (rmem, wmem, wmar,  wpc,  rpc,
              wacc, racc,  wir,  rir, wrtn,
              rrtn, warg, alu0, alu1, rbuf,
              datain, dataout, dreq, outwmem,
              addr, opc, zero, iack) /\
    ControlUnit (dack, idle, ireq, iack,
                 zero, opc,
                 rmem, wmem, wmar,  wpc,  rpc,
                 wacc, racc,  wir,  rir, wrtn,
                 rrtn, warg, alu0, alu1, rbuf)
```

This completes the description of the phase level model. We stop here in the descent through architectural levels. But the first part of this book covers how to manage from this level below, so our incompleteness is not very important.

6.5 Planning Verification

In this section we will try to plan verification of a microprocessor. We will refer to descriptions of TAMARACK-3 we gave in the preceding sections. But we will try to give you a general view of the approach and of the problems we have to solve.

We begin with Programming Level Model and we end with real hardware. We will develop the goal to be proved at each level and why these goals have those structure.

6.5.1 Programming Level Adequacy

When we look a project of a microprocessor, specifications are informal and they covers, in most cases, three topics:

- how the programming level has to be
- the general architecture of microprogramming level
- details on hardware constraints

It is left to designers to provide a programming level which satisfies the goal the microprocessor is developed for. Normally it is easy to see that this goal is achieved.

But we want to point out that the first step in the verification of a microprocessor should be to prove that its design is adequate for the purpose the microprocessor has to achieve.

For example, if we develop a general purpose microprocessor, one that, probably, will be the heart of a computer system, then we need that its assembler, and the way it uses interrupts, is able to code an operative system.

There is a theoretical way to do this: we can prove in a formal way that assembler, without memory size limitations, is able to compute every partial recursive function. This fact implies that the assembler is able to represent, in principle, every computable function.

Proving this goal is simple: if we can write programs to compute the $zero$ function, the $successor$ function and the $projection$ function, and we are able to develop program frames which implement the $composition$ operator, the $primitiverecursion$ operator and the $minimalization$ operator, then we have proved that the programming level model is correct. Obviously we have to prove in a formal way that the code we produced performs the intended operations.

There is also a practical concern: the microprocessor is adequate if it is able to run a set of test programs. These test programs must be chosen according to the goal the microprocessor was developed for, i.e., if it has to be able to run an operative system, then it must be tested using the most significative routines of OS, for example the interrupt handler, the process scheduler, the system call dispatcher, the simplest I/O routines.

We have to choose carefully our test programs, and, if we are able to write them correctly in microprocessor's assembler, then microprocessor's programming level model is correct.

When we have done these tests on the correctness of the programming level model, we are ready to use the model itself as a specification of the intended microprocessor's behaviour.

6.5.2 Verification of Microprogramming Level Model

The difference between programming level model and microprogramming level model can be summarized in two points:

- from outside to inside the microprocessor.
- from an imperative paradigm to a functional one.

The first point expresses the fact that one level is the implementation of the other one.

The second point move attention from states to communications. The functional paradigm represents the fact that what implements an imperative language, such as assembler, in an hardware system, is a set of communicating devices, and the structure of functional description mirrors the communication structure.

To verify microprogramming level against programming level, we have to show that the interpreter of machine language, which is embedded in the microprogramming level, is correct. We have to show also that interrupts are managed in the way described by programming level model, and that memory (and, in general, I/O) interface satisfies the requirements imposed by the upper model.

Technically, we have to show that exists a function τ from programming model's states to microprogramming's such that the following diagram commutes:

$$\begin{array}{ccc} States & \xrightarrow{\tau} & \mu States \\ \pi\downarrow & & \downarrow\pi' \\ States & \xrightarrow{\tau} & \mu States \end{array}$$

where π is a function which represents a sequence of programming level events (assembler instructions and hardware interrupts) and π' is the function that microprogramming model computes when it evaluates the events of π.

In other words, we have to prove that exists a function τ such that, for every function π, $\pi \circ \tau = \tau \circ \pi'$, where π' is the function described by the microprogramming level formalization.

6.5.3 Verification of Phase Level Model

The main difference between microprogramming level and phase level is that one is abstract and the other one is concrete. This means that entities in the phase level are represented as bits, and as operations on them, whereas

in the microprogramming level they are instances of abstract (data)types and operations on them.

But there is another difference between the two models, and it is more subtle but not less important. The accent in the microprogramming level description is posed on the control unit, while, in the phase level, the central point in the description is datapath with its components.

This difference is due to the fact that the upper model's first problem is to convert machine language instructions into sequences of microinstructions, while the lower model's problem is to describe how subcomponents are related to each other in order to concurrently execute microinstructions.

The verification process is quite simple: we have a translation τ from microprogramming level's microinstructions to phase level's ones. We have to prove that microprocessor's description of the lower level implies the upper's modulo τ.

To do so we have to prove correctness of the control unit implementation and of the datapath. The first task is simple because we have already presented the technique to formally verify finite state machines. The second task is a normal correctness proof, because we have a set of fully specified components, assembled to constitute a complex device, the datapath, and we have a formal specification of its intended behaviour.

So these proofs could be complex, but the general way to perform them is clear.

6.5.4 Real Hardware Verification

To verify real hardware, we use the standard techniques, we have presented in the previous chapters.

The problem we have to solve is what to verify. Using the homomorphic property, and considering that the overall design of the microprocessor was already verified at the phase level, we could limit ourself to the formal verification of the internal components of the datapath and the control unit.

From these proofs, we know that elementary components of the microprocessor are correct. From the phase level model verification, we deduce that the datapath and the control unit are correct with respect to the microprogramming level's intended behaviour.

The microprogramming level verification tells us that the microprocessor structure is correct, and, because of the lower level's proofs, we can safely say that the microprocessor is correct, and this means that the microprocessor implements the programming behaviour correctly.

The programming level verification tells us that the microprocessor design is adequate to the purpose it is intended to satisfy.

6.6 Final Considerations

To close this chapter, we want to introduce some overall considerations.

First of all, we have to note that the specification of programming level is quite plain; we have presented in an informal way how to describe a generic processor.

The second consideration is that from microprogramming level to hardware level, specifications are closely microprocessor dependent. So we provided an example, TAMARACK-3, which is simple enough to be presented in a plain way, but it is not too elementary. Microprocessor's architecture is quite standardized so the distinction between control unit and datapath is general, but modern microprocessors have other complex components that we have not presented here: an example is given by pipelines.

As we said in the introduction of this chapter, formal verification of microprocessors is still a research topic, and, because of this, we are not able to give established techniques to approach problems we tried to present.

Our approach is supported by a couple of examples, and, perhaps, it could be extended to more complex microprocessors. We believe so, but this is an opinion, not a mathematical statement.

Chapter 7

Other Techniques

7.1 Introduction

Now we have presented a method to specify and verify circuits. What we have tried to explain is the most known, studied and used way to represent circuits and verify them. In this chapter we want to introduce you some different approaches.

Because formal verification is still a research field, we will not provide an overview, but we give some ideas of other basical approaches, and we choose them instead of others because they can be integrated in the main method.

In particular we will see that circuits can also be formalized as functions, not only as relations (predicates). Sometimes this is simpler, or more natural. The second section is devoted to present this topic.

We have seen that we can verify circuits by comparing them to other circuits. Third section will show a powerful technique to synthesize correct circuits, when their specifications are given as programs or algorithms. And in the final section we will discuss how to use other logics, for example temporal logic, to analyze the behaviour of complex circuits.

7.2 Formalizing Circuits as Functions

Till now, we have formalized circuits as predicates. This approach leads to simple representations, which enjoy some nice properties, as homomorphicity and compositionality. But this approach is, in some way, unnatural.

Common sense tells us that circuits compute functions. The goal of this section is to present the way to formalize circuits by means of mathematical functions.

From an abstract point of view, every digital circuit could be represented using only the following predicates:

$$\begin{array}{lcl} \text{NAND } a\ b\ out & = & \forall\, t.out\ t = \neg(a\ t \wedge b\ t) \\ \text{D0\_FF } a\ out & = & \forall\, t.(out\ 0 = false) \wedge (out\ (t+1) = a\ t) \end{array}$$

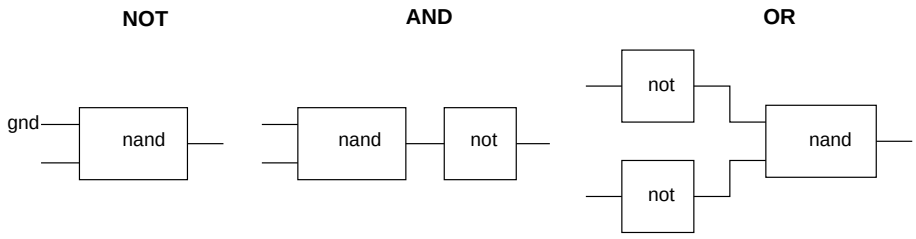


Figure 7.1: Gates

We allow as valid combinations: wiring and black boxing whose formal counterparts are predicate conjunctions and definitions.

We can do the same using a pure functional approach:

$$\begin{array}{lcl} nand\ a\ b & = & \lambda\, t.\neg(a\ t \wedge b\ t) \\ d0\_ff\ a & = & \lambda\, t.(t = 0) \rightarrow false \mid a\ (t-1) \end{array}$$

It is easy to prove that the following statement holds:

$$(\forall\ a\ b.NAND\ a\ b\ (nand\ a\ b)) \wedge (\forall\ c.D0\_FF\ c\ (d0\_ff\ c))$$

This fact proves equivalence of the representations of basic components.

To represent more complex circuits, we need to compose elementary devices through wires. We will use function application to represent wires. Sometimes it is convenient to shorten expression, and we could use function composition, but this is just a shortcut because:

$$(f \circ g)\ x = f\ (g\ x)$$

With these instruments it is easy to describe elementary gates and combinational circuits. For example:

$$\begin{array}{lcl} gnd & = & \lambda\, t.false \\ pwr & = & \lambda\, t.true \\ not & = & nand\ true \\ and & = & not \circ nand \\ or & = & \lambda\, x\ y.nand\ (not\ x)\ (not\ y) \end{array}$$

To represent sequential circuits, we have to note that they could be regarded as a combinational circuit plus some feedback wires. Our problem is how to represent these particular wires.

Let's reason on an example: figure 7.2 represents an implementation of an SR flip-flop.

The feedback wire is the one between $d0\_ff$ and not components. We could imagine to cut this wire obtaining a new circuit, which can be represented using application. It has one more input line: let's call it x.

We can formalize the extended circuit as follows

$$\begin{array}{lcl} G & = & \lambda\, a\ b\ c\ x.(d0\_ff \circ or2)\ (nor3\ (not\ (x\ s\ r))\ a\ b)\ c \\ H & = & \lambda\, x\ s\ r.G\ s\ r\ (and2\ s\ (not\ r))\ (x s r) \end{array}$$

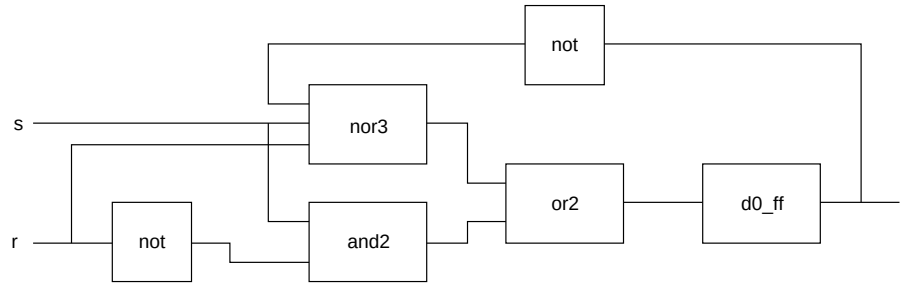


Figure 7.2: Implementation of the SR flip-flop

