Interactive Formal Verification
12: Modelling Hardware

Tjark Weber
(Slides: Lawrence C Paulson)
Computer Laboratory
University of Cambridge
Basic Principles of Modelling
Basic Principles of Modelling

• Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).
Basic Principles of Modelling

- Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).
- Whenever possible, use *definitions* — not axioms!
Basic Principles of Modelling

• Define mathematical abstractions of the objects of interest (systems, hardware, protocols,...).
• Whenever possible, use definitions — not axioms!
• Ensure that the abstractions capture enough detail.
  • Unrealistic models have unrealistic properties.
  • Inconsistent models will satisfy all properties.
Basic Principles of Modelling

• Define *mathematical abstractions* of the objects of interest (systems, hardware, protocols,...).

• Whenever possible, use *definitions* — not axioms!

• Ensure that the abstractions capture enough detail.
  • Unrealistic models have unrealistic properties.
  • Inconsistent models will satisfy *all* properties.

All models involving the real world are *approximate*!
Hardware Verification
Hardware Verification

• Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.
- Works hierarchically from arithmetic units and memories right down to flip-flops and transistors.
Hardware Verification

- Pioneered by M. J. C. Gordon and his students, using successive versions of the HOL system.
- Used to model substantial hardware designs, including the ARM6 processor.
- Works hierarchically from arithmetic units and memories right down to flip-flops and transistors.
- Crucially uses higher-order logic, modelling signals as boolean-valued functions over time.
Devices as Relations

A relation in $a, b, c, d$

$g \rightarrow s = d$

The relation describes the possible combinations of values on the ports.

Values could be bits, words, signals (functions from time to bits), etc.
Relational Composition
Relational Composition

Consider the following two devices:

\[ D_1 \xrightarrow{a} x \]

\[ S_1[a, x] \]

\[ D_2 \xrightarrow{x} b \]

\[ S_2[x, b] \]

Logical conjunction (\( \land \)) models the effect of connecting components together:

\[ D_1 \land D_2 \]

\[ S_1[a, x] \land S_2[x, b] \]

two devices modelled by two formulas
Relational Composition

Consider the following two devices:

\[ D_1 \quad a \rightarrow x \quad S_1[a, x] \]
\[ \quad x \rightarrow \quad D_2 \quad b \quad S_2[x, b] \]

Logical conjunction (\( \land \)) models the effect of connecting components together:

\[ D_1 \land D_2 \quad a \land b \quad x \quad S_1[a, x] \land S_2[x, b] \]

two devices modelled by two formulas

the connected ports have the same value
Relational Composition

Consider the following two devices:

\[ D_1 \quad D_2 \]
\[ a \rightarrow x \quad x \rightarrow b \]
\[ S_1[a, x] \quad S_2[x, b] \]

Logical conjunction (\( \land \)) models the effect of connecting components together:

\[ D_1 \land D_2 \]
\[ a \land b \quad x \]
\[ S_1[a, x] \land S_2[x, b] \]

Modelling Hardware: TFM/MN/MJCG – p.10/3

the connected ports have the same value

Existential quantification (\( \exists \)) models the effect of making wires internal to the design:

\[ \exists x. \]
\[ D_1 \land D_2 \]
\[ a \land b \quad x \]
\[ \exists x. S_1[a, x] \land S_2[x, b] \]

the connected ports have some value
Specifications and Correctness
Specifications and Correctness

• The *implementation* of a device in terms of other devices can be expressed by composition.
Specifications and Correctness

• The *implementation* of a device in terms of other devices can be expressed by composition.

• The *specification* of the device’s intended behaviour can be given by an abstract formula.
Specifications and Correctness

• The *implementation* of a device in terms of other devices can be expressed by composition.

• The *specification* of the device’s intended behaviour can be given by an abstract formula.

