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

     Spec
  2. write specifications of the design components

     Part-$1, \ldots$ Part-n
  3. define a formal model of the design

     $\vdash$ Design $=$ Part-1 $+ \cdots +$ Part-n
  4. formulate and prove correctness

     $\vdash$ Design satisfies Spec
- This general verification approach
  - underlies various specific formal methods
  - requires mechanized support for large designs
  - is usually applied hierarchically

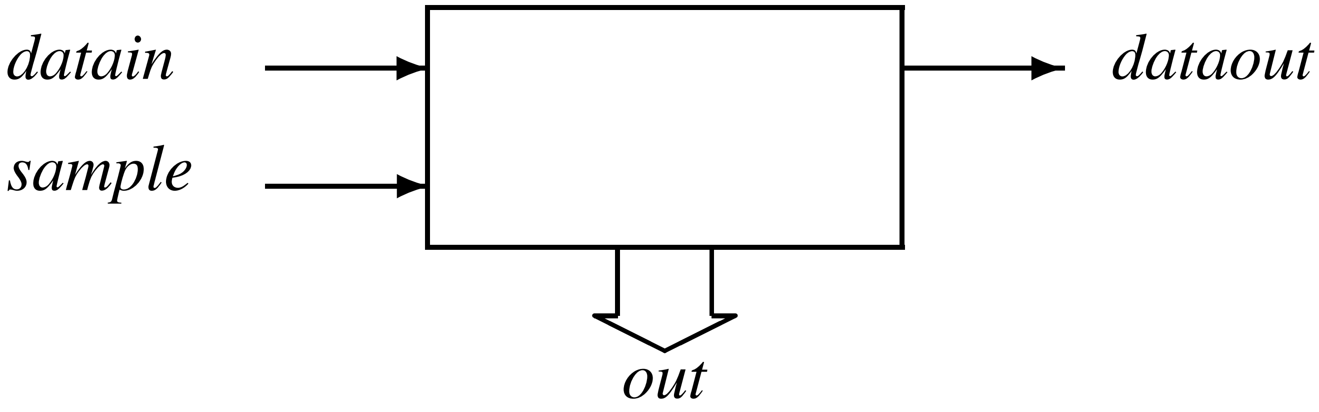
Limitations of the Method

- 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):



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.

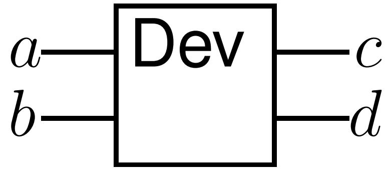
Why Formal Specification?

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:

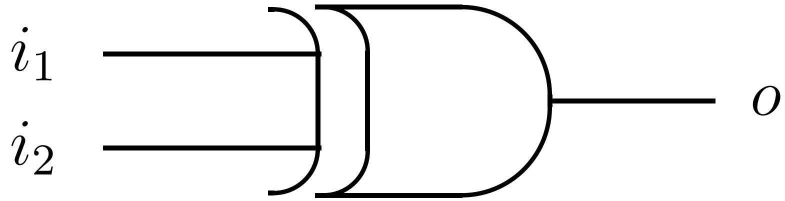


This is specified by a boolean term $S[a, b, c, d]$ with free variables a, b, c, and d.

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

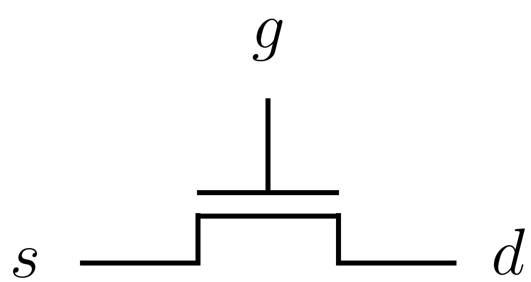
Specification Examples

- Simple combinational behaviour:



