**Limitations of the Method**

- Formal proof can’t guarantee actual chips will work:
  - design models are not always accurate
  - there may be fabrication defects

- Specifications may not capture requirements:
  - large specifications may be unreadable
  - some input conditions may be ignored

**Why Formal Specification?**

Consider this device (J. Herbert’s example):

\[
\begin{array}{c}
\text{datain} \\
\text{sample} \\
\end{array} \quad \rightarrow \quad \begin{array}{c}
\text{dataout} \\
\end{array} \quad \begin{array}{c}
\text{out} \\
\end{array}
\]

This can be specified *informally* by

The input line `datain` accepts a stream of bits, and the output line `dataout` emits the same stream delayed by four cycles. The bus `out` is four bits wide. If the input `sample` is false then the 4-bit word at `out` is the last four bits input at `datain`. Otherwise, the output word is all zeros.

**Modelling Hardware in Higher Order Logic**

Original slides by Tom Melham and Michael Norrish
(edited by Mike Gordon)

**Hardware Verification Method**

- Classical method of hardware verification:
  1. write a specification of intended behaviour
     \[ \text{Spec} \]
  2. write specifications of the design components
     \[ \text{Part-1}, \ldots, \text{Part-n} \]
  3. define a formal model of the design
     \[ \vdash \text{Design} = \text{Part-1} + \cdots + \text{Part-n} \]
  4. formulate and prove correctness
     \[ \vdash \text{Design satisfies Spec} \]

- This general verification approach
  - underlies various specific formal methods
  - requires mechanized support for large designs
  - is usually applied hierarchically
**Why Formal Specification?**

The informal specification is

- vague: does ‘the last four bits input’ include the current bit?
- incomplete: what is the value at dataout during the first three cycles?
- unusable: a natural language specification can’t be simulated or compiled!

**Formal Specification in HOL**

- Consider the following device:

  ![Device Diagram]

  This is specified by a boolean term \( S[a, b, c, d] \) with free variables \( a, b, c, \) and \( d \).

  - The idea is that
    - \( a, b, c, d \) model externally-observable values
    - \( S[a, b, c, d] = \begin{cases} T & \text{if } a, b, c, \text{ and } d \text{ could occur simultaneously on the } \\
    \text{corresponding external wires of the} & \text{device Dev} \\
    F & \text{otherwise} \end{cases} \)

**Specification Examples**

- Simple combinational behaviour:

  ![Combinational Circuit]

  \[ \vdash \text{Xor}(i_1, i_2, o) = (o = \neg(i_1 = i_2)) \]

- Bidirectional wires:

  ![Bidirectional Wires Diagram]

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

- Sequential (time-dependent) behaviour:

  ![Sequential Circuit Diagram]

  \[ \vdash \text{Dtype}(ck, d, q) = \forall t. q(t+1) = \begin{cases} \text{if } \text{Rise } ck \ t \ \text{then } d \ t \ \text{else } q \ t \end{cases} \]

  \[ \vdash \text{Rise } ck \ t = \neg(ck(t) \wedge ck(t+1)) \]
**Specifying the Sampler**

- We can specify the sampler formally by

\[
\forall t: \text{time}.
\]

\[
(dataout(t) = datain(t-4)) \land
\]

\[
(out(t) = \begin{cases} 
[F; F; F; F] & \text{if } sample(t) \\
[datain(t-4); datain(t-3); datain(t-2); datain(t-1)] & \text{else}
\end{cases}
\]

**Composing Behaviours**

- Consider the following two devices:

```
\[\begin{array}{cc}
 a & D_1 & x & D_2 & b \\
 S_1[a, x] & S_2[x, b]
\end{array}\]
```

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

```
\[\begin{array}{cc}
 a & D_1 & x & D_2 & b \\
 S_1[a, x] \land S_2[x, b]
\end{array}\]
```

**Hiding Internal Structure**

- Consider the composite device

```
\[\begin{array}{cc}
 a & D_1 & x & D_2 & b \\
 S_1[a, x] \land S_2[x, b]
\end{array}\]
```

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

```
\[\begin{array}{cc}
 a & D_1 & x & D_2 & b \\
 \exists x. S_1[a, x] \land S_2[x, b]
\end{array}\]
```

- Existential quantification is called a **hiding** operator—it ‘hides’ internal wires.
Hierarchical Verification

The hierarchical verification method:

Level 0

Model:
\[ \vdash M = \exists z. S_1 \land S_2 \]
Correctness:
\[ \vdash M \text{ sat } S \]