• Sometimes the implementation and specification can be proved equivalent: $Imp \iff Spec$. 
Specifications and Correctness

- The *implementation* of a device in terms of other devices can be expressed by composition.
- The *specification* of the device’s intended behaviour can be given by an abstract formula.
- Sometimes the implementation and specification can be proved equivalent: $\text{Imp} \leftrightarrow \text{Spec}$.
- The property $\text{Imp} \Rightarrow \text{Spec}$ ensures that every possible behaviour of the $\text{Imp}$ is permitted by $\text{Spec}$. 
Specifications and Correctness

• The *implementation* of a device in terms of other devices can be expressed by composition.

• The *specification* of the device’s intended behaviour can be given by an abstract formula.

• Sometimes the implementation and specification can be proved equivalent: \( \text{Imp} \Leftrightarrow \text{Spec} \).

• The property \( \text{Imp} \Rightarrow \text{Spec} \) ensures that every possible behaviour of the \( \text{Imp} \) is permitted by \( \text{Spec} \).

*Impossible* implementations satisfy *all* specifications!
The Switch Model of CMOS

\[ \text{Ptran}(g, s, d) = (\neg g \Rightarrow (d = s)) \]

\[ \text{Ntran}(g, s, d) = (g \Rightarrow (d = s)) \]

\[ \text{Gnd} \, g = (g = \text{F}) \]

\[ \text{Pwr} \, p = (p = \text{T}) \]
The Switch Model of CMOS

Ptran\((g, s, d) = (\neg g \Rightarrow (d = s))\)

Ntran\((g, s, d) = (g \Rightarrow (d = s))\)

Gnd \(g = (g = F)\)

Pwr \(p = (p = T)\)

subsection{*[ Specification of CMOS primitives *]}

text{*[ P and N transistors *]}
definition "Ptran = (\lambda(g,a,b). (~g \rightarrow a = b))"
definition "Ntran = (\lambda(g,a,b). (g \rightarrow a = b))"

text{*[ Power and Ground*]}
definition "Pwr p = (p = True)"
definition "Gnd p = (p = False)"
Full Adder: Specification

• An $n$-bit ripple-carry adder:

\begin{align*}
\text{Add1} & \quad \text{Add1} \\
\text{cin} & \quad \text{cout} \\
\text{sum} & \quad \text{sum}
\end{align*}

\[2 \times \text{cout} + \text{sum} = a + b + \text{cin}\]
We wish to prove that:

\[ 2 \times cout + \text{sum} = a + b + cin \]

There are, as usual, three steps:

1. Define a model of the circuit in logic.
2. Formulate the correctness of the circuit.
3. Prove the correctness of the circuit.
Full Adder: Implementation
Full Adder in Isabelle

definition "Add1Imp = (λ(a,b,cin,sum,cout).
   ∃p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11.
   Ptran(p1,p0,p2) ∧ Ptran(cin,p0,p3) ∧
   Ptran(b,p2,p3) ∧ Ptran(a,p2,p4) ∧
   Ptran(p1,p3,p4) ∧ Ntran(a,p4,p5) ∧
   Ntran(p1,p4,p6) ∧ Ntran(b,p5,p6) ∧
   Ntran(p1,p5,p11) ∧ Ntran(cin,p6,p11) ∧
   Ptran(a,p0,p7) ∧ Ptran(b,p0,p7) ∧
   Ptran(a,p0,p8) ∧ Ptran(cin,p7,p1) ∧
   Ptran(b,p8,p1) ∧ Ntran(cin,p1,p9) ∧
   Ntran(b,p1,p10) ∧ Ntran(a,p9,p11) ∧
   Ntran(b,p9,p11) ∧ Ntran(a,p10,p11) ∧
   Pwr(p0) ∧ Ptran(p4,p0,sum) ∧
   Ntran(p4,sum,p11) ∧ Gnd(p11) ∧
   Ptran(p1,cout,p11) ∧ Ntran(p1,cout,p11))"

lemma Add1Correct:
"Add1Imp(a,b,cin,sum,cout) = Add1Spec(a,b,cin,sum,cout)"
by (simp add: Pwr_def Gnd_def Ntran_def Ptran_def Add1Spec_def
Add1Imp_def bit_val_def ex_bool_eq)
Full Adder in Isabelle