$$\vdash \mathsf{Xor}(i_1, i_2, o) = (o = \neg(i_1 = i_2))$$

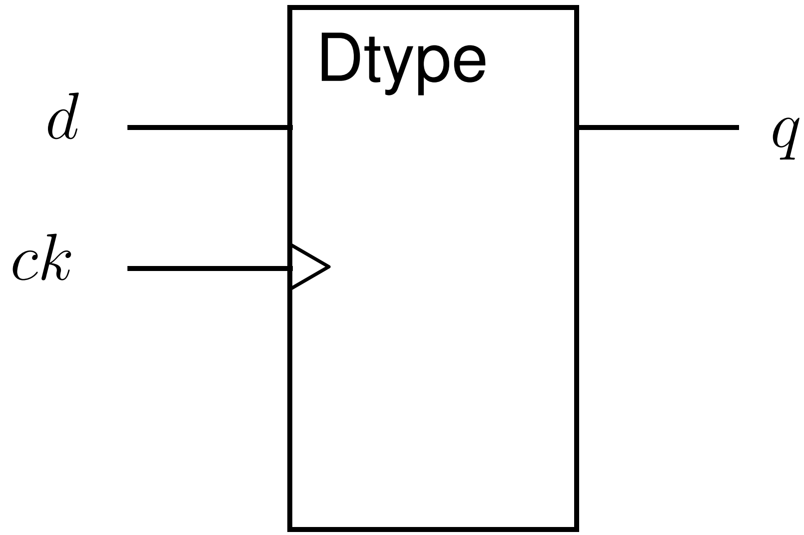
- Bidirectional wires:



$$\vdash \mathsf{Ntran}(g, s, d) = (g \Rightarrow (d = s))$$

Specification Examples

Sequential (time-dependent) behaviour:



$\vdash \mathsf{Dtype}(ck, d, q) = \forall t.\, q(t{+}1) = (\mathsf{if\ Rise}\ ck\ t\ \mathsf{then}\ d\ t\ \mathsf{else}\ q\ t)$

$\vdash \mathsf{Rise}\ ck\ t = \neg ck(t) \wedge ck(t{+}1)$

Specification of the Sampler

- We can specify the sampler formally by

$$
\begin{array}{l}
\forall t{:}\textit{time}. \\
\quad (\textit{dataout}(t) = \textit{datain}(t{-}4)) \\
\quad \wedge \\
\quad (\textit{out}(t) = \ \ \textsf{if}\ \textit{sample}(t) \\
\qquad\qquad\qquad\quad \textsf{then}\ [\textsf{F};\textsf{F};\textsf{F};\textsf{F}] \\
\qquad\qquad\qquad\quad \textsf{else}\ \ [\textit{datain}(t{-}4);\ \textit{datain}(t{-}3); \\
\qquad\qquad\qquad\qquad\quad\ \ \textit{datain}(t{-}2);\ \textit{datain}(t{-}1)\ ])
\end{array}
$$

Specification of the Sampler

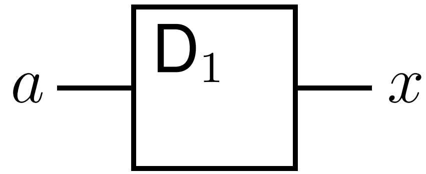
- We can specify the sampler formally by

$$
\begin{array}{l}
\forall t{:}\mathit{time}. \\
\quad (\mathit{dataout}(t) = \mathit{datain}(t{-}4)) \\
\quad \wedge \\
\quad (\mathit{out}(t) = \ \text{if}\ \mathit{sample}(t) \\
\qquad\qquad\qquad \text{then}\ [\mathsf{F}; \mathsf{F}; \mathsf{F}; \mathsf{F}] \\
\qquad\qquad\qquad \text{else}\ \ [\mathit{datain}(t{-}4);\ \mathit{datain}(t{-}3); \\
\qquad\qquad\qquad\qquad\ \ \mathit{datain}(t{-}2);\ \mathit{datain}(t{-}1)\ ])
\end{array}
$$