The extension wire, x is formalized as a function of s and r. We need to do in this way in order to state that x is not really free, but its value depends on the ones on s and r lines.

We want to formalize the flip-flop circuit. The relation between this circuit and H is simple: H represents the sequence

$$\{H\ \bot, H\ (H\ \bot), H\ (H\ (H\ \bot)), \ldots\}$$

where $\bot$ represents the indeterminate value.

Our reasoning is: we don't know the initial value of x, so it is $\bot$, but, because x is a feedback wire, it is computed again and again, leading to the above sequence.

What we want to model is the limit of this sequence. It can be proved that it exists and it is unique. We call it the *minimum fixed point* and it is represented using the μ operator. It satisfies the fundamental property:

$$\mu\ F = F(\mu\ F) \tag{7.1}$$

So our circuit can be formalized as

$$sr0 = \mu\ H$$

We can characterize better sr0 by calculating it using property 7.1.

$$\begin{array}{ll} & sr0 \\ = & \\ & \mu\ H \\ = & \\ & H(\mu\ H) \\ = & \\ & \lambda\ s\ r.G\ s\ r\ (and2\ s\ (not\ r))\ ((\mu\ H)\ s\ r) \\ = & \\ & \lambda\ s\ r.(d0\_ff \circ or2)\ (nor3\ (not\ ((\mu\ H)\ s\ r))\ s\ r)\ (and2\ s\ (not\ r)) \\ = & \\ & \lambda\ s\ r\ t.t = 0 \quad \rightarrow false \mid \\ & \qquad\qquad \neg(\neg(\mu\ H)\ s\ r\ (t-1) \vee s\ (t-1) \vee r\ (t-1)) \vee \\ & \qquad\qquad (s\ (t-1) \wedge \neg r\ (t-1)) \end{array}$$

So

$$(sr0\ s\ r)\ 0 \quad = \quad false$$

and

$$\begin{array}{l} (sr0\ s\ r)\ (t+1) \\ = \\ \neg(\neg(\mu\ H)\ s\ r\ t \vee s\ t \vee r\ t) \vee (s\ t \wedge \neg r\ t) \\ = \\ (s\ t \wedge \neg r\ t) \vee (\neg s\ t \wedge \neg r\ t \wedge (\mu\ H)\ s\ r\ t) \end{array}$$

With this information we can afford the correctness proof of $sr0$.

In a functional approach specifications are given as predicates depending on a function. Their meaning is: a specification's predicate is true when its parameter satisfies the correctness property that the predicate represents. In our example a reasonable specification is:

$$\begin{array}{lll} S\ spec = & (\forall\ s\ r\ t: \neg(s\ t \wedge r\ t): & \\ & (spec\ s\ r\ 0 = false) \wedge & \\ & (\neg s\ t \wedge \neg r\ t \Rightarrow (spec\ s\ r)\ (t+1) = (spec\ s\ r)\ t) \wedge & \\ & (s\ t \wedge \neg r\ t \Rightarrow (spec\ s\ r)\ (t+1) = true) \wedge & \\ & (\neg s\ t \wedge r\ t \Rightarrow (spec\ s\ r)\ (t+1) = false) & \end{array}$$

We have to prove that $S\ sr0 = true$. Let's calculate:

$$\begin{array}{lll} & S\ sr0 & \\ = & & \\ & (\forall\ s\ r\ t: & \neg(s\ t \wedge r\ t): \\ & & (sr0\ s\ r\ 0 = false) \wedge \\ & & (\neg s\ t \wedge \neg r\ t \Rightarrow (sr0\ s\ r)\ (t+1) = (sr0\ s\ r)\ t) \wedge \\ & & (s\ t \wedge \neg r\ t \Rightarrow (sr0\ s\ r)\ (t+1) = true) \wedge \\ & & (\neg s\ t \wedge r\ t \Rightarrow (sr0\ s\ r)\ (t+1) = false) \\ = & & \\ & (\forall\ s\ r\ t: & \neg(s\ t \wedge r\ t): true \wedge \\ & & (\neg s\ t \wedge \neg r\ t \Rightarrow ((s\ t \wedge \neg r\ t) \vee (\neg s\ t \wedge \neg r\ t \wedge sr0\ s\ r\ t) = \\ & & \qquad (sr0\ s\ r)\ t)) \wedge \\ & & (s\ t \wedge \neg r\ t \Rightarrow ((s\ t \wedge \neg r\ t) \vee (\neg s\ t \wedge \neg r\ t \wedge sr0\ s\ r\ t))) \wedge \\ & & (\neg s\ t \wedge r\ t \Rightarrow \neg((s\ t \wedge \neg r\ t) \vee (\neg s\ t \wedge \neg r\ t \wedge sr0\ s\ r\ t)))) \\ = & & \\ & (\forall\ s\ r\ t: \neg(s\ t \wedge r\ t): true) & \\ = & & \\ & true & \end{array}$$

This end our correctness proof.

To compare the functional approach with the standard one, we must say two important facts:

- the functional approach is not so used as the predicate based representation.

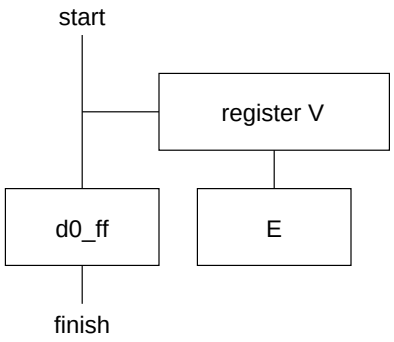


Figure 7.3: Implementation of assignement V := E

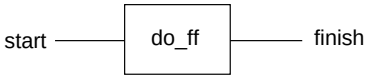


Figure 7.4: Implementation of SKIP

- there is a fundamental equivalence between the two methods, that is, every predicate approach could be converted into a functional one, and viceversa.

The main difference lays in the instruments we can use to prove and in the way we can specify properties.

In some way the functional approach is *more mathematical* and sometimes people find its representations to be more *natural*.

Surely the way to specify what a circuit is intended to do is more direct because we can declare what the circuital representation must satisfy instead of saying a property is equivalent to (or that must be a consequence of) the circuital representation.

But logic has a long tradition and a rich set of powerful instruments to prove things, while fixed point algebras are complex and their proof are often involved.

The functional approach is very interesting principally because it uses the same instruments that are widely used to represent semantics of computer programs. So we can induce that there is a strong link between circuits and programs. And this is the main topic of the next section.

7.3 Synthesizing Circuits

A possible form of the correctness argument is a circuit description. We used this technique to prove correctness of finite state machines. We have a circuit and we want to prove it is correct; if we have another circuit which performs the same function and if we know that it is correct, then we prove correctness of the former one comparing it with the latter.

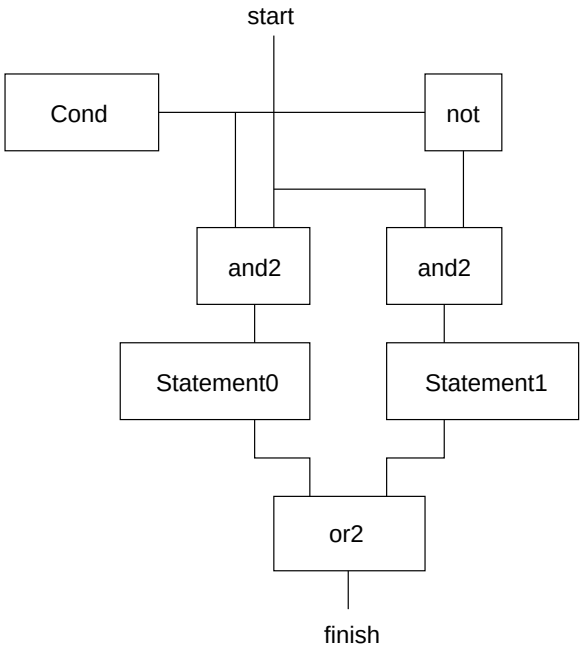


Figure 7.5: Implementation of IF Cond THEN Statement0 ELSE Statement1

Now we want to introduce a method to compile programs into circuits and we will prove that this compilation preserves correctness. So, if our specification is a program, or an algorithm, we can prove correctness by comparing the target circuit with the compiled specification, which is another circuit.

We will use a subset of Occam, called Handel, as the language whose programs we will compile. Admittable values are booleans, integers (with a fixed size) and arrays. The main construct is the command and its syntax is:

$$
\begin{array}{rcl}
\langle command \rangle & ::= & \langle variable \rangle := \langle expression \rangle \\
& | & \text{SKIP} \\
& | & \text{IF } \langle expression \rangle \text{ THEN } \langle command \rangle \text{ ELSE } \langle command \rangle \\
& | & \text{WHILE } \langle expression \rangle \ \langle command \rangle \\
& | & \text{SEQ } \langle command \rangle^{+} \\
& | & \text{PAR } \langle command \rangle^{+}
\end{array}
$$

Our compiler will produce synchronous circuits, and every command will be compiled in a control circuit where the following conditions hold:

- every enabled register change its state at every clock pulse.
- every control circuit has an input line (labelled start) and an output line (finish).
- every signal on the start line is valid for exactly one clock cycle.
- no start signal is given until a finish signal is generated.

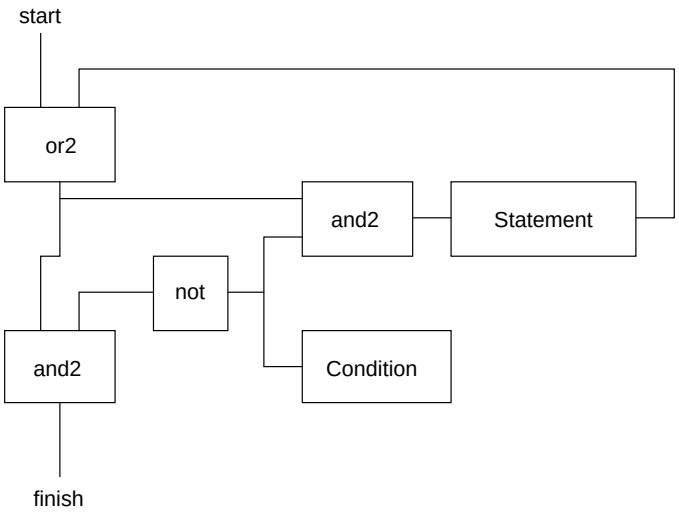


Figure 7.6: Implementation of WHILE Condition Statement

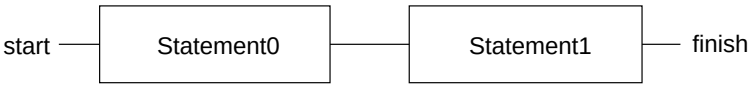


Figure 7.7: Implementation of SEQ Statement0 Statement1

- every expression could be computed in exactly one clock cycle.

in figures 7.3 to 7.8 you can see the implementation of commands.

Let's prove that these circuits are correct.

We begin with assignement; implementation is given by the following definition

$$
\begin{array}{l}
\forall\ start\ finish\ V\ E.\\
\quad assignement\ start\ finish\ V\ E =\\
\quad\quad \exists\ in.\\
\quad\quad\quad\quad d0\_ff\ start\ finish\\
\quad\quad\quad \wedge\ reg\ in\ start\ V\\
\quad\quad\quad \wedge\ \forall\ t.in\ t = E\ t
\end{array}
$$

the corresponding specification is

$$
\begin{array}{l}
\forall\ start\ finish\ V\ E.\\
\quad assignement\_spec\ start\ finish\ V\ E = \forall\ t.\\
\quad\quad start\ t \rightarrow (finish\ (t+1) \wedge V\ (t+1) = E\ t) \mid \neg finish\ (t+1)
\end{array}
$$

we want to prove that

$$
\begin{array}{l}
\forall\ start\ finish\ V\ E.\\
\quad assignement\ start\ finish\ V\ E \Rightarrow\\
\quad assignement\_spec\ start\ finish\ V\ E
\end{array}
$$

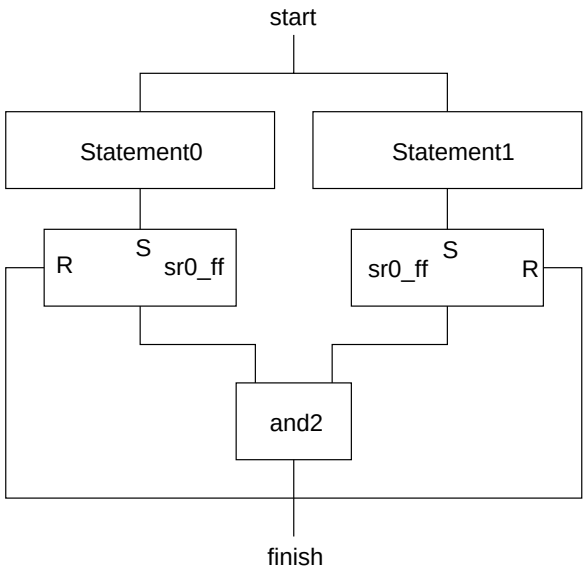


Figure 7.8: Implementation of PAR Statement0 Statement1

Here is the proof:

$$
\begin{aligned}
& \mathit{assignement\ start\ finish\ V\ E} \\
= \quad & \\
& \exists\, \mathit{in.d0\_ff\ start\ finish} \wedge \mathit{reg\ in\ start\ V} \wedge \forall\, \mathit{t.in\ t} = \mathit{E\ t} \\
= \quad & \\
& \mathit{d0\_ff\ start\ finish} \wedge \mathit{reg\ E\ start\ V} \\
= \quad & \\
& \neg \mathit{finish\ 0} \wedge (\forall\, \mathit{t.finish}\ (t+1) = \mathit{start\ t}) \\
& \wedge\, V\ 0 = \mathit{zero} \wedge \forall\, \mathit{t.start\ t} \rightarrow V\ (t+1) = E\ t \mid V\ (t+1) = V\ t \\
\Rightarrow \quad & \\
& \forall\, \mathit{t.start\ t} \rightarrow (\mathit{finish}\ (t+1) \wedge V\ (t+1) = E\ t) \mid \\
& \qquad\qquad (\neg \mathit{finish}\ (t+1) \wedge V\ (t+1) = V\ t) \\
\Rightarrow \quad & \\
& \mathit{assignement\_spec\ start\ finish\ V\ E}
\end{aligned}
$$

Let's do the same for the SKIP statement:

- implementation

$$\forall\, \mathit{start\ finish.skip\ start\ finish} = \mathit{d0\_ff\ start\ finish}$$

- specification

$$\forall\, \mathit{start\ finish.skip\_spec\ start\ finish} = \forall\, \mathit{t.finish}\ (t+1) = \mathit{start\ t}$$

- correctness argument

$$\forall\, \mathit{start\ finish.skip\ start\ finish} \Rightarrow \mathit{skip\_spec\ start\ finish}$$

- proof

$$
\begin{array}{l}
\quad \text{skip start finish} \\
= \\
\quad \text{d0\_ff start finish} \\
= \\
\quad \text{finish } 0 \wedge \forall\, \text{t.finish } (\text{t} + 1) = \text{start t} \\
\Rightarrow \\
\quad \text{skip\_spec start finish}
\end{array}
$$

And now, let's prove correctness of the IF statement:

- implementation

$$
\begin{array}{l}
\forall \text{ start finish Cond S0 S1.} \\
\quad \text{if\_then\_else start finish Cond S0 S1} = \\
\qquad \exists \text{ c c1 s0 s1 r0 r1.} \\
\qquad\quad (\forall \text{ t.not (c t) (c1 t)}) \\
\qquad\quad \wedge\ (\forall \text{ t.c t} = \text{Cond t}) \\
\qquad\quad \wedge\ (\forall \text{ t.and2 (c t) (start t) (s0 t)}) \\
\qquad\quad \wedge\ (\forall \text{ t.and2 (c1 t) (start t) (s1t)}) \\
\qquad\quad \wedge\ \text{S0 s0 r0} \\
\qquad\quad \wedge\ \text{S1 s1 r1} \\
\qquad\quad \wedge\ (\forall \text{ t.or2 (r0 t) (r1 t) (finish t)})
\end{array}
$$

- specification

$$
\begin{array}{l}
\forall \text{ start finish Cond S0 S1.} \\
\quad \text{if\_then\_else\_spec start finish Cond S0 S1} = \\
\qquad \forall \text{ t.Cond t} \rightarrow \text{ S0 start finish} \\
\qquad\qquad\qquad\qquad\quad \mid \text{S1 start finish}
\end{array}
$$

- correctness argument

$$
\begin{array}{l}
\forall \text{ start finish Cond S0 S1.} \\
\quad \text{if\_then\_else start finish Cond S0 S1} \Rightarrow \\
\quad \text{if\_then\_else\_spec start finish Cond S0 S1}
\end{array}
$$

- proof

$$
\begin{array}{l}
\quad \text{if\_then\_else start finish Cond S0 S1} \\
= \\
\quad \exists \text{ c c1 s0 s1 r0 r1.} \\
\qquad\quad (\forall \text{ t.c1 t} = \neg \text{c t}) \\
\qquad \wedge\ (\forall \text{ t.c t} = \text{Cond t}) \\
\qquad \wedge\ (\forall \text{ t.s0 t} = \text{c t} \wedge \text{start t}) \\
\qquad \wedge\ (\forall \text{ t.s1 t} = \text{c1 t} \wedge \text{start t}) \\
\qquad \wedge\ \text{S0 s0 r0} \\
\qquad \wedge\ \text{S1 s1 r1} \\
\qquad \wedge\ (\forall \text{ t.finish t} = \text{r0 t} \vee \text{r1 t}) \\
= \\
\end{array}
$$

$$
\begin{array}{l}
= \\
\quad \exists\, r0\ r1. \\
\qquad\quad S0\ (\lambda\, t.Cond\ t \wedge start\ t)\ r0 \\
\qquad \wedge\ S1\ (\lambda\, t.\neg Cond\ t \wedge start\ t)\ r1 \\
\qquad \wedge\ (\forall\, t.finish\ t = r0\ t \vee r1\ t) \\
= \\
\quad (\forall\, t.Cond\ t \rightarrow S0\ start\ r0 \mid S1\ start\ r1) \\
\quad \wedge\ (\forall\, t.finish\ t = r0\ t \vee r1\ t) \\
\Rightarrow \\
\quad if\_then\_else\_spec\ start\ finish\ Cond\ S0\ S1
\end{array}
$$

Now is time to prove correctness of the *WHILE* statement:

- implementation

$$
\begin{array}{l}
\forall\, start\ finish\ Cond\ S. \\
\quad while\ start\ finish\ Cond\ S = \\
\qquad \exists\, r\ x\ c\ c1\ y. \\
\qquad\quad (\forall\, t.or2\ (start\ t)\ (r\ t)\ (x\ t)) \\
\qquad \wedge\ (\forall\, t.and2\ (x\ t)\ (c1\ t)\ (finish\ t)) \\
\qquad \wedge\ (\forall\, t.not\ (c\ t)\ (c1\ t)) \\
\qquad \wedge\ (\forall\, t.and2\ (x\ t)\ (c\ t)\ (y\ t)) \\
\qquad \wedge\ (\forall\, t.c\ t = Cond\ t) \\
\qquad \wedge\ S\ y\ r
\end{array}
$$

- specification

$$
\begin{array}{l}
\forall\, start\ finish\ Cond\ S. \\
\quad while\_spec\ start\ finish\ Cond\ S = \\
\qquad \exists\, r.S\ (lambda\ t.(start\ t \vee r\ t) \wedge Cond\ t)\ r
\end{array}
$$

- correctness argument

$$
\begin{array}{l}
\forall\, start\ finish\ Cond\ S. \\
\quad while\ start\ finish\ Cond\ S \Rightarrow \\
\quad while\_spec\ start\ finish\ Cond\ S
\end{array}
$$

- proof

$$
\begin{array}{l}
\quad while\ start\ finish\ Cond\ S \\
= \\
\quad \exists\, r\ x\ c\ c1\ y. \\
\qquad\quad (forall\ t.x\ t = start\ t \vee r\ t) \\
\qquad \wedge\ (\forall\, t.finish\ t = x\ t \wedge c1\ t) \\
\qquad \wedge\ (\forall\, t.c1\ t = \neg c\ t) \\
\qquad \wedge\ (\forall\, t.y\ t = x\ t \wedge c\ t) \\
\qquad \wedge\ (\forall\, t.c\ t = Cond\ t) \\
\qquad \wedge\ S\ y\ r \\
=
\end{array}
$$