Level 1

Models:
\[ \vdash M_1 = \exists x. P_1 \land P_2 \]
\[ \vdash M_2 = \exists y. P_3 \land P_4 \]
Correctness:
\[ \vdash M_1 \text{ sat } S_1 \]
\[ \vdash M_2 \text{ sat } S_2 \]

Shallow embedding of Verilog

Some typical structural Verilog

```
module COMP (p1, ... , pm);
wire w1, ..., wn;
COMP1 M1 (...);
COMP2 M2 (...);
endmodule
```

Assume formulas for COMP1, COMP2 already defined

Logical representation:

\[ \text{COMP}(p1, ..., pm) = \exists w1 \ldots wn. \text{COMP1}(\ldots) \land \text{COMP2}(\ldots) \]

Formulating Correctness

A key part of formal hardware verification is formalizing what ‘correctness’ means.

The strongest formulation is equivalence:

\[ \vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] = S[v_1, \ldots, v_n] \]

For partial specifications, use implication:

\[ \vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n] \]

In general, the satisfaction relationship

\[ \vdash M[v_1, \ldots, v_n] \text{ sat } S[\text{abs}(v_1), \ldots, \text{abs}(v_n)] \]

must be one of abstraction. The specification will be an abstraction of the design model. Various kinds of abstractions on signals \( \text{abs} \) will be discussed later.

Hierarchical Design—Advantages

- Each type of module verified only once
  - the statement of its correctness will be reused many times
- Controls complexity through abstraction
  - each verification is done at the appropriate level of complexity
Design Model and Correctness

- We define the design model using composition and hiding, as follows:

\[ \vdash \text{Inv}(i, o) = \exists g \ p. \ \text{Pwr}(p) \land \text{Gnd}(g) \land \text{Ntran}(i, g, o) \land \text{Ptran}(i, p, o) \]

- Correctness is formulated by the equivalence:

\[ \vdash \forall i \ o. \ \text{Inv}(i, o) = (o = \neg i) \]

This follows by purely logical inference...

A Simple Correctness Proof

- Here is the design of a CMOS inverter:

\[ \begin{align*}
\text{Ntran}(i, g, o) & = (g \Rightarrow (o = s)) \\
\text{Ptran}(i, p, o) & = (i \Rightarrow (o = p)) \\
\end{align*} \]

- Suppose we wish to verify that \( o = \neg i \).

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

CMOS Primitives

- Formal specifications of primitives:

\[ \begin{align*}
\text{Ntran}(g, s, d) & = (g \Rightarrow (d = s)) \\
\text{Ptran}(g, s, d) & = (\neg g \Rightarrow (d = s)) \\
\text{Gnd}(g) & = (g = F) \\
\text{Pwr}(p) & = (p = T) \\
\end{align*} \]

- This is the so-called switch model of CMOS.

The Correctness Proof

- Definition of Inv:

\[ \vdash \text{Inv}(i, o) = \exists g \ p. \ \text{Pwr}(p) \land \text{Gnd}(g) \land \text{Ntran}(i, g, o) \land \text{Ptran}(i, p, o) \]

- Expanding with definitions:

\[ \vdash \text{Inv}(i, o) = \exists g \ p. \ (p = T) \land (g = F) \land (i \Rightarrow (o = g)) \land (\neg i \Rightarrow (o = p)) \]

- By simple logical reasoning:

\[ \vdash \text{Inv}(i, o) = (i \Rightarrow (o = F)) \land (\neg i \Rightarrow (o = T)) \]
**The Correctness Proof continued**

- Simplifying gives:
  \[ \vdash \text{Inv}(i, o) = (i \Rightarrow \neg o) \land (\neg i \Rightarrow o) \]
- By the law of the contrapositive:
  \[ \vdash \text{Inv}(i, o) = (o \Rightarrow \neg i) \land (\neg i \Rightarrow o) \]
- By the definition of boolean equality:
  \[ \vdash \text{Inv}(i, o) = (o = \neg i) \]
- Generalizing the free variables gives:
  \[ \vdash \forall i. \text{Inv}(i, o) = (o = \neg i) \]

**Another Example**

- An \((n+1)\)-bit ripple-carry adder:

  \[
  \begin{array}{c}
  \text{cout} \\
  \downarrow \\
  a_n b_n \\
  \vdots \\
  a_2 b_2 \\
  \downarrow \\
  a_1 b_1 \\
  \downarrow \\
  a_0 b_0 \\
  \downarrow \\
  s_n \\
  \downarrow \\
  s_2 \\
  \downarrow \\
  s_1 \\
  \downarrow \\
  s_0 \\
  \text{cin}
  \end{array}
  \]