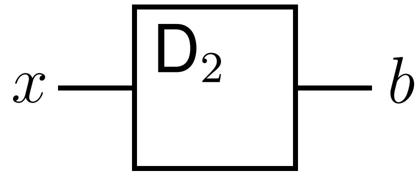
- The formal specification is
  - precise: 'last four bits input' doesn't include current bit
  - complete: can infer that *dataout* equals $\mathit{datain}(0)$ during the first three cycles.
  - usable: logic notation can be processed by machine

Composing Behaviours

- Consider the following two devices:

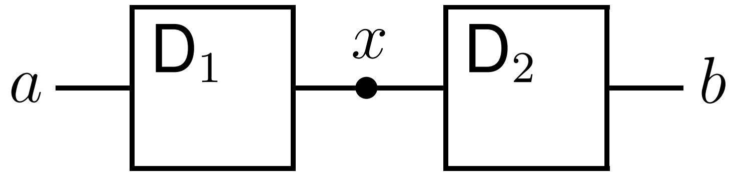


$\mathsf{S}_1[a, x]$



$\mathsf{S}_2[x, b]$

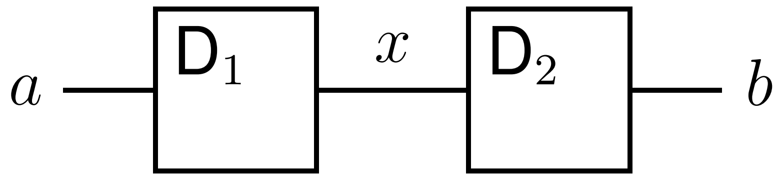
- Logical conjunction ($\wedge$) models the effect of connecting components together:



$\mathsf{S}_1[a, x] \wedge \mathsf{S}_2[x, b]$

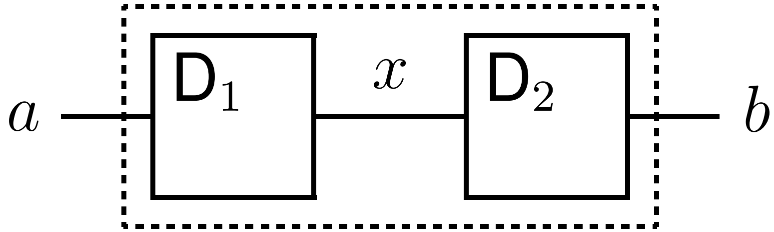
Hiding Internal Structure

- Consider the composite device



$$\mathsf{S}_1[a,x] \wedge \mathsf{S}_2[x,b]$$

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



$$\exists x.\, \mathsf{S}_1[a,x] \wedge \mathsf{S}_2[x,b]$$

- Existential quantification is called a *hiding* operator—it 'hides' internal wires.

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:

$$\mathtt{COMP}(\mathtt{p1}, \ldots, \mathtt{pm}) = \exists \mathtt{w1} \ldots \mathtt{wn}.\ \mathtt{COMP1}(\ldots)\ \wedge\ \mathtt{COMP2}(\ldots)$$

Formulating Correctness

- A key part of formal hardware verification is formalizing what ‘correctness’ *means*.
- The strongest formulation is *equivalence*:

$$\vdash \forall v_1 \ \ldots \ v_n.\ \mathbf{M}[v_1, \ldots, v_n] = \mathbf{S}[v_1, \ldots, v_n]$$

- For *partial* specifications, use *implication*:

$$\vdash \forall v_1 \ \ldots \ v_n.\ \mathbf{M}[v_1, \ldots, v_n] \Rightarrow \mathbf{S}[v_1, \ldots, v_n]$$

- In general, the satisfaction relationship

$$\vdash \mathbf{M}[v_1, \ldots, v_n] \ \underset{abs}{\textsf{sat}} \ \mathbf{S}[abs(v_1), \ldots, abs(v_n)]$$

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

Hiererchical Verification

The hierarchical verification method:

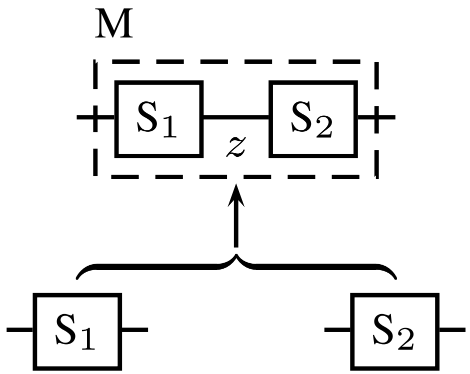
Level 0

Model:

$\vdash \mathrm{M} = \exists z.\, \mathrm{S}_1 \wedge \mathrm{S}_2$

Correctness:

$\vdash \mathrm{M} \underset{F}{\textsf{sat}} \mathrm{S}$



Level 1

Models:

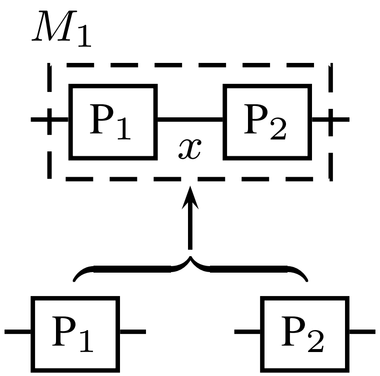
$\vdash \mathrm{M}_1 = \exists x.\, \mathrm{P}_1 \wedge \mathrm{P}_2$

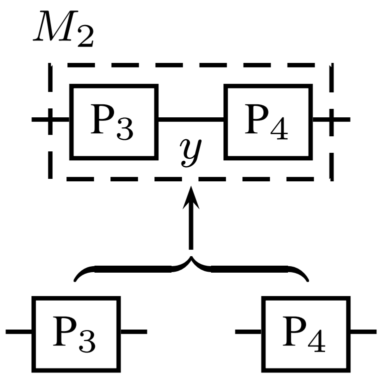
$\vdash \mathrm{M}_2 = \exists y.\, \mathrm{P}_3 \wedge \mathrm{P}_4$

Correctness:

$\vdash \mathrm{M}_1 \underset{G}{\textsf{sat}} \mathrm{S}_1$

$\vdash \mathrm{M}_2 \underset{G}{\textsf{sat}} \mathrm{S}_2$



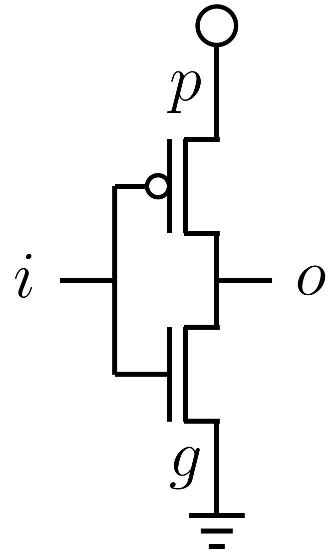


Hierarchical Design—Advantages

- 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

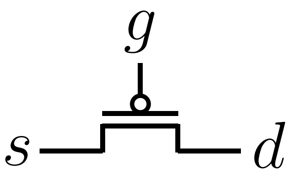
A Simple Correctness Proof

- Here is the design of a CMOS inverter:
- 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

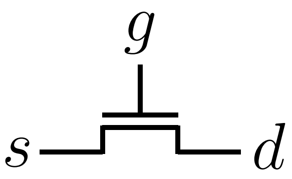


CMOS Primitives

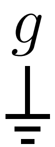
- Formal specifications of primitives:



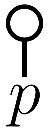
$$\vdash \textsf{Ptran}(g, s, d) = (\neg g \Rightarrow (d = s))$$



$$\vdash \textsf{Ntran}(g, s, d) = (g \Rightarrow (d = s))$$



$$\vdash \textsf{Gnd}\ g = (g = \textsf{F})$$



$$\vdash \textsf{Pwr}\ p = (p = \textsf{T})$$

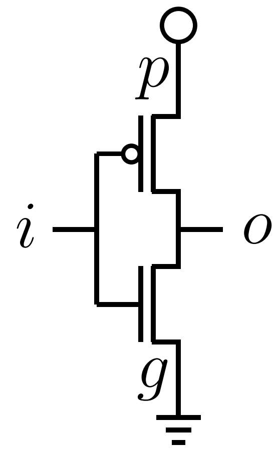
- This is the so-called *switch model* of CMOS.

