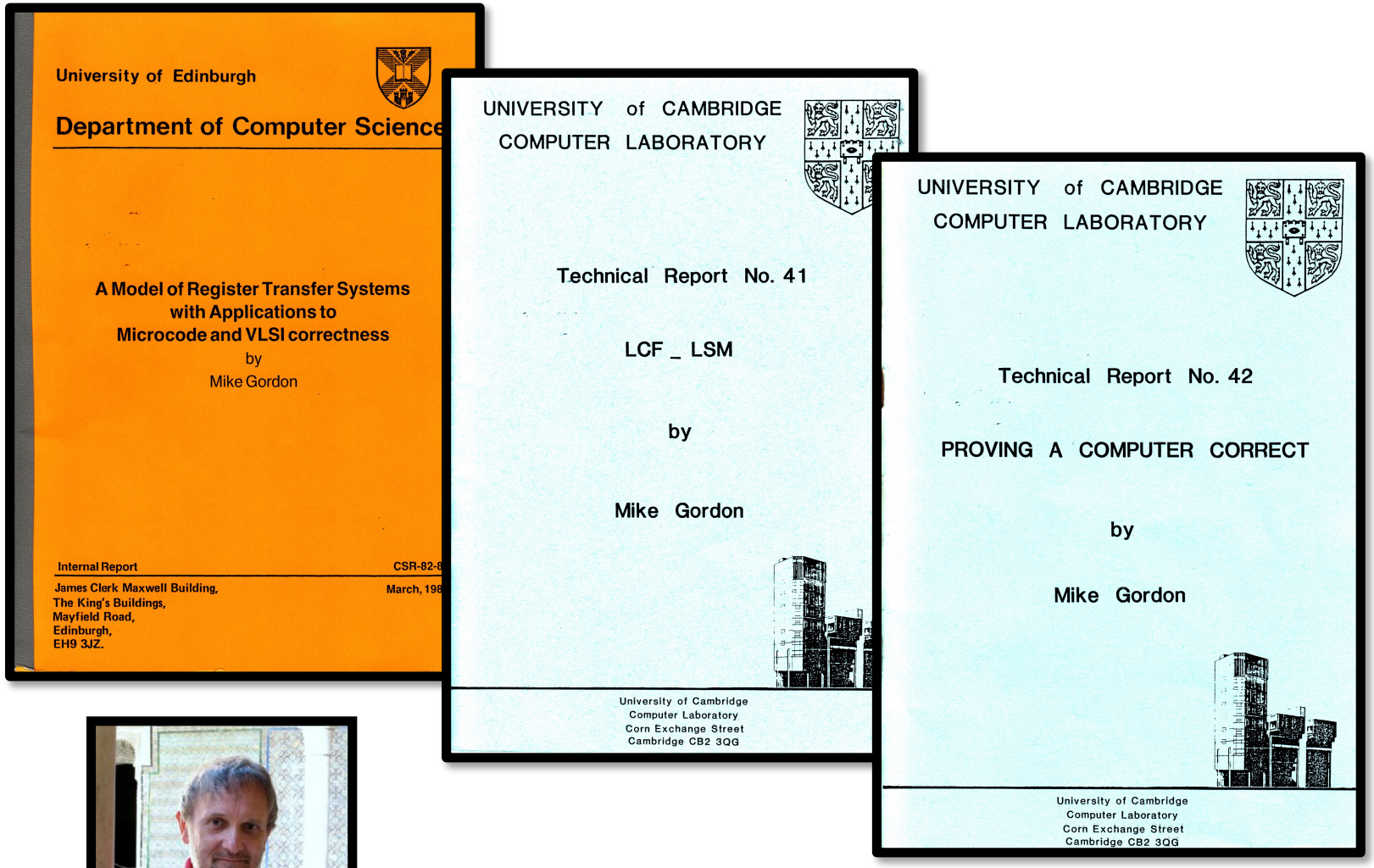
Proof Programming
from LCF_LSM to Goaled via HOL