  We wish to prove that:
  \[ (2^{n+1} \times \text{cout}) + s = a + b + \text{cin} \]

- 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

**Defining the Model: types**

- Specification uses numbers, i.e. values of type `num`
- Implementation uses words – values of type `word`
  - \(n^{th}\) bit of \(w\) denoted by \(w \llbracket n \rrbracket\)
  - \(w \llbracket m : n \rrbracket\) denotes bits \(m\) to \(n\) of \(w\)
  - \(\text{Bv}(b)\) is the number represented by bit \(b\)
  - \(\text{V}(w)\) is the natural number represented by word \(w\)
- Abstraction from words to numbers (data abstraction):
  \[ \vdash \text{Bv} b \quad \text{if } b \text{ then } 1 \text{ else } 0 \]
  \[ \vdash \text{V} w[0 : 0] = \text{Bv} w[0] \]
  \[ \vdash \text{V} w[n+1 : 0] = 2^{n+1}(\text{Bv} w[n+1]) + \text{V} w[n : 0] \]

**Scope of the Method**

- The inverter example is, of course, trivial!
- But the same method has been applied to
  - a commercial CMOS cell library
  - several complete microprocessors (e.g. ARM)
  - floating point algorithms and hardware
- Features of the approach:
  - the specification language is just logic
    - *logic can mimic HDL constructs*
  - the rules of reasoning are also pure logic
    - *special-purpose derived rules are possible*
  - big formal proofs require machine assistance
Defining the Model: recursive definition

- If \( n > 0 \) an \((n+1)\)-bit adder is built from an \(n\)-bit adder

```
AdderImp(n-1)(a[n-1:0], b[n-1:0], cin, s[n-1:0], cout)
```

Defining the Model

- Recursive view of an \(n+1\)-bit adder:

```
Add1(a[n], b[n], cin, s[n], cout)
```

Formulation of Correctness

- Logical formulation of correctness:

```
Spec(n)(a, b, cin, s, cout) = ((2^n+1 coup) + s = a + b + cin)
```

```
∀n a b cin s coup. AdderImp(n)(a, b, cin, s, coup) ⇒ Spec(n)(V a[n:0], V b[n:0], Bv cin, V s[n:0], Bv coup)
```

- Note the data abstraction (abs in an earlier slide)
- This is easy to prove (done later in the course)
**Temporal Abstraction**

- Example—abstracting to unit delay:

\[
\Delta i o \vdash \Delta i o (i, o) = \forall t. o(t+1) = i t
\]

- Notions of time involved:
  - coarse grain of time—unit time = 1 clock cycle
  - fine grain of time—unit time \(\approx\) 1 gate delay

**Formulating Correctness**

- Then correctness is stated by:

\[
\Delta \forall ck. \text{Inf}(\text{Rise } ck) \Rightarrow
\forall d q. \text{Dtype}(ck, d, q) \Rightarrow
\Delta \Delta \text{Del}(d \text{ when } (\text{Rise } ck), q \text{ when } (\text{Rise } ck))
\]

- Note the formal validity condition:

\[
\Delta \text{Inf } P = \forall t. \exists t'. t' > t \land P t'
\]

**Industry use of theorem proving**

- Intel
  - floating point algorithms (uses HOL Light system)
  - hardware (uses internal tools Forte/reFLect)
- AMD
  - floating point (uses ACL2 prover)
- Sun
  - high level architecture verification (PVS)
- Rockwell Collins
  - low level code verification (ACL2)

- Use of model checking widespread
- discussed in latter part of the course

- Define the temporal abstraction functions:

\[
\Delta \text{Timeof } P \equiv \text{the time on } t_c \text{ such that } P \text{ true for } n\text{th time}
\]

\[
\Delta \text{signal when } P = \text{signal } \circ (\text{Timeof } P)
\]

where \((f \circ g)x = f(gx)\) [\(\circ\) is function composition]
Summary

- Specifying behaviour:
  - predicates—\(S[a, b, c, d]\)

- Specifying structure:
  - composition—\(S_1[a, x] \land S_2[x, b]\)
  - hiding—\(\exists x. S_1[a, x] \land S_2[x, b]\)

- Formulating correctness:
  - \(\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] = S[v_1, \ldots, v_n]\)
  - \(\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[v_1, \ldots, v_n]\)
  - \(\vdash \forall v_1 \ldots v_n. M[v_1, \ldots, v_n] \Rightarrow S[abs v_1, \ldots, abs v_n]\)

- Abstraction
  - data: \(w \mapsto V(w)\)
  - temporal: \(\text{sig} \mapsto \text{sig when (Rise clk)}\)