Design Model and Correctness

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

$$\vdash \mathsf{Inv}(i, o) =$$

$$\exists g\, p.\ \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g\ \wedge$$
$$\mathsf{Ntran}(i, g, o) \wedge \mathsf{Ptran}(i, p, o)$$



- Correctness is formulated by the equivalence:

$$\vdash \forall i\, o.\ \mathsf{Inv}(i, o) = (o = \neg i)$$

This follows by purely logical inference…

The Correctness Proof

- Definition of Inv:

$$
\begin{aligned}
\vdash \mathsf{Inv}(i, o) = {} & \\
& \exists g\, p.\ \mathsf{Pwr}\ p \wedge \mathsf{Gnd}\ g \wedge \\
& \qquad \mathsf{Ntran}(i, g, o) \wedge \mathsf{Ptran}(i, p, o)
\end{aligned}
$$

- Expanding with definitions:

$$
\begin{aligned}
\vdash \mathsf{Inv}(i, o) = {} & \\
& \exists g\, p.\ (p = \mathsf{T}) \wedge (g = \mathsf{F}) \wedge \\
& \qquad (i \Rightarrow (o = g)) \wedge (\neg i \Rightarrow (o = p))
\end{aligned}
$$

- By simple logical reasoning:

$$\vdash \mathsf{Inv}(i, o) = (i \Rightarrow (o = \mathsf{F})) \wedge (\neg i \Rightarrow (o = \mathsf{T}))$$

The Correctness Proof continued

- Simplifying gives:

$$\vdash \mathsf{Inv}(i, o) = (i \Rightarrow \neg o) \land (\neg i \Rightarrow o)$$

- By the law of the contrapositive:

$$\vdash \mathsf{Inv}(i, o) = (o \Rightarrow \neg i) \land (\neg i \Rightarrow o)$$

- By the definition of boolean equality:

$$\vdash \mathsf{Inv}(i, o) = (o = \neg i)$$

- Generalizing the free variables gives:

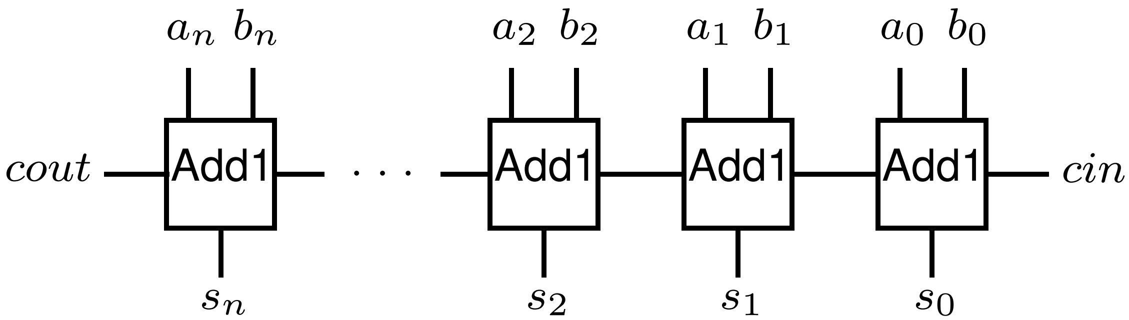
$$\vdash \forall i\, o.\ \mathsf{Inv}(i, o) = (o = \neg i)$$