Professor Tom Melham, FRSE FBCS CEng

University of Oxford

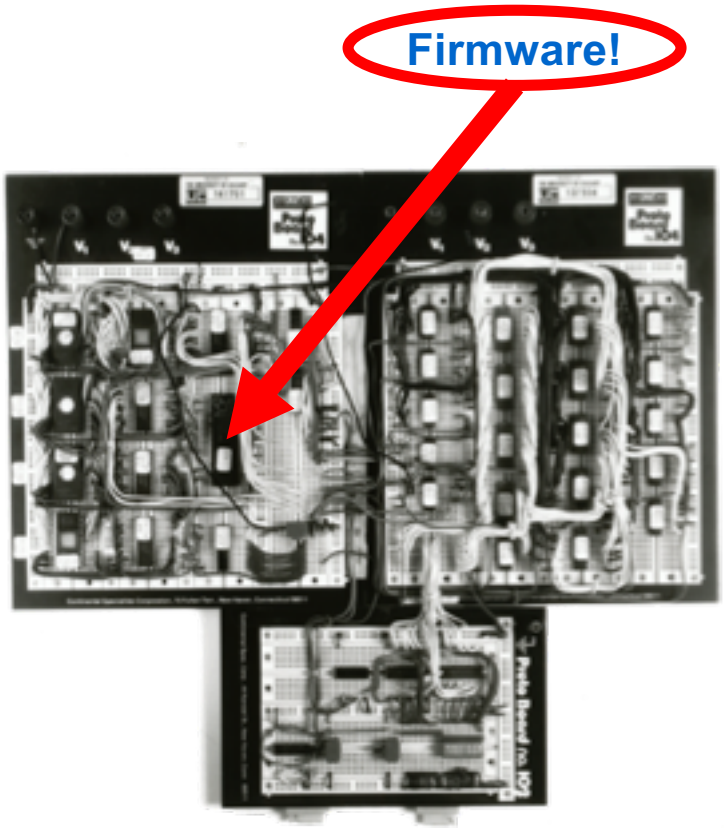
My Introduction to Theorem Proving

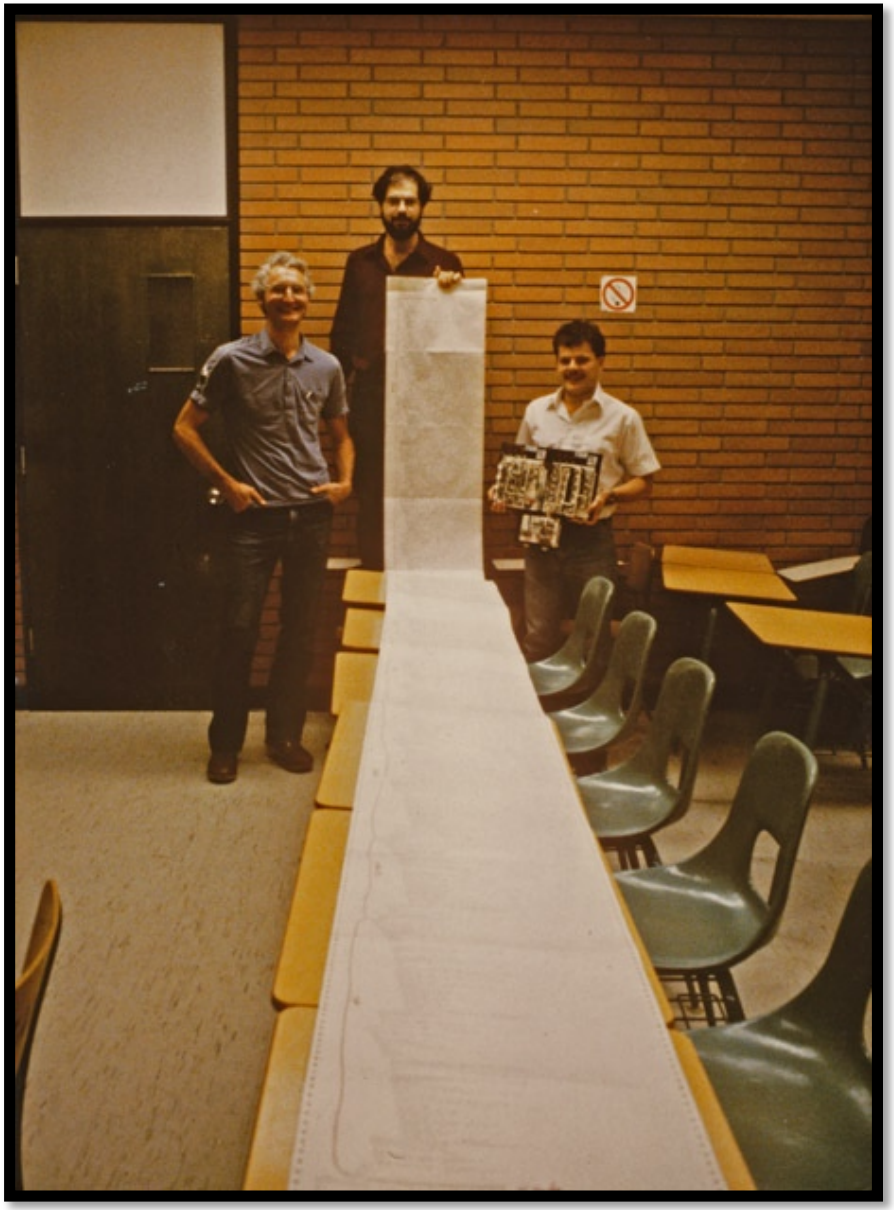


The LCF_LSM System c. 1983

Verification with LCF_LSM

Mostly hand-guided forward symbolic simulation by rewriting...





It Was Really Painful...

```
% FLOODING SINK INTERFACE ASSOCIATIVE MEMORY CORRECTNESS PROOF  %
%                                                              %
% COMPONENT      : DETECT                                      %
% THEORY         : detect_v  VERIFICATION                      %
% PARENTS        : detect_s detect_i                           %
% DESCRIPTION    : Verification of device DETECT.              %
%                                                              %
% AUTHOR         : T. MELHAM                                   %
% DATE           : 84.02.05                                    %

% Create the theory detect_v
new_theory `detect_v`;;

% Parent theories are detect_
map new_parent [`detect_i`; `

% We also need some LIST and
map new_parent [`list_ax`; `w


% Fetch device specifications

let NAND8 = axiom `primitives

let SPLIT16 = axiom `primitiv

let DETECT_IMP = axiom `detec

let DETECT = axiom `detect_s`


% Expand the implementation b
let thm1 = EXPAND_IMP [] [NAN

% Fetch equality axiom from l
let LIST8_EQ = axiom `list_ax`

% Specialize to equality with
let lem1 = SPECL ["T";"T";"T"

% Specialize the rest of the
let lem2 = SPEC_ALL lem1;;

% We need a rewrite going the
let lem3 = SYM lem2;;

%
```

```
%


% Set up a goal to prove.                                      %
let goal1 = ([], "!b. T = b == b");;

% This function will produce the symmetric <=> thm.            %
let SYMIFF th = let conj = IFF_CONJ (SPEC_ALL th)
                in let c1 = CONJUNCT1 conj
                in let c2 = CONJUNCT2 conj
                in CONJ_IFF (CONJ c2 c1);;

% We need a tactic for BOOL cases.                             %
let CASES_SPEC_TAC (w1,w) =
    (GEN_TAC THEN
     STRUCT_CASES_TAC (SPEC (fst (dest_forall w)) BOOL_CASES)) (w1,w);;

% Take BOOL cases then rewrite then use axiom BOOL_EQ_DISTINCT  %
let tac1 = REPEAT CASES_SPEC_TAC
           THEN   REWRITE_TAC [EQ;SYMIFF(NEG_EQ)]
           THEN   ACCEPT_TAC (CONJUNCT1 BOOL_EQ_DISTINCT) ;;

% Prove goal1 using tac1 giving lemma lem4.                    %
let lem4 = TAC_PROOF (goal1,tac1);;

% Rewrite  T = x to x using lemma lem4.                        %
let lem5 = REWRITE_RULE [lem4] lem3;;

% Use lem5 to rewrite thm1.                                    %
let thm2 = REWRITE_RULE [(GEN_ALL lem5)] thm1;;

% Fetch LOW_BYTE axiom.                                        %
let LOW_BYTE = axiom `constants` `LOW_BYTE`;;

% Fetch HIGH_BYTE axiom.                                       %
let HIGH_BYTE = axiom `constants` `HIGH_BYTE`;;

% Rewrite LOW_BYTE and HIGH_BYTE in DETECT.                    %
let thm3 = REWRITE_RULE [LOW_BYTE;HIGH_BYTE] DETECT;;

%
```