(∃b. P b) = (P True ∨ P False)
An $n$-bit Ripple-Carry Adder

\[(2^n \times cout) + s = a + b + cin\]
An \( n \)-bit Ripple-Carry Adder

\[
(2^n \times \text{cout}) + s = a + b + \text{cin}
\]

- Cascading several full adders yields an \( n \)-bit adder.
An $n$-bit Ripple-Carry Adder

\[ (2^n \times \text{cout}) + s = a + b + \text{cin} \]

- Cascading several full adders yields an $n$-bit adder.
- The implementation is expressed recursively.
An $n$-bit Ripple-Carry Adder

\[ (2^n \times cout) + s = a + b + cin \]

- Cascading several full adders yields an $n$-bit adder.
- The implementation is expressed recursively.
- The specification is obvious mathematics.
Adder Specification

\[(2^n \times cout) + s = a + b + cin\]
Adder Specification

\((2^n \times cout) + s = a + b + cin\)

values of n-bit words
Adder Specification

\[(2^n \times cout) + s = a + b + cin\]

values of n-bit words

```
(* Unsigned number denoted by bitstring f(n-1)...f(0) *)

fun bits_val where
  "bits_val f 0 = 0"
| "bits_val f (Suc n) = 2^n * bit_val(f n) + bits_val f n"

(* Specification of an n-bit adder *)

definition
  "AdderSpec n = (\(a, b, cin, sum, cout\).
    2^n * bit_val cout + bits_val sum n = bits_val a n + bits_val b n + bit_val cin)"
```
Adder Specification

\[(2^n \times cout) + s = a + b + cin\]

values of n-bit words

---

definition

"AdderSpec n = (\(a, b, cin, sum, cout\).
\[2^n \times \text{bit\_val\_cout} + \text{bit\_val\_sum} n = \text{bit\_val\_a} n + \text{bit\_val\_b} n + \text{bit\_val\_cin}\)"
Adder Specification

\[(2^n \times cout) + s = a + b + cin\]

Values of n-bit words

```plaintext
fun bits_val where
  "bits_val f 0     = 0"
| "bits_val f (Suc n) = 2^n * bit_val(f n) + bits_val f n"

definition
"AdderSpec n = (\(a, b, cin, sum, cout\)).
  2^n * bit_val cout + bits_val sum n =
  bits_val a n + bits_val b n + bit_val cin)"
```
Adder Specification

\[(2^n \times cout) + s = a + b + cin\]
Adder Implementation

Another Example

- An \( n \)-bit ripple-carry adder:

\[
\begin{align*}
&\text{Add1} & \text{Add1} & \text{Add1} & \text{Add1} \\
&a_{n-1} b_{n-1} & a_2 b_2 & a_1 b_1 & a_0 b_0 \\
&\text{cout} & \cdots & \text{cout} & \text{cin} \\
&s_{n-1} & s_2 & s_1 & s_0
\end{align*}
\]

- We wish to prove that:

\[
(2^n \times \text{cout}) + s = a + b + \text{cin}
\]

- There are, as usual, three steps:
  1. define a model of the circuit in logic
  2. formulate the correctness of the circuit
  3. prove the correctness of the circuit
Adder Implementation

\[ \begin{array}{cccccc}
  a_{n-1} & b_{n-1} & a_2 & b_2 & a_1 & b_1 \\
  s_{n-1} & \text{Add1} & s_2 & \text{Add1} & s_1 & \text{Add1} \\
  \text{cout} & \cdots & \text{Add1} & \text{Add1} & \text{Add1} & \text{cin} \\
\end{array} \]