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

Another Example

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



- We wish to prove that:

$$(2^{n+1} \times cout) + s = a + b + cin$$

- 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

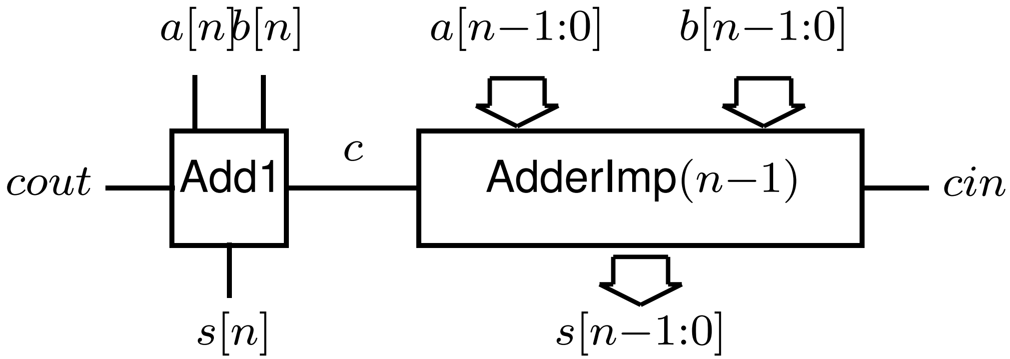
Defining the Model: types

- Specification uses numbers, i.e. values of type num
- Implementation uses words – values of type $word$
  - n^{th} bit of w denoted by $w\,[\,n\,]$
  - $w\,[\,m : n\,]$ denotes bits m to n of w
  - $\mathsf{Bv}(b)$ is the number represented by bit b
  - $\mathsf{V}(w)$ is the natural number represented by word w
- Abstraction from words to numbers (data abstraction):

$$
\begin{aligned}
&\vdash \mathsf{Bv}\ b && = \mathsf{if}\ b\ \mathsf{then}\ 1\ \mathsf{else}\ 0 \\
&\vdash \mathsf{V}\ w[0:0] && = \mathsf{Bv}\ w[0] \\
&\vdash \mathsf{V}\ w[n{+}1:0] && = 2^{n+1}(\mathsf{Bv}\ w[n{+}1])\ +\ \mathsf{V}\ w[n:0]
\end{aligned}
$$

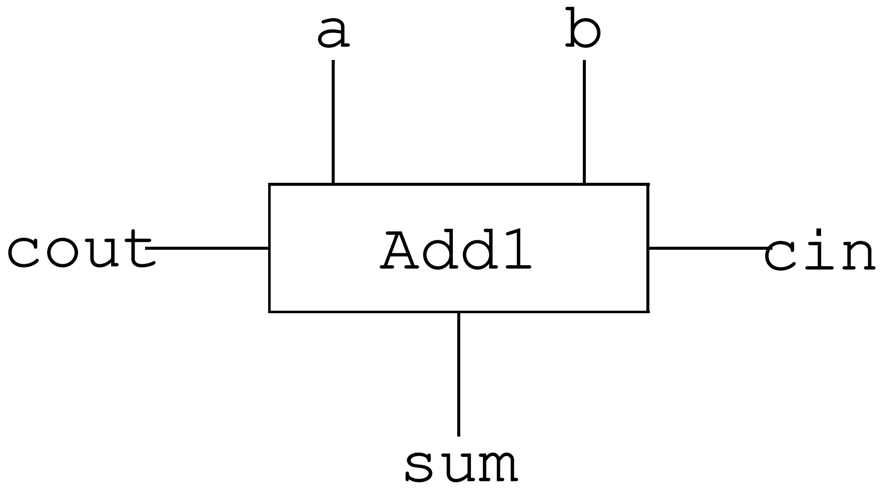
Defining the Model: recursive definition

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



Defining the Model: Add1

- Diagram of a 1-bit full adder:



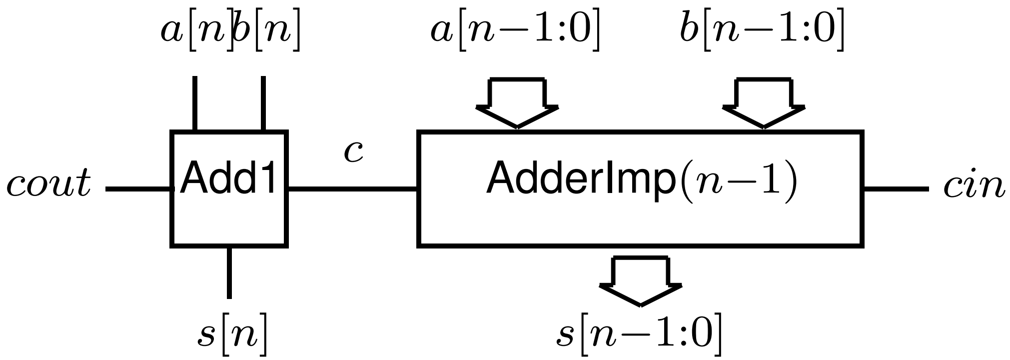
- Lines `a`, `b`, `cin`, `sum` and `cout` carry boolean values

- Specification (note data abstraction from *bool* to *num*):

$$\mathsf{Add1}(\mathtt{a},\mathtt{b},\mathtt{cin},\mathtt{sum},\mathtt{cout}) \;=\; (2\times\mathsf{Bv}(\mathtt{cout})+\mathsf{Bv}(\mathtt{sum}) = \mathsf{Bv}(\mathtt{a})+\mathsf{Bv}(\mathtt{b})+\mathsf{Bv}(\mathtt{cin}))$$

Defining the Model

- Recursive view of an $n{+}1$-bit adder:



- Primitive recursive definition in logic:

$$\mathsf{AdderImp}(0)(a, b, cin, s, cout) = \\ \quad \mathsf{Add1}\,(a[0], b[0], cin, s[0], cout)$$

$$\mathsf{AdderImp}\ n\ (a, b, cin, s, cout) = \\ \quad \exists c.\ \mathsf{Add1}\,(a[n], b[n], c, s[n], cout) \wedge \\ \qquad \mathsf{AdderImp}(n{-}1)(a[n{-}1{:}0], b[n{-}1{:}0], cin, s[n{-}1{:}0], c)$$

Formulation of Correctness

- Logical formulation of correctness:

$$\mathsf{Spec}\ n\ (a, b, cin, s, cout) = ((2^{n+1}\ cout)+s = a+b+cin)$$

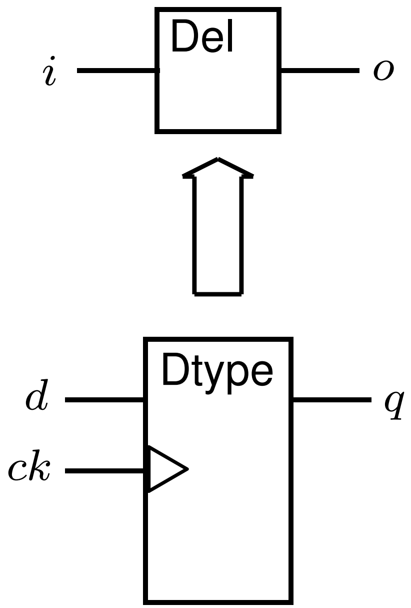
$$\forall n\ a\ b\ cin\ s\ cout.$$
$$\mathsf{AdderImp}\ n\ (a, b, cin, s, cout)$$
$$\Rightarrow$$
$$\mathsf{Spec}\ n\ (\mathsf{V}\ a[n{:}0], \mathsf{V}\ b[n{:}0], \mathsf{Bv}\ cin, \mathsf{V}\ s[n{:}0], \mathsf{Bv}\ cout)$$

- 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:



$\vdash \mathsf{Del}(i,o) = \forall t.\, o(t{+}1) = i\ t$

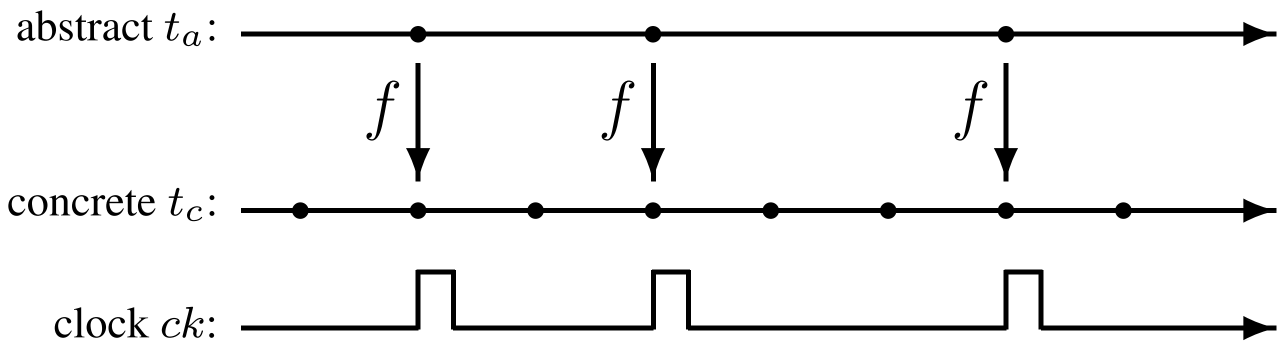
$\vdash \mathsf{Rise}\ ck\ t = \neg ck(t) \wedge ck(t{+}1)$

$\vdash \mathsf{Dtype}(ck,d,q)\ =$
$\quad \forall t.\, q(t{+}1) = \mathsf{if}\ \mathsf{Rise}\ ck\ t\ \mathsf{then}\ d\ t\ \mathsf{else}\ q\ t$

- 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

- A mapping between time-scales:



- Define the temporal abstraction functions:

$\vdash$ Timeof P n = *the time on* t_c *such that P true for* n*th time*

$\vdash$ $signal$ when $P = signal \circ (\mathsf{Timeof}\ P)$

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

Formulating Correctness

- Then correctness is stated by:

$$\begin{aligned}\vdash \forall ck.\ \mathsf{Inf}(\mathsf{Rise}\ ck) \Rightarrow \\ \forall d\ q.\ \mathsf{Dtype}(ck, d, q) \Rightarrow \\ \mathsf{Del}(d\ \mathsf{when}\ (\mathsf{Rise}\ ck), q\ \mathsf{when}\ (\mathsf{Rise}\ ck))\end{aligned}$$

- Note the formal *validity condition*:

$$\vdash \mathsf{Inf}\ P = \forall t.\ \exists\, t'.\ t' > t\ \wedge\ P\ t'$$

Industry use of theorem proving

- Intel
  - floating point algorithms (uses HOL Light system)
  - hardware (uses internal tools Forte/reFL^{ect})
- 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

Summary

- Specifying behaviour:
  - predicates—$\mathrm{S}[a, b, c, d\,]$
- Specifying structure:
  - composition—$\mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$
  - hiding—$\exists x.\ \mathrm{S}_1[a, x] \wedge \mathrm{S}_2[x, b]$
- Formulating correctness:
  - $\vdash \forall v_1 \ \ldots \ v_n.\ \mathbf{M}[v_1, \ldots, v_n] = \mathbf{S}[v_1, \ldots, v_n]$
  - $\vdash \forall v_1 \ \ldots \ v_n.\ \mathbf{M}[v_1, \ldots, v_n] \Rightarrow \mathbf{S}[v_1, \ldots, v_n]$
  - $\vdash \forall v_1 \ \ldots \ v_n.\ \mathbf{M}[v_1, \ldots, v_n] \Rightarrow \mathbf{S}[abs\ v_1, \ldots, abs\ v_n]$
- Abstraction
  - data: $w \mapsto \mathsf{V}(w)$
  - temporal: $sig \mapsto sig\ \mathsf{when}\ (\mathsf{Rise}\ clk)$