```
=
    ∃ r.    (forall t.finish t = (start t ∨ r t) ∧ ¬Cond t)
          ∧ S (λ t.(start t ∨ r t) ∧ Cond t) r
⇒
    while_spec start finish Cond S
```

Now it remains to prove correctness of SEQ statement:

- implementation

```
∀ start finish S0 S1.
    seq start finish S0 S1 =
    ∃ x.S0 start x ∧ S1 x finish
```

- specification

```
∀ start finish S0 S1.
    seq_spec start finish S0 S1 =
    ∃ x.S0 start x ∧ S1 x finish
```

- correctness argument

```
∀ start finish S0 S1.
    seq start finish S0 S1 =
    seq_spec start finish S0 S1
```

- proof

```
immediate
```

And finally, let's see correctness of the PAR statement:

- implementation

```
∀ start finish S0 S1.
    par start finish S0 S1 =
    ∃ s0 s1 r0 r1.
          S0 start s0
        ∧ S1 start s1
        ∧ sr0_ff s0 finish r0
        ∧ sr0_ff s1 finish r1
        ∧ (∀ t.and2 (r0 t) (r1 t) (finish t))
```

- specification

```
∀ start finish S0 S1.
    par_spec start finish S0 S1 =
        ∃ f0 f1.S0 start f0 ∧ S1 start f1
```

- correctness argument

```
∀ start finish S0 S1.
    par start finish S0 S1 ⇒
    par_spec start finish S0 S1
```

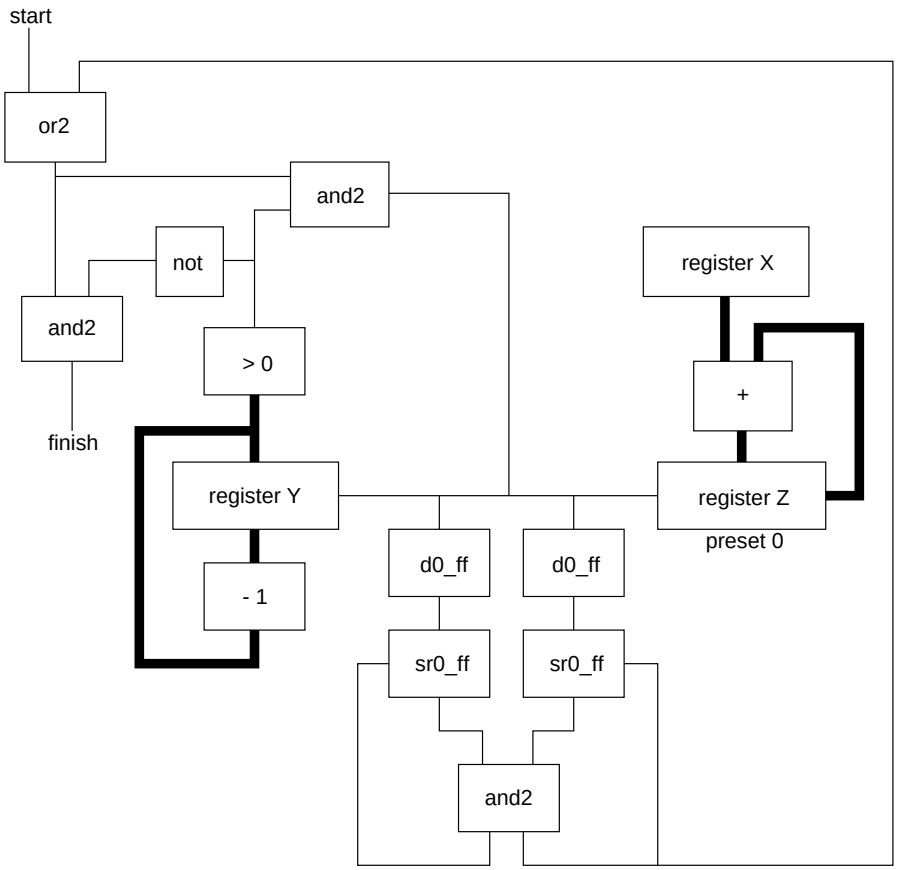


Figure 7.9: Multiplication circuit

- proof

$$
\begin{aligned}
& \mathit{par}\ \mathit{start}\ \mathit{finish}\ S0\ S1 \\
= \\
& \quad \exists\, s0\ s1\ r0\ r1. \\
& \qquad S0\ \mathit{start}\ s0 \\
& \qquad \wedge\ S1\ \mathit{start}\ s1 \\
& \qquad \wedge\ \mathit{sr0\_ff}\ s0\ \mathit{finish}\ r0 \\
& \qquad \wedge\ \mathit{sr0\_ff}\ s1\ \mathit{finish}\ r1 \\
& \qquad \wedge\ (\forall\, t.\mathit{and2}\ (r0\ t)\ (r1\ t)\ (\mathit{finish}\ t)) \\
\Rightarrow \\
& \quad \mathit{par\_spec}\ \mathit{start}\ \mathit{finish}\ S0\ S1
\end{aligned}
$$

Now we have seen that we can really compile programs into circuits. We have proven that the basics of the compiler are correct.

To end the presentation we give an example of an algorithm to see how a compiled circuit appears.

Our algorithm is very simple; it computes multiplication of two integer num-

bers by performing sums.

It can be implemented in Handel and it is:

```
INT  x, y : INPUT
INT  z := 0 : OUTPUT

WHILE y > 0
  PAR
    z := z + x
    y := y - 1
```

It computes in z the product of x and y. The resulting circuit is shown in figure 7.9.

7.4 Using Different Logics

In this section we will describe how to use other logics. Sometimes the classical higher order logic is not expressive enough to model in a simple way complex behaviours.

Probably the most interesting logic for our purposes is temporal logic. It is useful to describe complex timing behaviours like handshake protocols.

The temporal logic we will describe here is technically called *Linear Temporal Logic*.

7.4.1 An Overview of Temporal Logic

Linear temporal logic is based on four temporal operators and three logical connectives:

$\Box$	henceforth
$\Diamond$	eventually
$\bigcirc$	next
$\cup$	until

and

not	logical negation
$\rightarrow$	logical implication
and	logical conjunction

These operators are usually complemented by equality, represented by the Eq operator.

There are specific inference rules for temporal operators, but they are not interesting for us in this presentation. We are interested in:

- representing operators and connectives by means of higher order logic
- using temporal logic to reason about handshaking protocols.

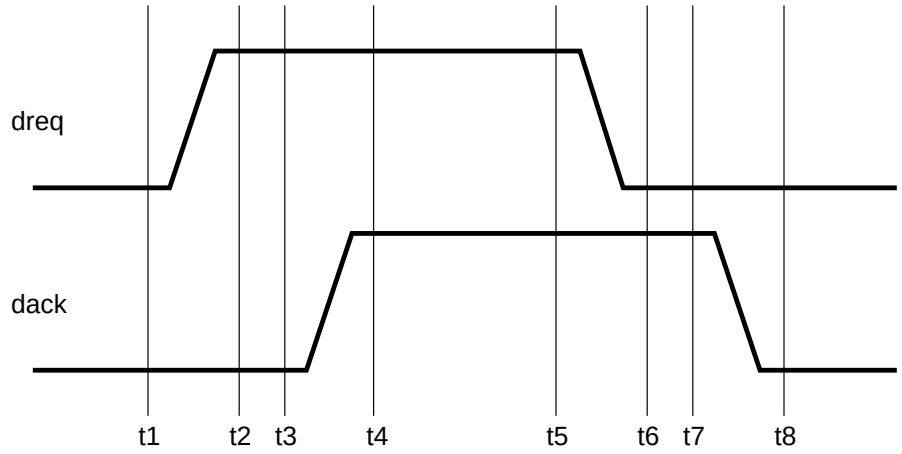


Figure 7.10: Handshaking Timing Diagram

For these reasons we stop here, leaving only the intuitive meaning of connectives: later we will describe precisely the meaning of them in term of higher order logic.

7.4.2 Formalizing Handshake Protocol

Figure 7.10 shows a typical handshaking timing diagram.

The handshaking protocol is specified by a set of temporal logic axioms. Each axiom specify what happens during a time period (from t_i to t_{i+1}).

More precisely:

- from t_1 to t_2

$$not\ dack \rightarrow \Diamond\ dreq$$

  that is, when $dack$ is false, sooner or later, but surely before it turns to true, $dreq$ becomes true.

- from t_2 to t_3

$$dreq \rightarrow dreq \cup dack$$

  that is, when $dreq$ is true, it remains true until also $dack$ becomes true.

- from t_3 to t_4

$$dreq \rightarrow \Diamond\ dack$$

  that is, when $dreq$ is true, eventually also $dack$ will become true.

- from t_4 to t_5

$$dack \rightarrow dack \cup (not\ dreq)$$

  when $dack$ is true then it retains this value at least until $dreq$ becomes false.

- from t_5 to t_6

$$dack \rightarrow \Diamond\ (not\ dreq)$$

when $dack$ is true then surely, sooner or later, $dreq$ becomes false.

- from t_6 to t_7

$$(not\ dreq) \rightarrow (not\ dreq) \cup (not\ dack)$$

that is, when $dreq$ is false, it remains so until also $dack$ becomes false.

- from t_7 to t_8

$$(not\ dreq) \rightarrow \Diamond\ (not\ dack)$$

when $dreq$ is false, then also $dack$ becomes eventually false.

- from t_8 to t_1

$$(not\ dack) \rightarrow (not\ dack) \cup dreq$$

and finally $dack$ stays false until a new handshaking request is generated.

Handshaking protocol is specified by a small, simple and compact set of axioms, where signals are related to each other using only the words "until" and "eventually". This suffices in the specification of most of hardware protocols, and, in this way, timing details are hidden behind temporal operators.

7.4.3 Representing Temporal Logic in HOL

We need to represent temporal logic terms to manipulate them in HOL. This technique is called embedding.

The embedding of linear temporal logic is very simple; we define temporal logic's operators and connectives in HOL:

$$\begin{array}{rcl}
\Box & = & \lambda\ P\ t.\forall\ n.P\ (t+n) \\
\Diamond & = & \lambda\ P\ t.\exists\ n.P\ (t+n) \\
\bigcirc & = & \lambda\ P\ t.P\ (t+1) \\
\cup & = & \lambda\ P\ Q\ t.\forall\ n.(\forall\ m.m < n \Rightarrow \neg Q\ (t+m)) \Rightarrow P\ (t+n) \\
not & = & \lambda\ P\ t.\neg P\ t \\
\rightarrow & = & \lambda\ P\ Q\ t.P\ t \Rightarrow Q\ t \\
and & = & \lambda\ P\ Q\ t.P\ t \wedge Q\ t \\
Eq & = & \lambda\ P\ Q\ t.P\ t = Q\ t
\end{array}$$

To represent truth of temporal logic's terms, we need to define an higher order predicate that says that a particular temporal term is valid.

Its definition is:

$$VALID = \lambda\ P.\forall\ t.P\ t$$

VALID performs the translation from temporal logic to HOL logic. For example,

$$\begin{aligned} & \mathit{VALID}\ (\mathrm{dreq} \rightarrow \Diamond\ \mathrm{dack}) \\ = \\ & \mathit{VALID}\ (\mathrm{dreq} \rightarrow \lambda\ \mathrm{t}.\exists\ \mathrm{n}.\mathrm{dack}\ (\mathrm{t}+\mathrm{n}) \\ = \\ & \mathit{VALID}\ (\lambda\ \mathrm{t}.(\mathrm{dreq}\ \mathrm{t} \Rightarrow \exists\ \mathrm{n}.\mathrm{dack}\ (\mathrm{t}+\mathrm{n}))) \\ = \\ & \forall\ \mathrm{t}.\mathrm{dreq}\ \mathrm{t} \Rightarrow \exists\ \mathrm{n}.\mathrm{dack}\ (\mathrm{t}+\mathrm{n}) \end{aligned}$$

7.4.4 An Application: Asynchronous Memory Interface

The handshaking protocol, explained above, is useful to model an asynchronous memory interface.

We need to refine a bit the protocol specification: we have two system, a *Sender* and a *Receiver* which communicate to each other using the handshaking protocol.

The *Sender* is a process which:

- when generates a request waits until it is acknowledged
- is not allowed to request something until it is receiving an acknowledge
- it assumes that an acknowledge is given until a request is no more active
- it assumes that no acknowledge is given until a request becomes active

The *Sender* can be modeled as in the following definition:

$$\begin{aligned} & \mathit{Sender}(\mathrm{dreq}, \mathrm{dack}) = \\ & \quad \mathit{VALID}\ (\mathrm{dreq} \rightarrow \mathrm{dreq} \cup \mathrm{dack}) \\ & \wedge\ \mathit{VALID}\ (\mathrm{not\ dreq} \rightarrow \mathrm{not\ dreq} \cup \mathrm{not\ dack}) \\ & \wedge\ (\mathit{VALID}\ (\mathrm{dack} \rightarrow \mathrm{dack} \cup \mathrm{not\ dreq}) \Rightarrow \\ & \quad \mathit{VALID}(\mathrm{dack} \rightarrow \Diamond\ \mathrm{not\ dreq})) \\ & \wedge\ (\mathit{VALID}\ (\mathrm{not\ dack} \rightarrow \mathrm{not\ dack} \cup \mathrm{dreq}) \Rightarrow \\ & \quad \mathit{VALID}(\mathrm{not\ dack} \rightarrow \Diamond\ \mathrm{dreq})) \end{aligned}$$

The *Receiver* is a process which:

- when generating an acknowledge wait until the request is retired
- is not allowed to generate an acknowledge when no request is given
- it assumes that a request stays active until it is acknowledged
- it assumes that no request is given until an acknowledge is up

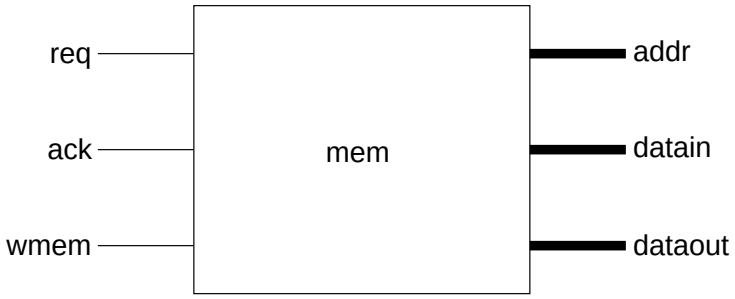


Figure 7.11: Asynchronous Memory

The $Receiver$ can be modeled as in the following definition:

$$\begin{aligned}
&Receiver(dreq, dack) = \\
&\quad VALID\ (dack \rightarrow dack \cup not\ dreq) \\
&\wedge\ VALID\ (not\ dack \rightarrow not\ dack \cup dreq) \\
&\wedge\ (VALID\ (dreq \rightarrow dreq \cup dack) \Rightarrow \\
&\quad\ VALID(dreq \rightarrow \Diamond\ dack)) \\
&\wedge\ (VALID\ (not\ dreq \rightarrow not\ dreq \cup not\ dack) \Rightarrow \\
&\quad\ VALID(not\ dreq \rightarrow \Diamond\ not\ dack))
\end{aligned}$$

What we need to specify is an asynchronous memory. It is a device (see fig 7.11) that takes three inputs, has a state, and produces two outputs.

The req and ack lines are used to control when data are transferred to or from memory; when req is true a request (read or write, depending on the value of $wmem$, false or true) is made to memory; when ack becomes true, this means that the request has been processed and values on buses are ready to be used by external devices.

The memory device takes only one clock cycle to process a request, but it is asynchronous because it holds values on buses until req is driven to false.

It assumes that values on input buses are not changed while req is high.

To define this device using temporal logic, we need some auxiliary predicates:

$$\begin{aligned}
P\ when\ Q &= (Q \rightarrow P) \cup Q \\
stableuntil\ P\ Q &= P \rightarrow P \cup Q
\end{aligned}$$

When a read request has to be satisfied, the action to perform is:

$$\begin{aligned}
&ReadFunc(mem, wmem, addr, datain) = \\
&\quad wmem \mapsto datain \mid fetch\ (mem, addr)
\end{aligned}$$

while, when a write request arrives, the corresponding action is:

$$\begin{aligned}
&WriteFunc(mem, wmem, addr, datain) = \\
&\quad wmem \mapsto store\ (mem, addr, datain) \mid mem
\end{aligned}$$

So, if req is false, memory must not be changed; if req is raising to true, then, assuming $wmem$, $addr$ and $datain$ stable, $dataout$ will become, in the

next clock pulse, equal to the value of ReadFunc and mem equal to the value of WriteFunc. These values must be valid when ack is true and must remain so until it becomes false.

All these actions are performed by a device specified in the following definition:

$$
\begin{aligned}
&\mathrm{ReceiverData}(req, ack, mem, wmem, addr, datain, dataout) = \\
&\quad (\forall\, x.\mathrm{VALID}\ ((\mathrm{not}\ req\ \mathrm{and}\ mem\ \mathrm{Eq}\ x) \rightarrow \bigcirc\ (mem\ \mathrm{Eq}\ x))) \\
&\quad \wedge\ \mathrm{VALID}(\mathrm{not}\ req\ \mathrm{and}\ bigcirc\ req\ \rightarrow \\
&\qquad \bigcirc\ (\mathrm{stableuntil}\ wmem\ ack)\ \mathrm{and} \\
&\qquad \bigcirc\ (\mathrm{stableuntil}\ addr\ ack)\ \mathrm{and} \\
&\qquad \bigcirc\ (\mathrm{stableuntil}\ datain\ ack)\ \rightarrow \\
&\qquad\quad \bigcirc\ ((dataout\ \mathrm{Eq}\ \mathrm{ReadFunc}\ (mem, wmem, addr, datain))\ \mathrm{and} \\
&\qquad\quad (mem\ \mathrm{Eq}\ \mathrm{WriteFunc}\ (mem, wmem, addr, datain)) \\
&\qquad\quad when\ ack))
\end{aligned}
$$

Now asynchronous memory is nothing more than the composition of two devices: ReceiverData and Receiver, which generates the signal ack according to the handshake protocol:

$$
\begin{aligned}
&\mathrm{AsynMemory}(req, ack, mem, wmem, addr, datain, dataout) = \\
&\quad \mathrm{ReceiverData}(req, ack, mem, wmem, addr, datain, dataout) \\
&\quad \wedge\ \mathrm{Receiver}(req, ack)
\end{aligned}
$$

Appendix A

Mathematical Logic Summary

In this appendix we introduce the mathematical logic that HOL uses.

We present both syntax and semantics in a short way. We will not be very formal in semantics' presentation but we prefer to give an intuitive meaning rather than developing all details of a complete formal semantics.

We begin presenting the basic elements of our logic: models, types, terms, inference rules and theories. Then we present what are the basic theories HOL is built on, and the consistency preserving extension mechanism we used so far.

A.1 Universes