- \text{We wish to prove that:}
  \[ (2^n \times \text{cout}) + s = a + b + \text{cin} \]
- \text{There are, as usual, three steps:}
  - define a model of the circuit in logic
  - formulate the correctness of the circuit
  - prove the correctness of the circuit

```text
(* Implementation of an n-bit ripple-carry adder*)

fun AdderImp where
  "AdderImp 0 (a, b, cin, sum, cout) = (cout = cin)"
| "AdderImp (Suc n) (a, b, cin, sum, cout) =
  (\exists c. AdderImp n (a, b, cin, sum, c) \land
  Add1Imp (a \ n, b \ n, c, sum \ n, cout))"
```
Adder Implementation

Another Example

• An $n$-bit ripple-carry adder:

```
Add1 Add1 Add1 Add1

\( a_{n-1} \ b_{n-1} \ \cdots \ a_2 \ b_2 \ a_1 \ b_1 \ a_0 \ b_0 \)
```

\( \text{cout} \quad \text{cin} \)

\( s_{n-1} \quad s_2 \quad s_1 \quad s_0 \)

\[ (2^n \times \text{cout}) + s_2 = a_0 + b_0 + \text{cin} \]

• We wish to prove that:

• There are, as usual, three steps:
  - define a model of the circuit in logic
  - formulate the correctness of the circuit
  - prove the correctness of the circuit

```
fun AdderImp where
  "AdderImp 0 (a, b, cin, sum, cout) = (cout = cin)"

  "AdderImp (Suc n) (a, b, cin, sum, cout) =
    (\exists c. AdderImp n (a, b, cin, sum, c) \land
     Add1Imp (a n, b n, c, sum n, cout))"
```

\[ \text{internal wire, to be hidden} \]
Adder Implementation

Another Example

- An \( n \)-bit ripple-carry adder:

\[
\begin{align*}
a_{n-1} & \quad b_{n-1} \\
\vdots & \quad \vdots \\
a_2 & \quad b_2 \\
a_1 & \quad b_1 \\
a_0 & \quad b_0 \\
\end{align*}
\]

- We wish to prove that:

\[
2^n \times \text{cout} + s_n = a + b + \text{cin}
\]

- There are, as usual, three steps:
  1. Define a model of the circuit in logic
  2. Formulate the correctness of the circuit
  3. Prove the correctness of the circuit

Modelling Hardware: TFM/MN/MJCG

```
text{ /* Implementation of an n-bit ripple-carry adder */}

fun AdderImp where
  "AdderImp 0 (a, b, cin, sum, cout) = (cout = cin)"

| "AdderImp (Suc n) (a, b, cin, sum, cout) = |
| (\exists c. AdderImp n (a, b, cin, sum, c) \land |
| Add1Imp (a n, b n, c, sum n, cout))"
```
Adder Implementation

An $n$-bit ripple-carry adder:

$\begin{array}{c}
\text{Add}1 \\
\vdots \\
\text{Add}1 \\
\text{Add}1
\end{array}$

We wish to prove that:

$2^n \times cout + s_n = a + b + cin$

There are, as usual, three steps:

1. Define a model of the circuit in logic
2. Formulate the correctness of the circuit
3. Prove the correctness of the circuit

A zero-bit adder simply connects the carry lines!
Partial Correctness Proof

lemma AdderCorrect: "AdderImp n (a, b, cin, sum, cout) \Rightarrow AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
    then obtain c
      where AddS: "AdderSpec n (a, b, cin, sum, c)"
      and Add1: "AddImp (a n, b n, c, sum n, cout)"
      by (auto intro: Suc)

this:
  AdderImp n (a, b, cin, sum, ?cout) \Rightarrow AdderSpec n (a, b, cin, sum, ?cout)
  AddImp (Suc n) (a, b, cin, sum, cout)