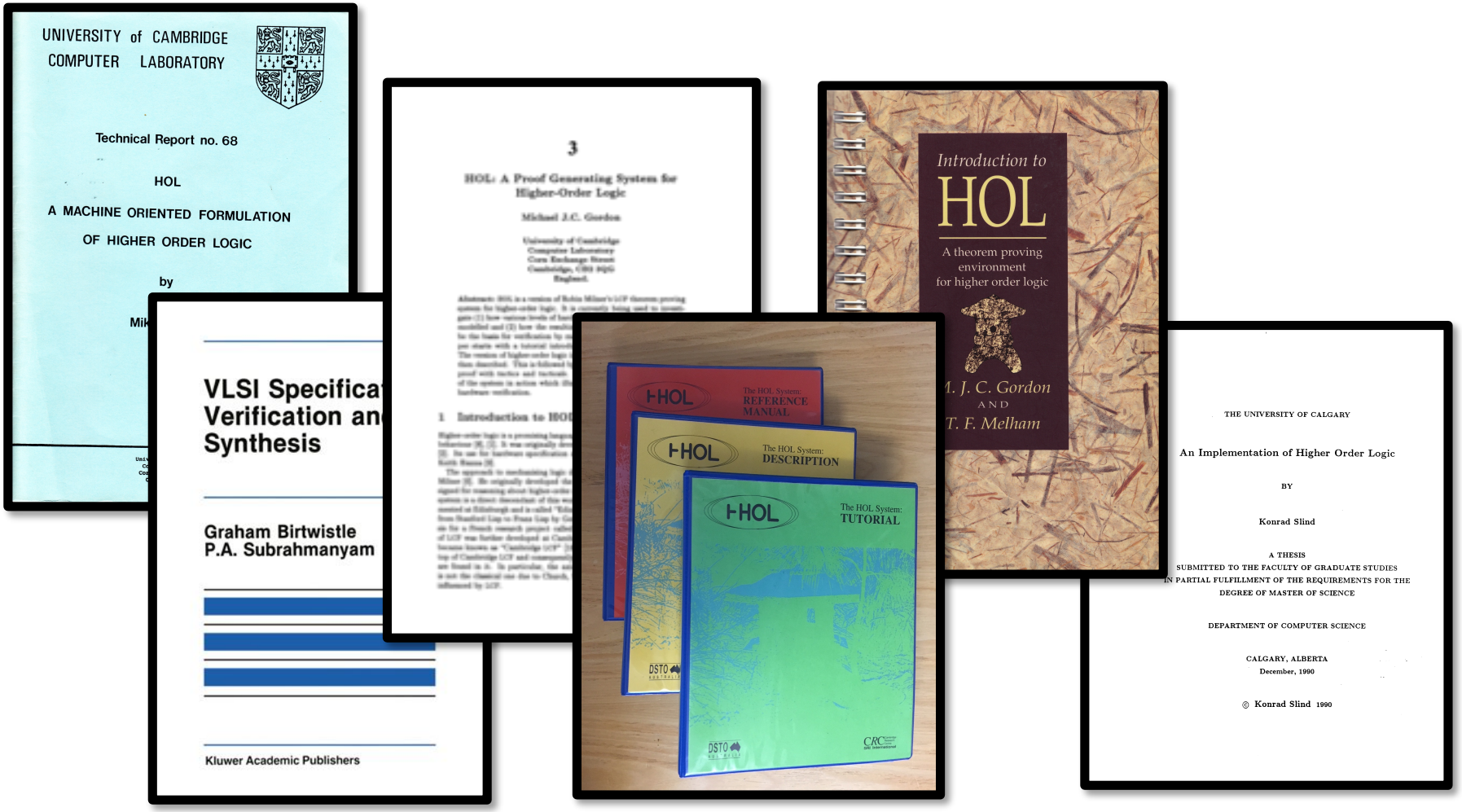
It Was Really Painful...

```
% FLOODING SINK INTERFACE ASSOCIATIVE MEMORY CORRECTNESS PROOF  %
%                                                              %
% COMPONENT      : DETECT                                      %
% THEORY         : detect_v  VERIFICATION                      %
% PARENTS        : detect_s detect_i                           %
% DESCRIPTION    : Verification of device DETECT.              %
%                                                              %
% AUTHOR         : T. MELHAM                                   %
% DATE           : 84.02.05

% Create the theory detect_v
new_theory `detect_v`;;

% Parent theories are detect_
map new_parent [`detect_i`; `

% We also need some LIST and
map new_parent [`list_ax`; `w
```

```
%

% Set up a goal to prove.                                      %
let goal1 = ([], "!b. T = b == b");;

% This function will produce the symmetric <=> thm.            %
let SYMIFF th = let conj = IFF_CONJ (SPEC_ALL th)
                in let c1 = CONJUNCT1 conj
                in let c2 = CONJUNCT2 conj
                in CONJ_IFF (CONJ c2 c1);;
```

```
% Rewrite  T = x to x using lemma lem4.                        %
let lem5 = REWRITE_RULE [lem4] lem3;;

% Use lem5 to rewrite thm1.                                    %
let thm2 = REWRITE_RULE [(GEN_ALL lem5)] thm1;;

% Fetch LOW_BYTE axiom.                                        %
let LOW_BYTE = axiom `constants` `LOW_BYTE`;;

% Fetch HIGH_BYTE axiom.                                       %
let HIGH_BYTE = axiom `constants` `HIGH_BYTE`;;

% Rewrite LOW_BYTE and HIGH_BYTE in DETECT.                    %
let thm3 = REWRITE_RULE [LOW_BYTE;HIGH_BYTE] DETECT;;
```