A universe $\mathfrak{U}$ is a fixed set of sets. We could safely assume that $\mathfrak{U}$ is a set of the standard Zermelo-Frænkel set theory with the axiom of choice.[1]

$\mathfrak{U}$ has to satisfy the following properties:

- Inhab: $\mathfrak{U} \neq \emptyset \wedge \forall x \in \mathfrak{U}.x \neq \emptyset$.
- Sub: $\forall x \in \mathfrak{U}\ y \subseteq x.y \neq \emptyset \Rightarrow y \in \mathfrak{U}$.
- Prod: $\mathfrak{U}$ is closed under the Cartesian product operator.
- Pow: $\mathfrak{U}$ is closed under the powerset operator.
- Infty: $\mathfrak{U}$ contains a distinguished (i.e. fixed) infinite set I.
- Choice: Exists a distinguished function, called the choice function, $ch \in \prod_{x \in \mathfrak{U}} x$ such that $\forall x.ch(x) \in x$.
- Fun. $\mathfrak{U}$ is closed under the function constructor $\longrightarrow$.[2]
- Unit: $\mathfrak{U}$ contains a distinguished one-element set $1 = \{\emptyset\}$.[3]
- Bool: $\mathfrak{U}$ contains a distinguished two-element set $1 = \{0, 1\}$.[4]

[1] This assumption is not strictly necessary, but we prefer to avoid complications of weaker axiomatic set theories.

[2] This property is implied by Prod, Sub and Pow.

[3] This property could be derived from the preceding ones.

[4] This property could be derived from the preceding ones.

A.2 Types

A type is a syntactical construction with the following form:

$$\sigma ::= \alpha \mid (\sigma_1, \ldots, \sigma_n)\ \mathrm{op}$$

α is a type variable; it stands for an arbitrary set in a given universe.

$(\sigma_1, \ldots, \sigma_n)\ \mathrm{op}$ is the syntax for type constructors. To lighten notation, we introduce some simplifications:

- if $n = 0$, we write op instead of $()\ \mathrm{op}$. In this case, op is a constant type, that is, a fixed set in a given universe. Predefined types are bool and ind, and they denote 2 and I, respectively.
- if $n = 2$ and $\mathrm{op} = \longrightarrow$, we write $\sigma_1 \longrightarrow \sigma_2$ instead of $(\sigma_1, \sigma_2)\ \longrightarrow$. The $\longrightarrow$ type constructor gives, from types σ_1 and σ_2 , the type of functions with domain σ_2 and range σ_1.

The semantics of types is very simple: a type denotes a set in a given universe. Axioms on universes ensure that we can take them as models for our type structures, if we are not allowed to introduce new constructors in any way we may want to.

Predefined constants are covered by Unit, Infty and Bool properties of universes, so they cannot introduce inconsistency in our models. An inconsistency will be a constructor which denotes the empty set, or which could produce a set not in $\mathfrak{U}$ from sets in $\mathfrak{U}$.

A.3 Terms

A term is a syntactical construction with the following form:

$$t_\sigma ::= x_\sigma \mid c_\sigma \mid (t'_{(\sigma_1 \longrightarrow \sigma)}\ t''_\sigma) \mid (\lambda\, x_{\sigma_1}.t'_{\sigma_2})_{(\sigma_1 \longrightarrow \sigma_2)}$$

x_σ is a term variable (variable, in short); it stands for an arbitrary element of type σ.

c_σ is a constant; it stands for a well fixed element of type σ. Predefined constants are:

$$\Rightarrow_{(\mathrm{bool} \longrightarrow \mathrm{bool} \longrightarrow \mathrm{bool})}$$
$$=_{(\alpha \longrightarrow \alpha \longrightarrow \mathrm{bool})}$$
$$\varepsilon_{((\alpha \longrightarrow \mathrm{bool}) \longrightarrow \alpha)}$$

$\Rightarrow$ is classical implication, $A \Rightarrow B$ is false iff A is true and B is false; $=$ is equality, $A = B$ is true iff A and B denote the same element; ε is Hilbert's choice operator, its value is:

$$\varepsilon(x) \text{ denotes } \begin{cases} \mathrm{ch}(x^{-1}(1)) & \text{if } x^{-1}(1) \neq \emptyset \\ \mathrm{ch}(\tau) & \text{otherwise} \end{cases}$$

where x has type $\tau \longrightarrow \mathrm{bool}$ and $x^{-1}(1) = \{y \in \tau \mid x(y) = 1\}$ and ch is the function defined by the Choice axiom in definition of universes.

$(t'_{(\sigma_1 \longrightarrow \sigma)}\ t''_{\sigma})$ denotes function application; t' denotes the function and t'' its argument.

$(\lambda\ x_{\sigma_1}.t'_{\sigma_2})_{(\sigma_1 \longrightarrow \sigma_2)}$ denotes functional abstraction; the term denotes the function whose values are given by the elements denoted by t' with x substituted with the argument.

A.4 Sequents

A sequent is a pair $\langle \Gamma, f \rangle$ where Γ is a finite set of formulae and f is a formula. We define a formula as a term of type bool. Γ is called the set of assumptions, while f is the conclusion.

A model (i.e. a particular universe) satisfies a sequent $\langle \Gamma, f \rangle$ iff any interpretation of free variables that makes all formulae in Γ true, make also f true.

Any variable in a term is free except those bounded in λ-forms: for example in $\lambda\ x.x\ y$, y is free, and x is bound.

A.5 Deductive Systems

A deductive system, or logic, $\mathfrak{D}$ is a set of pairs $\langle L, s \rangle$ where L is a (possibly empty) list of sequents and s is a sequent.

A sequent $\langle \Gamma, t \rangle$ follows from a set of sequents Δ by a deductive system $\mathfrak{D}$ iff there exists sequents $\langle \Gamma_1, t_1 \rangle$, ..., $\langle \Gamma_n, t_n \rangle$ such that:

- $\langle \Gamma, t \rangle = \langle \Gamma_n, t_n \rangle$
- for all i s.t. $1 \leq i \leq n$:
  - $\langle \Gamma_i, t_i \rangle \in \Delta$ or
  - $\langle L_i, \langle \Gamma_i, t_i \rangle \rangle \in \mathfrak{D}$ for some list L_i of members $\Delta \cup \bigcup_{j=1}^{i-1} \langle \Gamma_j, t_j \rangle$.

The sequence $\langle \Gamma_1, t_1 \rangle$, ..., $\langle \Gamma_n, t_n \rangle$ is a proof of $\langle \Gamma, t \rangle$ from Δ in the logic $\mathfrak{D}$.
Normally we write elements of $\mathfrak{D}$ as inference rules:
$\langle [\langle \Gamma_1, t_1 \rangle, \ldots, \langle \Gamma_n, t_n \rangle], \langle \Gamma, t \rangle \rangle$ is written as

$$\frac{\Gamma_1 \vdash t_1 \quad \cdots \quad \Gamma_n \vdash t_n}{\Gamma \vdash t}$$

A.6 The HOL Logic

The deductive system of the HOL logic is specified by eight inference rules:

- Assumption introduction:
$$\frac{}{t \vdash t}$$
- Reflexivity:
$$\frac{}{\vdash t = t}$$

- Beta-conversion:
$$\frac{}{\vdash (\lambda\, x.t_1)\ t_2 = t_1[x := t_2]}$$

- Substitution:
$$\frac{\Gamma_1 \vdash t_1 = t_1' \ \cdots\ \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \ldots \cup \Gamma_n \vdash t[t_1', \ldots, t_n']}$$

- Abstraction:
$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda\, x.t_1) = (\lambda\, x.t_2)}$$
where x is not free in Γ.

- Type instantiation:
$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\alpha_1 := \sigma_1, \ldots, \alpha_n := \sigma_n]}$$

- Discharging an assumption:
$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \Rightarrow t_2}$$

- Modus Ponens:
$$\frac{\Gamma_1 \vdash t_1 \Rightarrow t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

A.7 HOL Theories

A theory is a 3-tuple: $\langle T, C, A\rangle$ where

- T is the set of type constructors.
- C is the set of term constants.
- A is the set of axioms (which are sequents).

It represents a particular deductive system, where types and terms are constructed from T and C alphabets, and inference rules are extended with the following: for all $a \in A$
$$\frac{}{a}$$

The HOL theorem prover is based on three basical theories, which are defined one on the top of the other: MIN, LOG and INIT.

A.7.1 The Theory MIN

MIN is defined to be $\langle\{\mathrm{bool}, \mathrm{ind}\}, \{\Rightarrow, =, \varepsilon\}, \emptyset\rangle$ where bool and ind have arity 0, and $\Rightarrow$, $=$, ε have the types we have already presented.

MIN is the theory which is able to perform proofs using only the basic inference rules, and the minimal alphabet.

A.7.2 The Theory LOG

LOG is an extension of MIN. The set of constants is extended with:

- T_{bool},
- $\forall_{(\alpha \longrightarrow bool) \longrightarrow bool}$,
- $\exists_{(\alpha \longrightarrow bool) \longrightarrow bool}$,
- F_{bool},
- $\neg_{bool \longrightarrow bool}$,
- $\wedge_{bool \longrightarrow bool \longrightarrow bool}$,
- $\vee_{bool \longrightarrow bool \longrightarrow bool}$,
- $One\_One_{(\alpha \longrightarrow \beta) \longrightarrow bool}$,
- $Onto_{(\alpha \longrightarrow \beta) \longrightarrow bool}$, and
- $Type\_Definition_{(\alpha \longrightarrow bool) \longrightarrow (\beta \longrightarrow \alpha) \longrightarrow bool}$.

The behaviour of these symbols is stated by giving the following definitions [5], which are added to the set of axioms:

$$\vdash T = ((\lambda\, x_{bool}.x) = (\lambda\, x_{bool}.x))$$
$$\vdash \forall = (\lambda\, P_{\alpha \longrightarrow bool}.(P = (\lambda\, x.T)))$$
$$\vdash \exists = (\lambda\, P_{\alpha \longrightarrow bool}.P(\varepsilon P))$$
$$\vdash F = (\forall\, x_{bool}.x)$$
$$\vdash \neg = (\lambda\, x.x \Rightarrow F)$$
$$\vdash \wedge = (\lambda\, x_1\ x_2.(\forall\, y.(x_1 \Rightarrow (x_2 \Rightarrow y)) \Rightarrow y))$$
$$\vdash \vee = (\lambda\, x_1\ x_2.(\forall\, y.((x_1 \Rightarrow y) \Rightarrow ((x_2 \Rightarrow y) \Rightarrow y))))$$
$$\vdash One\_One = (\lambda\, f_{\alpha \longrightarrow \beta}.(\forall\, x_1\ x_2.((f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2))))$$
$$\vdash Onto = (\lambda\, f_{\alpha \longrightarrow \beta}.(\forall\, y.(\exists\, x.y = f\ x)))$$
$$\vdash Type\_Definition = (\lambda\, P_{\alpha \longrightarrow bool}\ rep_{\beta \longrightarrow \alpha}.$$
$$((One\_One\ rep) \wedge (\forall\, x.P\ x = (\exists\, y.x = rep\ y))))$$

A.7.3 The Theory INIT

INIT is an extension of LOG, generated by adding the following five axioms to the set of axioms of LOG:

BOOL_CASES_AX	$\vdash \forall\, b.(b = T) \vee (b = F)$
IMP_ANTISYM_AX	$\vdash \forall\, b_1\ b_2.(b_1 \Rightarrow b_2) \Rightarrow ((b_2 \Rightarrow b_1) \Rightarrow (b_1 \Rightarrow b_2))$
ETA_AX	$\vdash \forall\, f_{\alpha \longrightarrow \beta}.(\lambda\, x.f\ x) = f$
SELECT_AX	$\vdash \forall\, P_{\alpha \longrightarrow bool}\ x.P\ x \Rightarrow P(\varepsilon\ P)$
INFINITY_AX	$\vdash \exists\, f_{ind \longrightarrow ind}.One\_One\ f \wedge \neg(Onto\ f)$

It can be proved that INIT is consistent, that is, it cannot derive falsity from true assumptions.

[5] we relax syntax, so $\forall(\lambda\, x_\sigma.t)$ becomes $\forall\, x_\sigma.t$.

A.8 Extensions of Theories

We can extend INIT by adding type constructors, constants and axioms to it, but we are interested in preserving consistency.