goal (1 subgoal):
  1. \A n cout.
     [[\cout.
        AdderImp n (a, b, cin, sum, cout) \Rightarrow
        AdderSpec n (a, b, cin, sum, cout);
        AddImp (Suc n) (a, b, cin, sum, cout)]
     \Rightarrow AdderSpec (Suc n) (a, b, cin, sum, cout)]
Partial Correctness Proof

```
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout) ⇒ AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
    then obtain c
      where AddS: "AdderSpec n (a, b, cin, sum, c)"
      and Add1: "AddImp (n a b n, c, sum n, cout)"
      by (auto intro: Suc)
this:
  AdderImp n (a, b, cin, sum, ?cout) ⇒ AdderSpec n (a, b, cin, sum, ?cout)
  AdderImp (Suc n) (a, b, cin, sum, cout)

goal (1 subgoal):
  1. ∀n cout.
     (∀cout.
       AdderImp n (a, b, cin, sum, cout) ⇒
        AdderSpec n (a, b, cin, sum, cout);
        AdderImp (Suc n) (a, b, cin, sum, cout)⇒
        AdderSpec (Suc n) (a, b, cin, sum, cout)
)
```

assumptions
Partial Correctness Proof

```
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout) \implies AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
  where AddS: "AdderSpec n (a, b, cin, sum, c)"
  and Add1: "AddImp (a n b n c, sum n cout)"
  by (auto intro: Suc)
```

**Assumptions**

```
AdderImp n (a, b, cin, sum, ?cout) \implies AdderSpec n (a, b, cin, sum, ?cout)
AdderImp (Suc n) (a, b, cin, sum, cout)
```

**Conclusion**

```
\forall n cout. 
AdderImp n (a, b, cin, sum, cout) \implies 
AdderSpec n (a, b, cin, sum, cout); 
AdderImp (Suc n) (a, b, cin, sum, cout) \implies 
AdderSpec (Suc n) (a, b, cin, sum, cout)
```
Using the Induction Hypothesis

```plaintext
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout) ==> AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
    where AddS: "AdderSpec n (a, b, cin, sum, c)"
    and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
  by (auto intro: Suc)
have (∀c. [AdderSpec n (a, b, cin, sum, c);
  Add1Imp (a n, b n, c, sum n, cout)]
```
Using the Induction Hypothesis

```plaintext
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout) ⇒ AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
  by (simp add: AdderSpec_def)
next
case (Suc n)
  then obtain c
  where AddS: "AdderSpec n (a, b, cin, sum, c)"
  and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
  by (auto intro: Suc)

have (∀c. [AdderSpec n (a, b, cin, sum, c);
  Add1Imp (a n, b n, c, sum n, cout)]
  ⇒ ?thesis) ⇒
  ?thesis
```
Using the Induction Hypothesis

lemma AdderCorrect:
   "adderImp n (a, b, cin, sum, cout) ⇒ AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
   case 0 thus ?case
   by (simp add: AdderSpec_def)
next
   case (Suc n)
   then obtain c
   where AddS: "adderSpec n (a, b, cin, sum, c)"
   and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
   by (auto intro: Suc)
have (∀c. [adderSpec n (a, b, cin, sum, c);
       Add1Imp (a n, b n, c, sum n, cout)]
     ⇒ ?thesis) ⇒
  ?thesis
Using the Induction Hypothesis

lemma AdderCorrect:
"AdderImp n (a, b, cin, sum, cout) \implies AdderSpec n (a, b, cin, sum, cout)"

proof (induct n arbitrary: cout)
  case 0 thus ?case
  by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
  where AddS: "AdderSpec n (a, b, cin, sum, c)"
  and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
  by (auto intro: Suc)