The Emergence of HOL, HOL88, HOL90



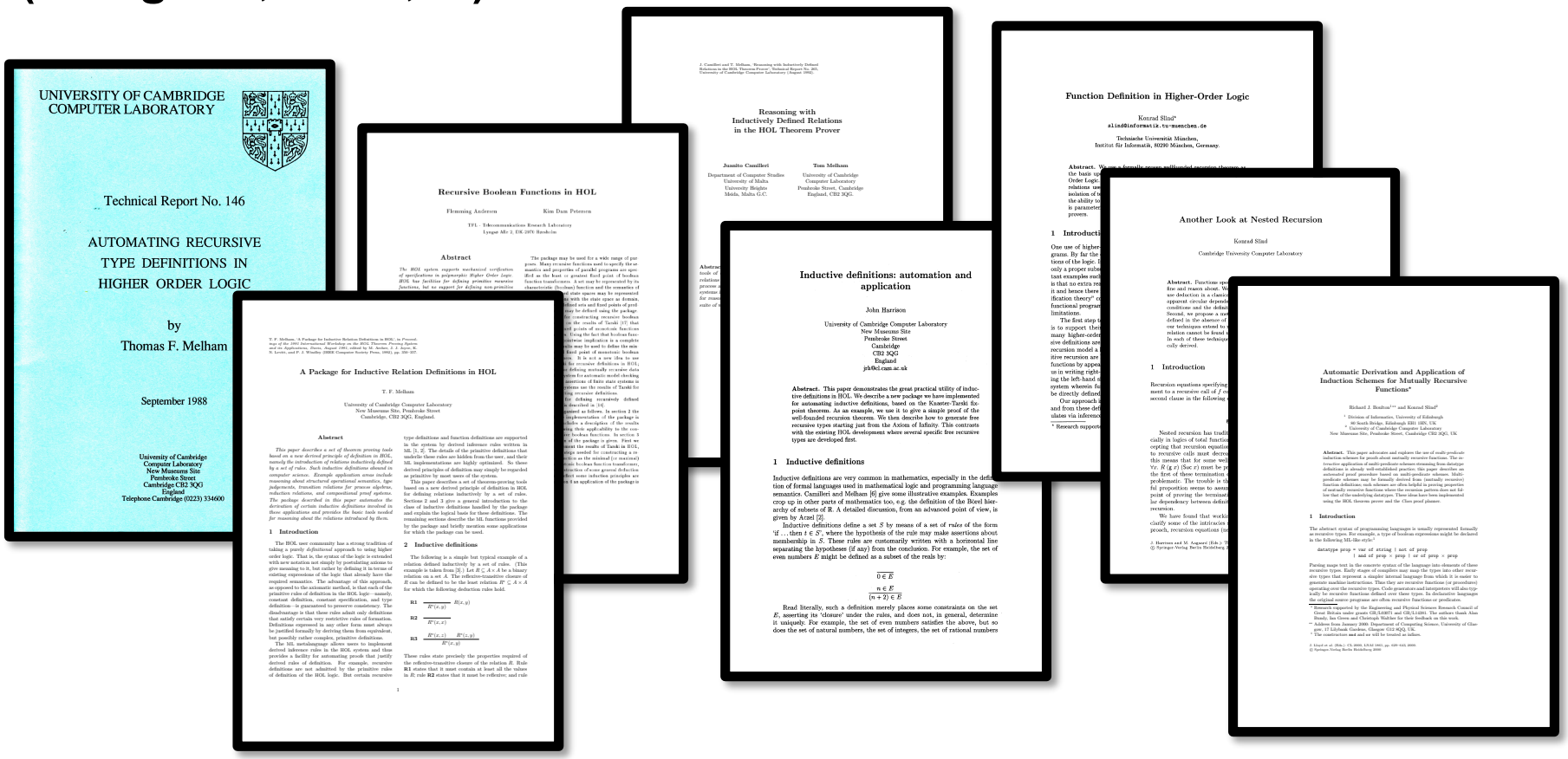
Derived Definitional Principles

Recursive types, functions (Melham, Gunter, ...)