We cannot add axioms in any way, and preserve consistency, so we would introduce here some kinds of axioms which do not affect the consistency character of a theory, so they can be safely added to. These classes are important because all axioms we introduced so far in this book fall in them.

A.8.1 Extension by Constant Definition

A constant definition is a formula of the form $c_\sigma = t_\sigma$ such that:

- c does not appear in the constants of current theory.
- t_σ is a term where c does not appear, and which does not contain free variables.
- all type variables in t_σ also occur in σ.

Let $\Upsilon = \langle T, C, A\rangle$ be the current theory, then the extended theory is:

$$\Upsilon^+ = \langle T, C \cup \{c_\sigma\}, A \cup \{c_\sigma = t_\sigma\}\rangle$$

With the given constraints, it can be proved that Υ^+ is consistent iff Υ is.

A.8.2 Extension by Constant Specification

Constant specifications introduce (set of) constants that satisfy arbitrary given (constant) properties.

Formally a constant specification for a theory $\Upsilon = \langle T, C, A\rangle$ is given by the pair:

$$\langle\{c_1, \ldots, c_n, \lambda\, x_1^{\sigma_1} \ldots x_n^{\sigma_n}.t_{bool}\rangle$$

satisfying the following conditions

- $\{c_1, \ldots, c_n\} \cap C = \emptyset$.
- $\lambda\, x_1^{\sigma_1} \ldots x_n^{\sigma_n}.t_{bool}$ is a closed term of Υ.
- for all $1 \leq i \leq n$, type variables in t_{bool} are exactly the same as in σ_i.
- $\exists\, x_1^{\sigma_1} \ldots x_n^{\sigma_n}.t_{bool}$ is a theorem of Υ, i.e. it can be proved in Υ.

The extended theory is defined as:

$$\Upsilon^+ = \langle T, C \cup \{c_1, \ldots, c_n\}, A \cup \{t[x_1 := c_1, \ldots, x_n := c_n]\}\rangle$$

Again it can be proved that Υ^+ is consistent iff Υ is.

A.8.3 Extension by Type Definition

To define a new type, we need

- to specify an existing type.
- to specify a subset of this type.
- to prove that this subset is not empty.
- to specify that the new type is represented by (i.e. it is isomorphic to) this subset.

Formally a type definition is a triple

$$\langle(\alpha_1, \ldots, \alpha_n)\ op, \sigma, p_{\sigma \longrightarrow bool}\rangle$$

satisfying the following conditions in the theory $\Upsilon = \langle T, C, A\rangle$:

- $op \notin T$.
- σ is a type of Υ such that $\{\alpha_1, \ldots, \alpha_n\}$ contains all type variables in σ.
- $p_{\sigma \longrightarrow bool}$ is a closed term in Υ whose type variables are in $\{\alpha_1, \ldots, \alpha_n\}$.
- $\exists\, x_\sigma.p\ x$ is a theorem of Υ.

The extended theory is defined as:

$$\Upsilon^+ = \langle T \cup \{op\}, C, A \cup \{\exists\, f_{(\alpha_1,\ldots,\alpha_n)\ op \longrightarrow \sigma}.\mathrm{Type\_Definition}\ pf\}\rangle$$

Again it can be proved that such an extension preserves consistency.

Appendix B

ML Summary

In this appendix we want to present a summary of the HOL version of the ML language.

We begin presenting a summary of the ML syntax, without types. Then we present types and finally a brief description of the principal predefined functions.

B.1 ML Syntax

Here is the basic kernel of ML syntax.

declaration	:=	let binding
	|	letref binding
	|	letrec binding
binding	:=	pattern = expression
	|	var $pattern_1 \ldots pattern_n$ = expression
	|	$binding_1$ and ... and $binding_n$
pattern	:=	_
	|	constant_expression
	|	constructor pattern
	|	$pattern_1$,$pattern_2$
	|	$pattern_1$,$pattern_2$
	|	[]
	|	[$pattern_1$;...;$pattern_n$]
expression	:=	constant_expression
	|	variable
	|	constructor
	|	$expression_1$ $expression_2$
	|	prefix expression
	|	$expression_1$ infix $expression_2$

| pattern $\underline{:=}$ expression
| $\underline{\text{failwith}}$ expression
| $\underline{\text{if}}$ expression$_1$ {$\underline{\text{then}}|\underline{\text{loop}}$} expression$'_1$
$\vdots$
{$\underline{\text{if}}$ expression$_n$ {$\underline{\text{then}}|\underline{\text{loop}}$} expression$'_n$}
{{$\underline{\text{then}}|\underline{\text{loop}}$} expression}
| expression {$\underline{??}|\underline{!!}$} expression$_1$ expression$'_1$
$\vdots$
{{$\underline{??}|\underline{!!}$} expression$_n$ expression$'_n$}
{{$\underline{?}|\underline{!}|\underline{?\backslash}$ identifier$|\underline{!\backslash}$ identifier} expression}
| $\underline{\text{while}}$ expression$_1$ $\underline{\text{do}}$ expression$_2$
| expression$_1\underline{;}\ldots$expression$_n$
| $\underline{[}$expression$_1\underline{;}\ldots\underline{;}$expression$_n\underline{]}$
| declaration $\underline{\text{in}}$ expression
| $\underline{\backslash}$pattern$_1\ldots$pattern$_n\underline{.}$expression
| $\underline{\text{fun}}$ pattern$_1\underline{.}$expression$_1\underline{|}\ldots\underline{|}$pattern$_n\underline{.}$expression$_n$

- **Declarations**: extend environment in the following way:
  - let b declares the variables specified in b to be ordinary (i.e. non assignable) variables and binds each one to the corresponding value produced by evaluating b.
  - letref b declares the variables specified in b to be assignable and binds each one to the corresponding value produced by evaluating b.
  - letrec b is similar to let b except that: b must consist only of function definitions and these functions are made mutually recursive.
- **Bindings**: evaluate to a set of variable-value pairs or fail. Specifically
  - $p = e$ produces a set of variable-value pairs or fails
  - id $p_1 \ldots p_n = e$ is just an abbreviation for id $= \backslash p_1 \ldots p_n.e$
  - $p_1 = e_1$ and$\ldots$and $p_n = e_n$ is just an abbreviation for $p_1, \ldots, p_n = e_1, \ldots, e_n$
- **Patterns**: when a pattern p is matched with a value E, it can fail or it can return a set of variable-value pairs
  - _ always succeeds and returns $\emptyset$
  - ce if E is equal to ce, returns $\emptyset$, otherwise fails
  - var always succeeds and returns $\{\langle var, E\rangle\}$
  - con p if E = con e, the result is given by matching p with e, otherwise fails
  - $p_1.p_2$ if $E = e_1.e_2$, the result is the union of the results of matching p_1 with e_1 and p_2 with e_2, otherwise fails.

- p_1, p_2 if $E = e_1, e_2$, the result is the union of the results of matching p_1 with e_1 and p_2 with e_2, otherwise fails.
- $[p_1; \ldots; p_n]$ if $E = [e_1; \ldots; e_2]$, the result is the union of the results of matching p_i with e_i, for $1 \leq i \leq n$ and p_2 with e_2, otherwise fails.

- **Expressions**: evaluate to a value. Their evaluation may not terminate. Evaluation acts as follows:
  - ce returns the value ce.
  - var returns the value which is associated with var in the environment.
  - con the value associated with con is returned.
  - e_1 e_2 first, e_1 and e_2 are evaluated, then the result of applying the value of e_1 to that of e_2 is returned.
  - px e first, e is evaluated, then the result of applying px to the value of e is returned. px can be not, $-$ or do.
  - e_1 ix e_2 first, e_1 and e_2 are evaluated, in this order, then the result of applying ix to the values of e_1 and of e_2 is returned. Exception to this rule are & and or operators.
  - p := e Every variable in p must be assignable. The effect of assignement is to update these variable with values from the evaluation of e, if this value matches p. Otherwise fails. The value of the whole expression is the value of e.
  - $failwith$ e e is evaluated and a failure with e's value is generated.
  - $\underline{\text{if}}$ e_1 $\{\underline{\text{then}}|\underline{\text{loop}}\}$ e'_1
    $\vdots$
    $\{\underline{\text{if}}$ e_n $\{\underline{\text{then}}|\underline{\text{loop}}\}$ $e'_n\}$
    $\{\{\underline{\text{then}}|\underline{\text{loop}}\}$ $e'\}$

    e_1,...,e_n are evaluated in turn. One of them, e_m say, returns true. If e_m is followed by $then$, the value of e'_m is returned. If e_m is followed by $loop$, e'_m is evaluated for its side effects, and the return value will be the value of the re-evaluation of the whole expression.

    If e_1,...,e_n evaluate all to false, then e' is evaluated. If it is preceded by $then$, that is the return value, otherwise the value to return is the value of the re-evaluation of the whole expression. If e' is not present, the value $void$ is returned.
  - e $\{\underline{??}|\underline{!!}\}$ e_1 e'_1
    $\vdots$
    $\{\{\underline{??}|\underline{!!}\}$ e_n $e'_n\}$
    $\{\{\underline{?}|\underline{!}|\underline{?\backslash}$ $id|\underline{!\backslash}$ $id\}$ $e'\}$

    e is evaluated and if this succeeds, its value is returned. Otherwise it fails with value str. Then e_1,...,e_n are evaluated in order until one of them, e_m say, returns a string list containing str.

If e_m is preceded by ??, the return value is given by the evaluation of e'_m. Otherwise e'_m is evaluated, and the return value is given by the re-evaluation of the whole expression.

If such an e_m cannot be found and the last option in the expression's syntax is absent, its evaluation fails with the same value as e's.

Otherwise e is evaluated, and if it is preceded by ?\ id or !\ id, its environment is extended with a variable id bound to str. If e' is preceded by a ? form, the return value of the whole expression is e's, otherwise it is the value of the re-evaluation of the whole expression.

- *while* e_1 do e_2 returns () if it succeeds. First e_1 is evaluated; if its value is true then e_2 is evaluated and, then, the whole construct again. If e_1's value is false, evaluation stops.
- $e_1;\ldots;e_n$ e_1,...,e_n are evaluated in that order, and if they succeed, the value of e_n is returned.
- $[e_1;\ldots;e_n]$ e_1,...,e_n are evaluated in that order, and the list of their results is returned. If $n = 0$ then [], the empty list, is returned.
- d *in* e d is evaluated, and the value of e in the extended environment is returned. Environment extension is local to e.
- $\backslash p_1 \ldots p_n.e$ always succeeds. Evaluates to a curried function which takes p_1,...,p_n as parameters and returns the value of e in an environment where appropriate bindings of p_i are made.
- *fun* $p_1.e_1|\ldots|p_n.e_n$ evaluates to that function which, when applied to some value E, matches p_i in turn to E, until one, p_m say, succeeds. Then its value is the value of e_m. If no p_i matches E, then the construct fails with value '*pattern*'.

B.2 ML Types

Types have the following syntax:

types	:=	standard_types
	|	types # types
	|	types + types
	|	types -> types
standard_types	:=	void
	|	int
	|	bool
	|	string