have (\forall c. [AdderSpec n (a, b, cin, sum, c);
  Add1Imp (a n, b n, c, sum n, cout)]
  \implies \text{thesis}) \implies ?thesis) \implies
A Tiresome Calculation

```
where  AddS: "AdderSpec n (a, b, cin, sum, c)"
    and  Add1: "Add1Imp (a n, b n, c, sum n, cout)"
    by (auto intro: Suc)
    have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
        (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"
    by (simp add: algebra_simps)
    also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
        (2 ^ n)"
    using Add1 by (simp add: Add1Correct Add1Spec_def)
    finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" using AddS
        by (simp add: AdderSpec_def algebra_simps)

calculation:
    bit_val (sum n) * 2 ^ n + bit_val cout * (2 * 2 ^ n) =
    (bit_val c + (bit_val (a n) + bit_val (b n))) * 2 ^ n
```
A Tiresome Calculation

rearranging the terms

```plaintext
where AddS: "AdderSpec n (a, b, cin, sum, c)"
and Add1: "Add1Imp (a n, b n, c, sum n, cout)"

by (auto intro: Suc)

have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
    (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"

by (simp add: algebra_simps)

also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) *
    (2 ^ n)"

using Add1 by (simp add: Add1Correct Add1Spec_def)

finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" using AddS
by (simp add: AdderSpec_def algebra_simps)
```

bit_val (sum n) * 2 ^ n + bit_val cout * (2 * 2 ^ n) =
(bit_val c + (bit_val (a n) + bit_val (b n))) * 2 ^ n
A Tiresome Calculation

rearranging the terms

replacing outputs by inputs
The Finished Proof

```
lemma AdderCorrect:
  "AdderImp n (a, b, cin, sum, cout) ⇒ AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case by (simp add: AdderSpec_def)
next
  case (Suc n)
    then obtain c
    where AddS:  "AdderSpec n (a, b, cin, sum, c)"
    and Add1:  "Add1Imp (a n, b n, c, sum n, cout)"
    by (auto intro: Suc)
    have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
        (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"
        by (simp add: algebra_simps)
    also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) * 
        (2 ^ n)"
      using Add1 by (simp add: Add1Correct Add1Spec_def)
  finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" using AddS
    by (simp add: AdderSpec_def algebra_simps)
qed
```
The Finished Proof

```
lemma AdderCorrect:
"AdderImp n (a, b, cin, sum, cout) ⇒ AdderSpec n (a, b, cin, sum, cout)"
proof (induct n arbitrary: cout)
  case 0 thus ?case
    by (simp add: AdderSpec_def)
next
  case (Suc n)
  then obtain c
  where AddS: "AdderSpec n (a, b, cin, sum, c)"
     and Add1: "Add1Imp (a n, b n, c, sum n, cout)"
  by (auto intro: Suc)
  have "bit_val (sum n) * (2 ^ n) + bit_val cout * (2 * 2 ^ n) =
       (bit_val (sum n) + (bit_val cout * 2)) * (2 ^ n)"
    by (simp add: algebra_simps)
  also have "... = (bit_val c + (bit_val (a n) + bit_val (b n))) * (2 ^ n)"
    using Add1 by (simp add: Add1Correct Add1Spec_def)
  finally show "AdderSpec (Suc n) (a, b, cin, sum, cout)" using AddS
    by (simp add: AdderSpec_def algebra_simps)
qed
```
Proving Equivalence

```isar
lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, sum, cout) =
   (\exists c. AdderSpec n (a, b, cin, sum, c) & Add1Spec (a n, b n, c, sum n, cout))"
apply (auto simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)
```

```
goal (16 subgoals):
1. [a n; b n; sum n; ¬ cout; cin;
   bits_val sum n = Suc (2 ^ n + (bits_val a n + bits_val b n))]
   ⇒ False
2. [a n; b n; sum n; ¬ cout; ¬ cin;
   bits_val sum n = 2 ^ n + (bits_val a n + bits_val b n)]
   ⇒ False
3. [a n; b n; ¬ sum n; ¬ cout; cin;
   bits_val sum n = Suc (2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
   ⇒ False
4. [a n; b n; ¬ sum n; ¬ cout; ¬ cin;
   bits_val sum n = 2 ^ n + bits_val a n + (2 ^ n + bits_val b n)]
   ⇒ False
5. [a n; ¬ b n; sum n; cout; cin;
   2 * 2 ^ n + bits_val sum n = Suc (bits_val a n + bits_val b n)]
   ⇒ False
6. [a n; ¬ b n; sum n; cout; ¬ cin;
   2 * 2 ^ n + bits_val sum n = bits_val a n + bits_val b n]
```