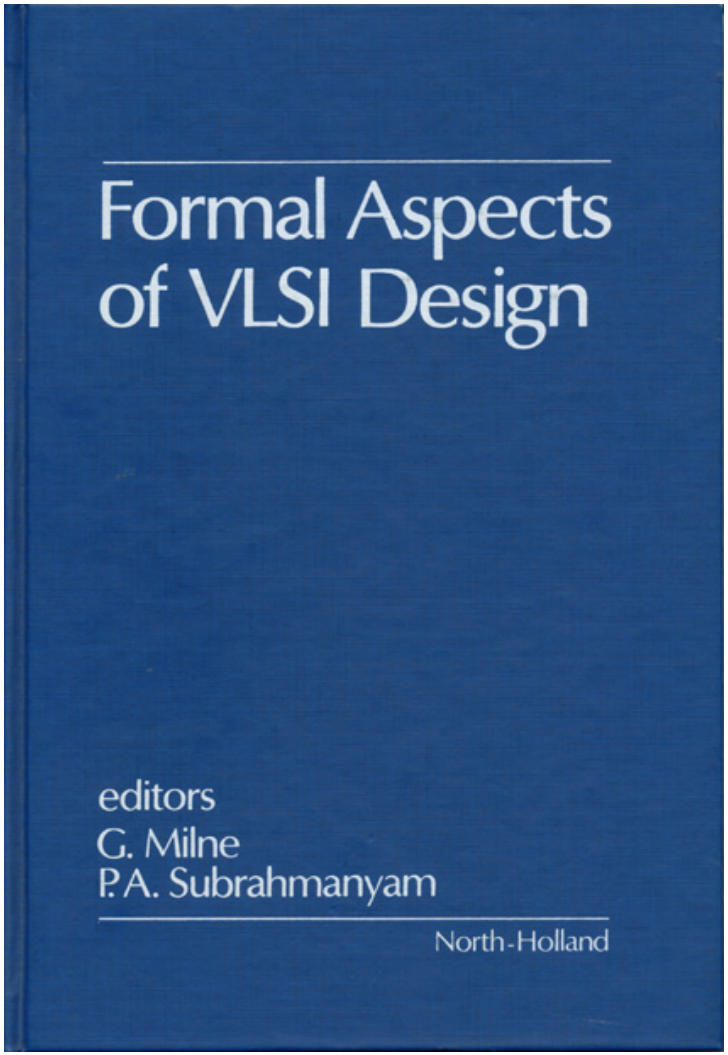
Recursive Boolean Functions (Andersen, Petersen)

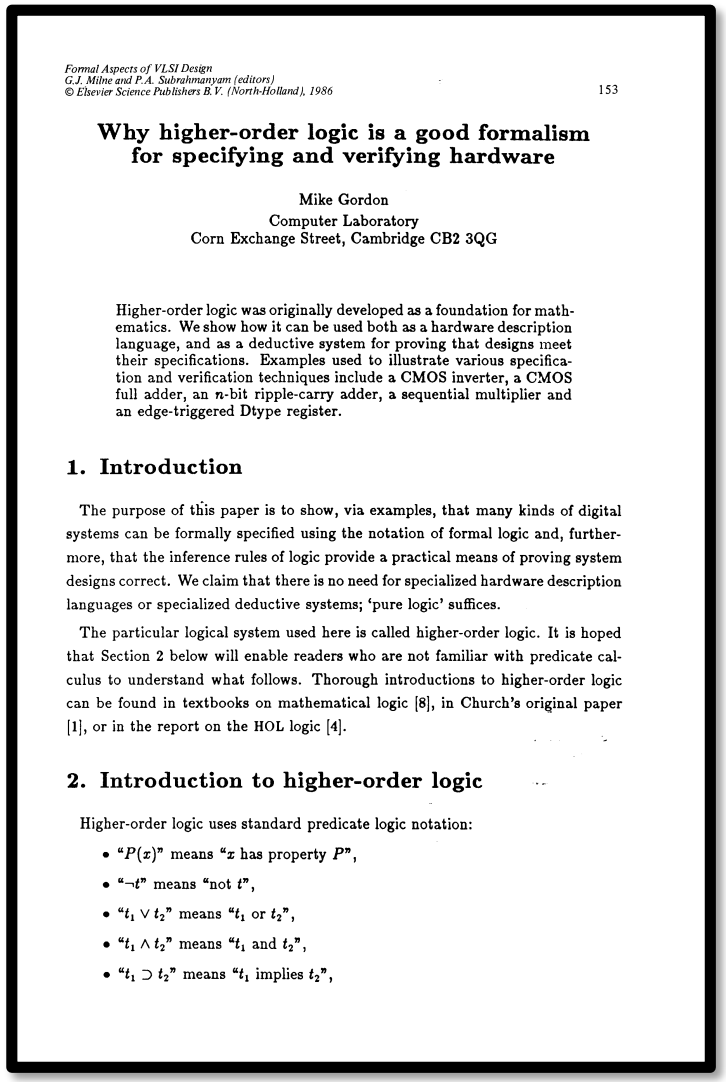
Inductive definitions (Melham, Harrison, ...)

General/mutual recursive functions using well-founded orderings (Ploegerts, Slind, ...)



Hardware Verification in Higher Order Logic





Formal Aspects of VLSI Design
G.J. Milne and P.A. Subrahmanyam (editors)

Why higher-order logic is a good formalism for specifying and verifying hardware

Mike Gordon
Computer Laboratory
Corn Exchange Street, Cambridge CB2 3QG

Higher-order logic was originally developed as a foundation for mathematics. We show how it can be used both as a hardware description language, and as a deductive system for proving that designs meet their specifications. Examples used to illustrate various specification and verification techniques include a CMOS inverter, a CMOS full adder, an n-bit ripple-carry adder, a sequential multiplier and an edge-triggered Dtype register.

1. Introduction

The purpose of this paper is to show, via examples, that many kinds of digital systems can be formally specified using the notation of formal logic and, furthermore, that the inference rules of logic provide a practical means of proving system designs correct. We claim that there is no need for specialized hardware description languages or specialized deductive systems; 'pure logic' suffices.