```
                    |  type_variables
                    |  type_constructors
                    |  type_arguments type_constructors
                    |  ( types )
type_variables      :=  any identifier preceded by one or more*
type_constructors   :=  identifiers
type_arguments      :=  standard_types
                    |  (types,...,types)
```

Types:

- $t_1 \# t_2$ is the cartesian product of t_1 and t_2.
- $t_1 + t_2$ is the disjoint sum of t_1 and t_2.
- $t_1 -> t_2$ is the function type with domain t_2 and range t_1.
- void denotes the mathematical set $\{()\}$.
- int denotes the mathematical set of (positive and negative) integers.
- bool denotes the mathematical set $\{true, false\}$.
- string denotes the set of all ASCII strings.
- tvar denotes an arbitrary type.

Here we will not present type constructions nor the way to assign types to ML syntax, because this work is beyond the scope of this book. If you are interested in, refer to the bibliography at the end of this book.

B.3 Some Predefined Functions

The product type is mathematically defined by the two standard projectors:

```
fst : * # ** -> *
snd : * # ** -> **
```

The following property holds: if x has type a#b,

$$x = (\text{fst } x, \text{snd } x)$$

The sum type is mathematically defined by the two standard injections:

```
inl : *  -> * + **
inr : ** -> * + **
```

Equality is defined as an infix operator:

```
= : * # ** -> bool
```

Other infix and prefix operators are:

```
not : bool -> bool
*, /, +, - : int # int -> int
>, < : int # int -> bool
```

Another important operator is function composition:

```
o : ((* -> **) # (*** -> *)) -> (*** -> **)
```

Lists are finite, and homogeneous in type. Their fundamental operations are construction, append, element extraction, length:

```
. : * # * list -> * list
@ : * list # * list -> * list
el : int -> * list -> *
length : * list -> int
```

The most fundamental operation on strings is concatenation

```
^ : string -> string -> string
```

Appendix C

HOL Summary

In this appendix we will describe all the HOL objects we have used when we have written this text. They are ordered by class. We describe functions, which let the user to define a theory, to define symbols, to setup goals, and to apply tactics. Then we describe tactics, tacticals, inference rules and conversions.

C.1 Functions

Here we describe functions, that are really ML functions, but are also the interface between user and theorem prover.

C.1.1 Environment

The three basic actions we can do on proof environments are:

- setting the theorem prover in *draft mode*
- setting the theorem prover in normal mode
- loading a library.

Draft mode is the HOL state in which we can extend the logic. We always extended HOL logic when we created new theories. Extending the logic means to add new symbols along with their definitions, and to add theorems we could recall when we load the theory.

The function we use to set HOL in draft mode is:

```
new_theory ‘theory name‘;;
```

To close a draft session, and to return to normal proof mode, we close the theory. This action writes the theory on a file. The function has the following syntax:

```
close_theory ();;
```

To load a library, we will use the following function:

```
load_library ‘library name‘;;
```

C.1.2 Goal Management

To begin a backward proof, we need to set the initial goal. We can do this using the following function:

```
set_goal([list of assumptions],"goal formula");;
```

Normally we have no assumptions, so the list is void.

To manipulate the goal we need a function to expand a tactic on it. The name of such a function is $expand$, but we always used the abbreviation:

```
e(TACTIC);;
```

C.1.3 Defining Symbols

To define a new symbol we use the following HOL syntax:

```
let DEFINITION_THEOREM = new_definition(
        ‘NAME OF THE DEFINITION‘,
        "defining formula");;
```

Here DEFINITION_THEOREM is the name of the ML symbol which value is the axiom we called $defining\ formula$. The HOL name of this axiom is NAME OF THE DEFINITION. This name is used to archive the definition in the current HOL theory, the one we create with new_theory.

A very important point to remember about definitions is that their axiom must have the form:

$$\forall x_1, \ldots, x_n.symbol\ x_1\ \ldots\ x_n = term\ x_1\ \ldots\ x_n$$

where $symbol$ is the name of the function (constant if $n = 0$) that we are defining, and $term$ is its value. We can regard $symbol(t_1, \ldots, t_n)$ as an abbreviation for $term[x_1/t_1, \ldots, x_n/t_n]$, so this kind of axioms don't affect consistency of the theory.

Sometimes we want to define new symbols by induction over some structure. We used the following construct to use the list induction:

```
let DEFINITION_THEOREM = new_list_rec_definition
        (‘NAME OF THE DEFINITION‘,
         "defining formula");;
```

Here the defining formula must have the following form:

$$\mathrm{symbol}\ [] = \mathrm{term} \wedge \mathrm{symbol}\ (\mathrm{CONS}\ x\ r) = F\ x\ r\ (\mathrm{symbol}\ r)$$

To define functions, we used also recursion over natural numbers. We are limited to primitive recursive functions, and the HOL syntax is:

```
let DEFINITION_THEOREM = new_prim_rec_definition
        (‘NAME OF THE DEFINITION‘,
         "defining formula");;
```

Here the defining formula must have the following form:

$$\mathrm{symbol}\ 0 = \mathrm{term} \wedge \mathrm{symbol}\ (\mathrm{SUC}n) = F\ n\ (\mathrm{symbol}\ n)$$

C.1.4 Compact Proof Generation

Often we need to prove a lemma inside a larger proof. We need a way to generate a theorem without interrupting the main proof.

The simplest way is the following:

```
let LEMMA = TAC_PROOF( goal , tac );;
```

Here $goal$ is the term representing the theorem we want to prove; tac is the (compound) tactic that proves it, and the resulting theorem will be stored in the ML identifier $LEMMA$.

Similar is the following function:

```
let LEMMA = prove_thm( name , goal , tac );;
```

The only difference from TAC_PROOF is that the so generated theorem is stored in the current theory with the given name.

C.1.5 Type Management

Sometimes we need to specialize a theorem for a particular type. When we prove a theorem where type variables occur, we need a way to instance those variables to a specific type, in order to apply that theorem somewhere. The right function to do this is:

```
let SPEC_THM = INST_TYPE
        [(typevar1,type1), ... ,(typevarN,typeN)]
        ORIG_THM
```

With this function we get in SPEC_THM a version of ORIG_THM, where type variables $typevar1, \ldots, typevarN$ are substituted with $type1, \ldots, typeN$.

C.2 Tactics

Tactics are applied to the current (sub)goal using the e function. They performs backward proofs, that is, they apply a logical inference rule whose conclusion match the goal, generating its premises as subgoals.

C.2.1 ALL_TAC

$$\text{ALL\_TAC}: tactic$$

This tactics never fails. When we apply it to the goal G, the resulting goal is G. It implements the logical inference rule:

$$G \vdash G$$

This tactic is interesting because it is the identity element of the THEN tactical.

C.2.2 ASM_REWRITE_TAC

$$\text{ASM\_REWRITE\_TAC}: thm\ list \longrightarrow tactic$$

This tactic rewrites the goal top-down, until no more matches are found. It uses as rewriting equations: basic rewrites, assumptions, the theorem list supplied by the user.

C.2.3 BETA_TAC

$$\text{BETA\_TAC}: tactic$$

This tactic beta-reduces all subterms with form $(\lambda x.A)B$ in the conclusion of the current goal. Formally, it implements the following logical inference rule:

$$F((\lambda x.A)B) \vdash F(A[x/B])$$

C.2.4 COND_CASES_TAC

$$\text{COND\_CASES\_TAC}: \text{tactic}$$

This tactic is used to create a case split on a conditional expression in the goal. Formally it implements the following logical inference rule:

$$F(t \Rightarrow u \mid v) \vdash (t \vdash F(u)) \wedge (\neg t \vdash F(v))$$

C.2.5 CONV_TAC

$$\text{CONV\_TAC}: \text{conv} \longrightarrow \text{tactic}$$

This tactic applies a conversion to the conclusion of the current goal. That is, if conv maps a term t to the theorem $t = t'$, then this tactic transform the goal formula in the following way:

$$F(t) \vdash F(t')$$

C.2.6 EQ_TAC

$$\text{EQ\_TAC}: \text{tactic}$$

This tactic reduces equivalence, i.e. equality between boolean terms, to forward and backward implication. Formally:

$$A = B \vdash (\vdash A \Rightarrow B) \wedge (\vdash B \Rightarrow A)$$

C.2.7 EXISTS_TAC

$$\text{EXISTS\_TAC}: \text{term} \longrightarrow \text{tactic}$$

This tactic reduces an existencially quantified goal by replacing the quantified variable with the given witness. Formally:

$$\exists x.A \vdash A[x/t]$$

Note that free variables in t could be renamed to avoid capturing.

C.2.8 GEN_TAC

$$\text{GEN\_TAC}: \text{tactic}$$

This tactic reduces an universal quantified goal by replacing the variable with a fresh one. Formally:

$$\forall x.A \vdash A[x/y] \qquad\qquad y \notin \mathit{FreeVars}(\lambda x.A)$$

C.2.9 INDUCT_TAC

$$\text{INDUCT\_TAC}: \text{tactic}$$

This tactic applies induction on natural numbers. Formally:

$$\forall n.A(n) \vdash (\vdash A(0)) \wedge (A(n) \vdash A(\text{SUC}n))$$

C.2.10 LIST_INDUCT_TAC

$$\text{LIST\_INDUCT\_TAC}: \text{tactic}$$

This tactic applies induction on lists. Formally:

$$\forall l.A(l) \vdash (\vdash A([])) \wedge (A(l) \vdash \forall x.A(\text{CONS } x\ l))$$

C.2.11 ONCE_REWRITE_TAC

$$\text{ONCE\_REWRITE\_TAC}: \text{thm list} \longrightarrow \text{tactic}$$

This tactic rewrites a goal only once. It uses as rewriting equations the basic rewritings, with the user supplied theorems.

C.2.12 REWRITE_TAC

$$\text{REWRITE\_TAC}: \text{thm list} \longrightarrow \text{tactic}$$

This tactic rewrites a goal. It uses as rewriting equations the built-in tautologies, and the user supplied theorems. It rewrites the goal until there are no more matches.

C.2.13 RULE_ASSUM_TAC

$$\text{RULE\_ASSUM\_TAC}: (\text{thm} \longrightarrow \text{thm}) \longrightarrow \text{tactic}$$

This tactic maps an inference rule over all the assumptions of a goal. It it used to perform forward reasoning from assumptions.

C.2.14 STRIP_TAC

$$\text{STRIP\_TAC}: \text{tactic}$$

This tactic is used to reduce a goal by application of the right tactics to remove to outermost functor if this is among $\forall$, $\neg$, $\wedge$, $\Rightarrow$.

C.2.15 TAUT_TAC

$$\text{TAUT\_TAC}: \text{tactic}$$

This tactic is defined in the library taut. It checks tautologies. If the current goal is a propositional formula, and it is semantically true, then this tactic proves it.

C.2.16 UNDISCH_TAC

$$\text{UNDISCH\_TAC}: \text{term} \longrightarrow \text{tactic}$$

This tactic undischarges the specified assumption. Formally:

$$(\{\ldots, A, \ldots\} \vdash B) \vdash (\{\ldots\} \vdash A \Rightarrow B)$$

C.3 Tacticals

Tacticals are functions on tactics, used to combine them, or to change their behaviour.

C.3.1 THEN

$$\text{THEN}: (\text{tactic} \longrightarrow (\text{tactic} \longrightarrow \text{tactic}))$$

This tactical implements sequencing. The semantic of T1 THEN T2 is: apply T1 to the current goal, if it solve it, stop, else apply T2 to every subgoal generated by T1.

C.3.2 REPEAT

$$\text{REPEAT}: (\text{tactic} \longrightarrow \text{tactic})$$

This tactical implements unconditional repetition. The semantic of REPEAT T is: apply T to the goal, and, while it succeeds, continues applying REPEAT T to all generated subgoals.

C.4 Inference Rules

Inference rules are used to reason in the forward direction. They are functions that take already proved theorems and, if needed, some parameters, and produce another theorem which can be derived from the given ones by means of a (chain of) logical inference rule.

C.4.1 CONJUNCT1

$$\text{CONJUNCT1}: \text{thm} \longrightarrow \text{thm}$$

This inference rule is the implementation of the following logical inference rule:

$$(A \wedge B) \vdash A$$

C.4.2 CONJUNCT2

$$\text{CONJUNCT2}: \text{thm} \longrightarrow \text{thm}$$

This inference rule is the implementation of the following logical inference rule:

$$(A \wedge B) \vdash B$$

C.4.3 CONV_RULE

$$\text{CONV\_RULE}: \text{conv} \longrightarrow \text{thm} \longrightarrow \text{thm}$$

This inference rule transforms a conversion into an inference rule. If conv applied to t gives $t = t'$, then CONV_RULE conv infers $\vdash t'$ from $\vdash t$.

C.4.4 GSPEC

$$\text{GSPEC}: \text{thm} \longrightarrow \text{thm}$$

Specialize the conclusion of a theorem with a unique variable. Formally

$$\forall x_1, \ldots, x_n.A \vdash A[x_1/x'_1, \ldots, x_n/x'_n]$$

where x'_i is a fresh variable for all the original formula.

C.4.5 SPEC

$$\text{SPEC}: \text{term} \longrightarrow \text{thm} \longrightarrow \text{thm}$$

Specialize the conclusion of a theorem with the given term. Formally

$$\forall x.A \vdash A[x/t]$$

C.4.6 SYM

$$\text{SYM}: \text{thm} \longrightarrow \text{thm}$$

This inference rule swaps left-hand and right-hand sides of an equation. Formally:

$$A = B \vdash B = A$$

C.5 Conversions

Conversions are functions that, given an instance, produce an equational theorem which can be used to *convert* the given instance to something else.

C.5.1 ARITH_CONV

$$\text{ARITH\_CONV}: conv$$

This conversion is defined in the $arith$ library. It is a partial decision procedure for the Presburger arithmetic, that is the one with the usual operators, along with the order relations.

C.5.2 DEPTH_CONV

$$\text{DEPTH\_CONV}: conv \longrightarrow conv$$

This function applies a conversion repeatedly to all subterms of a term in a bottom-up order.

C.5.3 FORALL_AND_CONV

$$\text{FORALL\_AND\_CONV}: conv$$

This conversion moves a universal quantification inwards through a conjunction. It produces theorems of the form:

$$(\forall x.A \wedge B) = (\forall x.A) \wedge (\forall x.B)$$

C.5.4 let_CONV

$$\text{let\_CONV}: conv$$

This conversion evaluates let term in HOL logic. It produces theorems of the form:

$$(\text{let } x_1 = t_1 \text{ and } \ldots \text{ and } x_n = t_n \text{ in } A) = A[x_1/t_1, \ldots, x_n/t_n]$$

C.5.5 num_CONV

$$num\_CONV: conv$$

This conversion provide definitional axioms for non-zero numerals: It produces theorem of the form

$$Y = (\text{SUC } X)$$

where $Y = X + 1$.

C.5.6 ONCE_REWRITE_CONV

$$\text{ONCE\_REWRITE\_CONV}: conv \longrightarrow conv$$

Applies a given conversion once to the first suitable subterm encountered in top-down order.

C.5.7 TOP_DEPTH_CONV

$$\text{TOP\_DEPTH\_CONV}: conv \longrightarrow conv$$

Applies top-down a given conversion to all subterms, retraversing changed ones.

C.5.8 TAUT_CONV

$$\text{TAUT\_CONV}: conv$$

This conversion is defined in the $taut$ library. It is a decision procedure for propositional calculus.

Bibliography

[1] P. Andrews. *An Introduction to Higher-Order Logic: to Truth through Proof.* Academic Press, New York, 1986.

[2] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North Holland, Amsterdam, 1984.

[3] J. Barwise. *Handbook of Mathematical Logic.* North Holland, Amsterdam, 1977.

[4] M. Benini and M. Vaccari. Una dimostrazione di correttezza per il circuito SCRAMBLER nella versione parallela. Technical report, SGS-Thomson, June 1994.

[5] G. Birtwistle and B. Graham. Verifying SECD in HOL. In J. Staunstrup, editor, *Proceedings of the IFIP TC10/WG10.5 Summer School on Formal Methods for VLSI Design*, Amsterdam, 1990. North Holland.

[6] G. Birtwistle and P. Subrahmanyam, editors. *VLSI Specification, Verification and Synthesis.* Kluwer, Norwell, Mass., 1988.

[7] G. Birtwistle and P. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving.* Springer Verlag, New York, 1989.

[8] G. Bochman. Hardware specification with temporal logic. *IEEE Transaction on Computer*, C-31, No 3:223–231, March 1982.

[9] A. Camilleri, M. Gordon, and T. Melham. Hardware verification using higher-order logic. In *From HDL Descriptions to Guaranteed Correct Circuit Designs.* Elsevier Science Publishers B.V. (North Holland), 1987.

[10] A. J. Camilleri. *Executing Behavioural Definitions in Higher-Order Logic.* PhD thesis, Cambridge University, February 1988. Technical Report No 140, Computer Laboratory, University of Cambridge.

[11] S-K. Chin. Verifying arithmetic hardware in higher-order logic. In *Proceedings of the ACM/IEEE 1991 Int. Workshop on Higher-order Logic Theorem Proving System and its Applications*, Davis, CA, 1991.

[12] A. Church. A formulation of the simple theory of types. *Journal of Symbolic logic*, 5, 1940.

[13] L. J. M. Cleasen, editor. *Formal VLSI Correctness Verification, VLSI Design Methods II.* North Holland, Amsterdam, 1990.

[14] L. J. M. Cleasen, editor. *Formal VLSI Specification and Synthesis, VLSI Design Methods I.* North Holland, Amsterdam, 1990.

[15] A. Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwistle and P. Subrahmanyan, editors, *VLSI Specification, Verification and Synthesis*, Boston, 1988. Kluwer Academic Publishers.

[16] A. Cohn. Correctness properties of the Viper block model: The second level. In G. Birtwistle and P. Subrahmanyan, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, New York, 1989. Springer Verlag.

[17] A. Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5, 1989.

[18] A. Cohn and M. Gordon. A mechanized proof of correctness of a simple counter. Techincal Report 94, Computer Laboratory, Cambridge University, 1986.

[19] G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson, and C. Wordsworth. *The ML Handbook.* INRIA, 1986.

[20] M. Gordon. A model of register transfer systems with applications to microcode and vlsi correctness. Internal Report CSR-82-81, Department of Computer Science, University of Edinburgh, 1981.

[21] M. Gordon. LCF-LSM. Report 41, Computer Laboratory, Cambridge University, 1983.

[22] M. Gordon. Proving a computer correct using the LCF-LSM hardware verification system. Report 42, Computer Laboratory, Cambridge University, 1983.

[23] M. Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, Computer Laboratory, Cambridge University, July 1985. (Revised version).

[24] M. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. Subrahmanyan, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, Amsterdam, 1986. North Holland.

[25] M. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, Norwell, Massachusetts, 1988.

[26] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic.* Cambridge University Press, 1993.

[27] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.* Lecture Notes in Computer Science. Springer Verlag, 1979.

[28] M. J. C. Gordon. *Programming Language Theory and Its Implementation.* Prentice Hall, London, 1988.

[29] B. Graham. SECD: The design and verification of a functional microprocessor. Master's thesis, Computer Science Department, University of Calgary, 1990.

[30] B. Graham. *The SECD Microprocessor, A Verification Case Study.* Kluwer Academic Publisher, Boston, 1992.

[31] B. Graham and G. Birtwistle. Formalising the design of an SECD chip. In *Proceedings of the Cornell Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, number 408 in LNCS, pages 40–66, New York, 1990. Springer Verlag.

[32] K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In G. Milne and P. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213. North Holland, Amsterdam, 1986.

[33] J. M. J. Herbert. Temporal abstraction of digital designs. Technical Report 122, Computer Laboratory, University of Cambridge, 1988.

[34] W. A. Hunt. *FM8501, A Verified Microprocessor.* PhD thesis, University of Texas, Austin, December 1985. Report No 47, Institute for Computing Science.

[35] INRIA. ML manual. Technical report, INRIA, Grenoble, 1988.

[36] H. Jifeng, I. Page, and J. Bowen. Towards a provably correct hardware implementation of Occam. In *Lecture Notes in Computer Science*, volume 683, Amsterdam, 1993. Springer Verlag. Also, in G. Milne and L. Pierre, eds., Correct Hardware Design and Verification Methods, Proc. IFIP WG10.2 Advanced Research Working Conference, CHARME '93.

[37] J. Joyce. Formal verification and implementation of a microprocessor. In G. Birtwistle and P. Subrahmanyan, editors, *VLSI Specification, Verification and Synthesis*, Boston, 1988. Kluwer Academic Publishers. Also, Proceedings of a workshop, 12–16 January, 1987.

[38] J. Joyce. Using higher-order logic to specify computer hardware and architecture. In D. Edwards, editor, *Design Methodologies for VLSI and Computer Architecture*, Amsterdam, 1989. North Holland. Also, Procs. of the IFIP TC10 Working Conference on Design Methodology in VLSI and Computer Architecture, Pisa, Italy, 19–21 September, 1988.

[39] J. Joyce, G. Birtwistle, and M. Gordon. Proving a computer correct in higher-order logic. Report 100, Computer Laboratory, Cambridge University, 1986.

[40] J. Joyce, E. Liu, J. Rushby, N. Shankar, and R. Suaya. From formal verification to silicon compilation. In *COMPCONSpring '91*, pages 450–455. IEEE Comput. Soc. Press, Los Alamitos,CA, 1991.

[41] D. May. Compiling Occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*. Addison Wesley, 1990.

[42] T. Melham. Abstraction mechanisms for hardware verification. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291. Kluwer, Norwell, Massachusetts, 1988.

[43] T. Melham. Using recursive types to reason about hardware in higher-order logic. In G. Milne, editor, *IFIP WG 10.2 International Working Conference on the Fusion of Hardware Design and Verification*, pages 26–49, Glasgow, 1988. University of Strathclyde.

[44] G. Milne and P. Subrahmanyam, editors. *Formal Aspects of VLSI Design*. North Holland, Amsterdam, 1986.

[45] L. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[46] The HOL System. Description. Technical Report 146, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1989.

[47] The HOL System. Reference manual. Technical Report 146, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1989.

[48] The HOL System. Tutorial. Technical Report 146, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1989.

[49] Booth, L. Taylor. *Sequential Machines and Automata Theory*. John Wiley & Sons, New York, 1967.

[50] P. Windley. *The Verification of AVM-1*. Cse-90-21, Computer Science, Davis, 1990.

[51] P. Windley, K. Levitt, and G. Cohen. Formal proof of the AVM-1 microprocessor using the concept of generic interpreters. NASA Contractor Report 187491, NASA, Langley, 1991.