-u:-**-  Adder.thy    82% L130    (Isar Utoks Abbrev; Scripting )
-u:-%%-  *goals*    2% L4    (Isar Proofstate Utoks Abbrev; )
Proving Equivalence

```lemma AdderSpec_Suc:
    "AdderSpec (Suc n) (a, b, cin, sum, cout) = 
    (∃c. AdderSpec n (a, b, cin, sum, c) & Add1Spec (a n, b n, c, sum n, cout))"
apply (auto simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)

-u:-**- Adder.thy  82% 1130  (Isar Utoks Abbrev; Scripting )

goal (16 subgoals):
1. [a n; b n; sum n; ¬ cout; cin;
   bits_val sum n = Suc (2 ^ n + (bits_val a n + bits_val b n))]
   ⇒ False
2. [a n; b n; sum n; ¬ cout; ¬ cin;
   bits_val sum n = 2 ^ n + (bits_val a n + bits_val b n)]
   ⇒ False
3. [a n; b n; ¬ sum n; ¬ cout; cin;
   bits_val sum n = Suc (2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
   ⇒ False
4. [a n; b n; ¬ sum n; ¬ cout; ¬ cin;
   bits_val sum n = 2 ^ n + bits_val a n + (2 ^ n + bits_val b n)]
   ⇒ False
5. [a n; ¬ b n; sum n; cout; cin;
   2 * 2 ^ n + bits_val sum n = Suc (bits_val a n + bits_val b n)]
   ⇒ False
6. [a n; ¬ b n; sum n; cout; ¬ cin;
   2 * 2 ^ n + bits_val sum n = bits_val a n + bits_val b n]
```

just need to prove this...
Proving Equivalence

```isar
lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, sum, cout) =
   (∃c. AdderSpec n (a, b, cin, sum, c) & Add1Spec (a n, b n, c, sum n, cout))"
apply (auto simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)

-u-:**- Adder.thy 82% L130 (Isar Utoks Abbrev; Scripting )
```

```
goal (16 subgoals):
1. [a n; b n; sum n; ~ cout; cin;
   bits_val sum n = Suc (2 ^ n + (bits_val a n + bits_val b n))]
   ==> False
2. [a n; b n; sum n; ~ cout; ~ cin;
   bits_val sum n = 2 ^ n + (bits_val a n + bits_val b n)]
   ==> False
3. [a n; b n; ~ sum n; ~ cout; cin;
   bits_val sum n = Suc (2 ^ n + bits_val a n + (2 ^ n + bits_val b n))]
   ==> False
4. [a n; b n; ~ sum n; ~ cout; ~ cin;
   bits_val sum n = 2 ^ n + bits_val a n + (2 ^ n + bits_val b n)]
   ==> False
5. [a n; ~ b n; sum n; cout; cin;
   2 * 2 ^ n + bits_val sum n = Suc (bits_val a n + bits_val b n)]
   ==> False
6. [a n; ~ b n; sum n; cout; ~ cin;
   2 * 2 ^ n + bits_val sum n = bits_val a n + bits_val b n]
-u-:%- *goals* 2% L4 (Isar Proofstate Utoks Abbrev;)---
```
A Crucial Lemma

```
lemma bits_val_less: "bits_val \( f \) \( n < 2^\( n \)\)"
  by (induct \( n \), auto simp add: bit_val_def)

lemma AdderSpec_Suc:
  "AdderSpec (Suc \( n \)) (a, b, cin, sum, cout) =
  (\( \exists c \). AdderSpec n (a, b, cin, sum, c) \& Add1Spec (a, n, b n, c, sum n, cout))"
using bits_val_less [of a n] bits_val_less [of b n] bits_val_less [of sum n]
  by (simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)
```