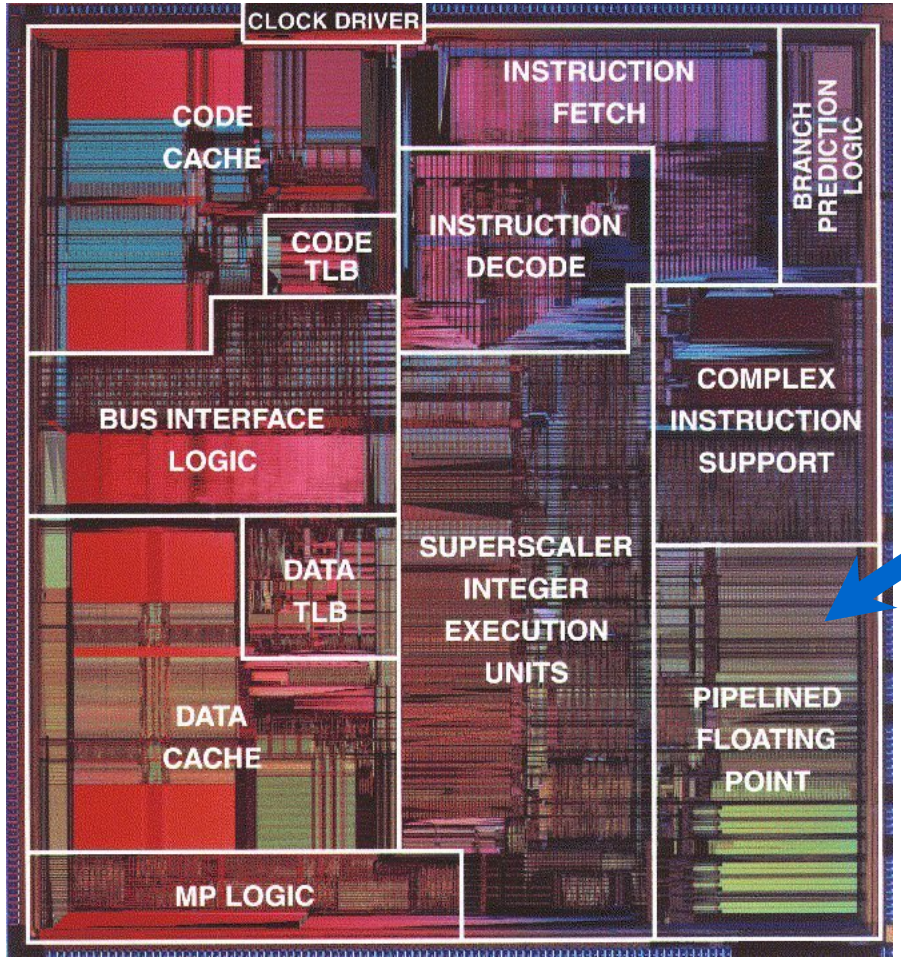
The particular logical system used here is called higher-order logic. It is hoped that Section 2 below will enable readers who are not familiar with predicate calculus to understand what follows. Thorough introductions to higher-order logic can be found in textbooks on mathematical logic [8], in Church's original paper [1], or in the report on the HOL logic [4].

2. Introduction to higher-order logic

Higher-order logic uses standard predicate logic notation:

- "$P(x)$" means "x has property P",
- "$\neg t$" means "not t",
- "$t_1 \vee t_2$" means "t_1 or t_2",
- "$t_1 \wedge t_2$" means "t_1 and t_2",
- "$t_1 \supset t_2$" means "t_1 implies t_2",

Fast Forward to... Intel's Infamous FDIV Bug




```
4195835.0 / 3145727.0

= 1.333 820 449 136 241 002 (Correct)
= 1.333 739 068 902 037 589 (Flawed)
```

Huge potential cost (100s of M$)

Processor Verification at Intel

IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 9, SEPTEMBER 2005 1381

An Industrially Effective Environment for Formal Hardware Verification

Carl-Johan H. Seger, Robert B. Jones, *Member, IEEE*, John W. O'Leary, *Member, IEEE*, Tom Melham, Mark D. Aagaard, *Member, IEEE*, Clark Barrett, and Don Syme

***Abstract*—The Forte formal verification environment for datapath-dominated hardware is described. Forte has proven to be effective in large-scale industrial trials and combines an efficient linear-time logic model-checking algorithm, namely the symbolic trajectory evaluation (STE), with lightweight theorem proving in higher-order logic. These are tightly integrated in a general-purpose functional programming language, which both allows the system to be easily customized and at the same time serves as a specification language. The design philosophy behind Forte is presented and the elements of the verification methodology that make it effective in practice are also described.**

***Index Terms*—BDDs, formal verification, model checking, symbolic trajectory evaluation, theorem proving.**

I. Introduction

FUNCTIONAL validation is one of the major challenges in chip design today, with conventional approaches to design validation a serious bottleneck in the design flow. Over the past ten years, formal verification [1] has emerged as a complement to simulation and has delivered promising results in trials on industrial-scale designs [2]–[6].

Formal equivalence checking is widely deployed to compare the behavior of two models of hardware, each represented as a finite state machine or simply a Boolean expression (often using binary decision diagrams (BDDs) [7]). It is typically used in industry to validate the output of a synthesis tool against a "golden model" expressed in a register-transfer level hardware description language (HDL), and in general to check consistency between other adjacent levels in the design flow.

Property checking with a model checker [8]–[11] also involves representing a design as a finite state machine, but it has wider capabilities than equivalence checking. Not only can one check that a design behaves the same as another model, one can also check that the hardware possesses certain desirable properties expressed more abstractly in a temporal logic. An example is checking that all requests are eventually acknowledged in a protocol. Model checking is currently much less widely used in practice than equivalence checking.

Theorem proving [12], [13] allows higher level and more abstract properties to be checked. It provides a much more expressive language for stating properties—for example, higher order logic [14]—and it can deal with infinite-state systems. In particular, it allows one to reason with unknowns and parameters, so a general class of designs can be checked—for example, parameterized IP blocks [15]. Industrially, theorem proving is still viewed as a very advanced technology, and its use is not widespread.

Equivalence checkers and model checkers both suffer from severe capacity limits. In practice, only small fragments of systems can be handled directly with these technologies, and much current research is aimed at extending capacity. Of course, it is unrealistic to expect a completely automatic model-checking solution. Instead, one needs to find good ways of using human intelligence to extract the maximum potential from model-checking algorithms and to decompose problems into appropriate pieces for automated analysis. One approach is to combine model-checking and BDD-based methods with theorem proving [16]–[18]. The hope is that theorem proving's power and flexibility will enable large problems to be broken down or transformed into tasks a model checker finds tractable. Another approach is to extend the top level of a model checker with ad hoc theorem-proving rules and procedures [19].

This paper describes a formal verification system called Forte that combines an efficient linear-time logic model checking algorithm, namely symbolic trajectory evaluation (STE) [20], with lightweight theorem proving in higher-order logic. Th
are interfaced to and tightly integrated with FL [21], a s
typed, higher order functional programming language
general-purpose programming language, FL allows th
environment to be customized and large proof effort
organized and scripted effectively. FL also serves as an
sive specification language at a level far above the te
logic primitives.

The Forte environment has proven to be highly e
in large-scale industrial trials on datapath-dominated ha
[3], [22], [23]. The restricted temporal logic of ST
not, however, limit Forte to pure datapath circuits. Man
control circuits are "datapath-as-control," and these can
handled effectively. In addition, the tight connection to
order logic and theorem proving provides great flexib

Manuscript received January 20, 2004; revised June 23, 2004. This paper was recommended by Associate Editor J. H. Kukula.

C.-J. H. Seger, R. B. Jones, and J. W. O'Leary are with Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124 USA (e-mail: Carl.Seger@intel.com; Robert.B.Jones@intel.com; John.W.O'Leary@intel.com).

T. Melham is with the Oxford University Computing Laboratory, Oxford OX1 3QD, U.K. (e-mail: Tom.Melham@comlab.ox.ac.uk).

M. D. Aagaard is with the Department of Electrical and Computer Engineering, University of Waterloo, Wateroo, ON N2L 3G1, Canada (e-mail: maagaard@uwaterloo.ca).

C. Barrett is with the Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY 10012 USA (e-mail: barrett@cs.nyu.edu).

D. Syme is with Microsoft Research, Cambridge CB3 0FB, U.K. (e-mail: dsyme@microsoft.com).

Digital Object Identifier 10.1109/TCAD.2005.850814

Formal Verification

Practical Formal Verification in Microprocessor Design

Robert B. Jones, John W. O'Leary, and Carl-Johan H. Seger
Intel

Mark D. Aagaard
University of Waterloo

Thomas F. Melham
University of Glasgow

Practical application of formal methods requires more than advanced technology and tools; it requires an appropriate methodology. A verification methodology for data-path-dominated hardware combines model checking and theorem proving in a customizable framework. This methodology has been effective in large-scale industrial trials, including verification of an IEEE-compliant floating-point adder.

FUNCTIONAL VALIDATION is one of the major challenges in chip design today, with test generation, test bench construction, and simulation consuming a significant portion of the design effort. Throughout the 1990s, formal verification emerged as a promising complement to conventional simulation-based validation.[1] Most formal verification research concerns algorithms and focuses on tool capacity limits. Yet almost any serious verification effort faces many practical difficulties besides capacity. In a large verification project, the effort required to organize the multitude of tasks, specifications, and verification scripts can limit the quality and productivity of the work.

We tackle this problem by coupling our research on verification algorithms and tools with research on verification methodology. Our goal is to address the realities of design practice—rapid changes and incomplete specifications—while producing high-quality results and improving verification productivity. Our methodology systematically organizes a large verification effort's many interdependent activities and provides a guiding structure for the verification process.

Any formal verification tool researcher is keenly aware that what is a routine verification for the technology expert or tool developer may be very difficult for others to duplicate. Our methodology addresses this problem by tailoring a formal, custom-built verification framework, Forte, to industrial-scale circuits and industrial design environments. Forte combines an efficient, linear temporal logic model-checking algorithm, called symbolic trajectory evaluation (STE),[2] with lightweight theorem proving. FL—a custom, general-purpose functional programming language—tightly integrates the model checker and the theorem prover. This combination of model checking, theorem proving, and a general-purpose programming language makes the verification environment customizable and lets large verification efforts be organized effectively.

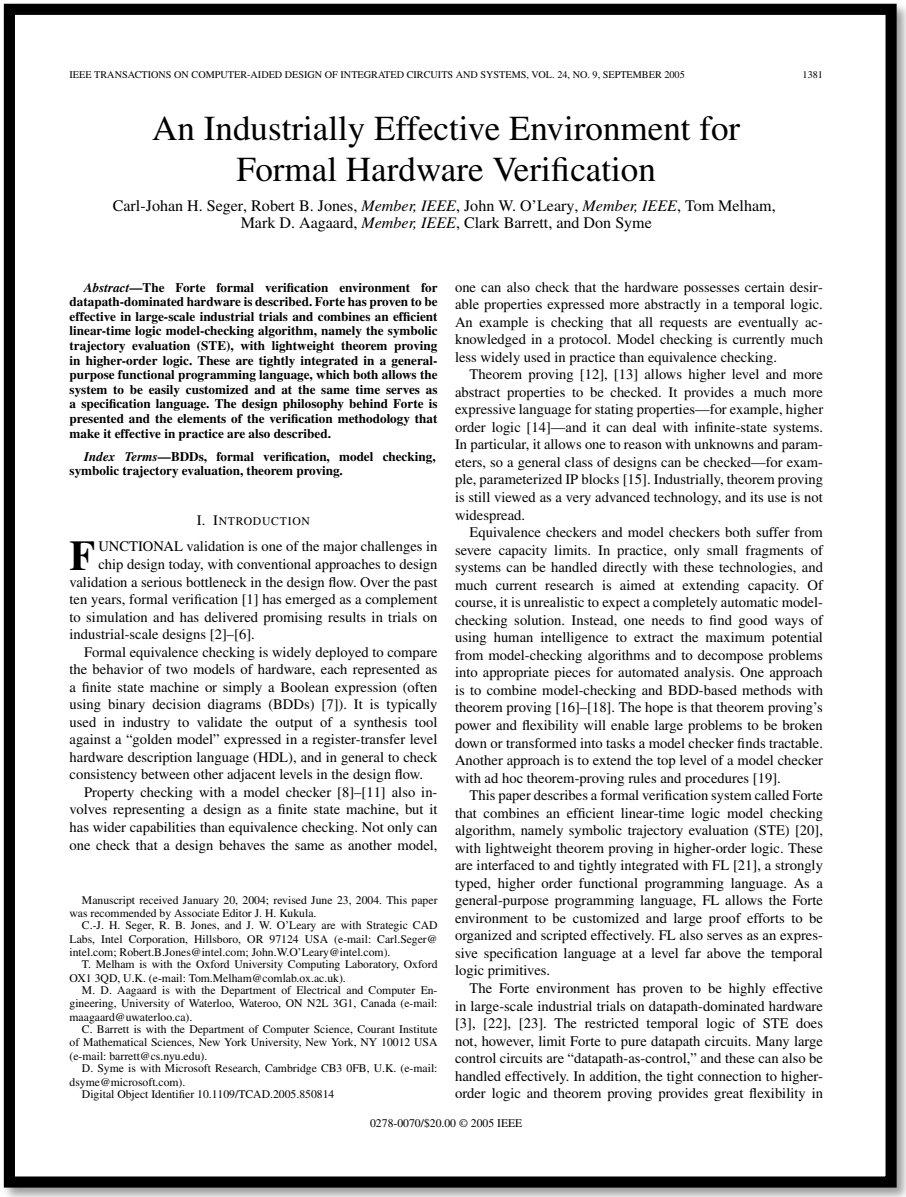
Our methodology has evolved over several years of use on fully custom, high-performance