- proof (prove): step 1

using this:
  bits_val \( a \) \( n < 2^\( n \)\)
  bits_val \( b \) \( n < 2^\( n \)\)
  bits_val \( \text{sum} \) \( n < 2^\( n \)\)

goal (1 subgoal):
  1. AdderSpec (Suc \( n \)) (a, b, cin, sum, cout) =
     (\( \exists c \). AdderSpec n (a, b, cin, sum, c) \&
     Add1Spec (a, n, b n, c, sum n, cout))
```
A Crucial Lemma

lemma bits_val_less: "bits_val f n < 2^n"
by (induct n, auto simp add: bit_val_def)

lemma AdderSpec_Suc:
  "AdderSpec (Suc n) (a, b, cin, sum, cout) =
   (\exists c. AdderSpec n (a, b, cin, sum, c) \& Add1Spec (a n, b n, c, sum n, cout))"
using bits_val_less [of a n] bits_val_less [of b n] bits_val_less [of sum n]
by (simp add: AdderSpec_def Add1Spec_def ex_bool_eq bit_val_def)

proof (prove): step 1

using this:
  bits_val a n < 2 ^ n
  bits_val b n < 2 ^ n
  bits_val sum n < 2 ^ n

goal (1 subgoal):
  1. AdderSpec (Suc n) (a, b, cin, sum, cout) =
     (\exists c. AdderSpec n (a, b, cin, sum, c) \&
       Add1Spec (a n, b n, c, sum n, cout))
A Crucial Lemma

a trivial upper bound on the value of a bit string

inserting three instances of that fact
A Crucial Lemma

A trivial upper bound on the value of a bit string.

Inserting three instances of that fact.

Now proof is trivial, by arithmetic.
The Opposite Implication

```
lemma AdderCorrect2:
  "AdderSpec n (a, b, cin, sum, cout) \implies AdderImp n (a, b, cin, sum, cout)"
apply (induct n arbitrary: cout)
apply (simp add: AdderSpec_def)
apply (auto simp add: AdderSpec_Suc Add1Correct)
done
```
The Opposite Implication

The implementation and specification are equivalent!
Making Instances of Theorems
Making Instances of Theorems

- $thm \ [\text{of } a \ b \ c]$ replaces variables by terms from left to right
Making Instances of Theorems

• \textit{thm [of \ a \ b \ c]}  
  replaces variables by terms from left to right

• \textit{thm [where \ x=a]}  
  replaces the variable \( x \) by the term \( a \)
Making Instances of Theorems

• $\text{thm [ of } a \ b \ c \text{]}$
  replaces variables by terms from left to right

• $\text{thm [ where } x=a\text{]$
  replaces the variable $x$ by the term $a$

• $\text{thm [ OF } \text{thm}_1 \ \text{thm}_2 \ \text{thm}_3 \text{]}$
  discharges premises from left to right
Making Instances of Theorems

- \texttt{thm \[ of \ a \ b \ c \]}
  replaces variables by terms from left to right

- \texttt{thm \[ where \ x=a \]}
  replaces the variable \( x \) by the term \( a \)

- \texttt{thm \[ OF \ thm_1 \ thm_2 \ thm_3 \]}
  discharges premises from left to right

- \texttt{thm \[ simplified \]}
  applies the simplifier to \textit{thm}
Making Instances of Theorems

- `thm [of a b c]` replaces variables by terms from left to right
- `thm [where x=a]` replaces the variable `x` by the term `a`
- `thm [OF thm1 thm2 thm3]` discharges premises from left to right
- `thm [simplified]` applies the simplifier to `thm`
- `thm [attr1, attr2, attr3]` applying multiple attributes
The End

You know my methods. Apply them!

Sherlock Holmes