0740-7475/01/$10.00 © 2001 IEEE

Intel's *Forte* System

A full *programming environment*

- executable specifications
- simulation property logic
- symbolic simulation
- abstraction
- SAT and BDDs
- functional scripting
- debugging support
- large set of libraries
- theorem proving in higher order logic. . .



IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 24, NO. 9, SEPTEMBER 2005 1381

An Industrially Effective Environment for Formal Hardware Verification

Carl-Johan H. Seger, Robert B. Jones, *Member, IEEE*, John W. O'Leary, *Member, IEEE*, Tom Melham, Mark D. Aagaard, *Member, IEEE*, Clark Barrett, and Don Syme

***Abstract*—The Forte formal verification environment for datapath-dominated hardware is described. Forte has proven to be effective in large-scale industrial trials and combines an efficient linear-time logic model-checking algorithm, namely the symbolic trajectory evaluation (STE), with lightweight theorem proving in higher-order logic. These are tightly integrated in a general-purpose functional programming language, which both allows the system to be easily customized and at the same time serves as a specification language. The design philosophy behind Forte is presented and the elements of the verification methodology that make it effective in practice are also described.**

***Index Terms*—BDDs, formal verification, model checking, symbolic trajectory evaluation, theorem proving.**

I. Introduction

FUNCTIONAL validation is one of the major challenges in chip design today, with conventional approaches to design validation a serious bottleneck in the design flow. Over the past ten years, formal verification [1] has emerged as a complement to simulation and has delivered promising results in trials on industrial-scale designs [2]–[6].

Formal equivalence checking is widely deployed to compare the behavior of two models of hardware, each represented as a finite state machine or simply a Boolean expression (often using binary decision diagrams (BDDs) [7]). It is typically used in industry to validate the output of a synthesis tool against a "golden model" expressed in a register-transfer level hardware description language (HDL), and in general to check consistency between other adjacent levels in the design flow.

Property checking with a model checker [8]–[11] also involves representing a design as a finite state machine, but it has wider capabilities than equivalence checking. Not only can one check that a design behaves the same as another model, one can also check that the hardware possesses certain desirable properties expressed more abstractly in a temporal logic. An example is checking that all requests are eventually acknowledged in a protocol. Model checking is currently much less widely used in practice than equivalence checking.

Theorem proving [12], [13] allows higher level and more abstract properties to be checked. It provides a much more expressive language for stating properties—for example, higher order logic [14]—and it can deal with infinite-state systems. In particular, it allows one to reason with unknowns and parameters, so a general class of designs can be checked—for example, parameterized IP blocks [15]. Industrially, theorem proving is still viewed as a very advanced technology, and its use is not widespread.

Equivalence checkers and model checkers both suffer from severe capacity limits. In practice, only small fragments of systems can be handled directly with these technologies, and much current research is aimed at extending capacity. Of course, it is unrealistic to expect a completely automatic model-checking solution. Instead, one needs to find good ways of using human intelligence to extract the maximum potential from model-checking algorithms and to decompose problems into appropriate pieces for automated analysis. One approach is to combine model-checking and BDD-based methods with theorem proving [16]–[18]. The hope is that theorem proving's power and flexibility will enable large problems to be broken down or transformed into tasks a model checker finds tractable. Another approach is to extend the top level of a model checker with ad hoc theorem-proving rules and procedures [19].

This paper describes a formal verification system called Forte that combines an efficient linear-time logic model checking algorithm, namely symbolic trajectory evaluation (STE) [20], with lightweight theorem proving in higher-order logic. These are interfaced to and tightly integrated with FL [21], a strongly typed, higher order functional programming language. As a general-purpose programming language, FL allows the Forte environment to be customized and large proof efforts to be organized and scripted effectively. FL also serves as an expressive specification language at a level far above the temporal logic primitives.

The Forte environment has proven to be highly effective in large-scale industrial trials on datapath-dominated hardware [3], [22], [23]. The restricted temporal logic of STE does not, however, limit Forte to pure datapath circuits. Many large control circuits are "datapath-as-control," and these can also be handled effectively. In addition, the tight connection to higher-order logic and theorem proving provides great flexibility in

Manuscript received January 20, 2004; revised June 23, 2004. This paper was recommended by Associate Editor J. H. Kukula.

C.-J. H. Seger, R. B. Jones, and J. W. O'Leary are with Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124 USA (e-mail: Carl.Seger@intel.com; Robert.B.Jones@intel.com; John.W.O'Leary@intel.com).

T. Melham is with the Oxford University Computing Laboratory, Oxford OX1 3QD, U.K. (e-mail: Tom.Melham@comlab.ox.ac.uk).

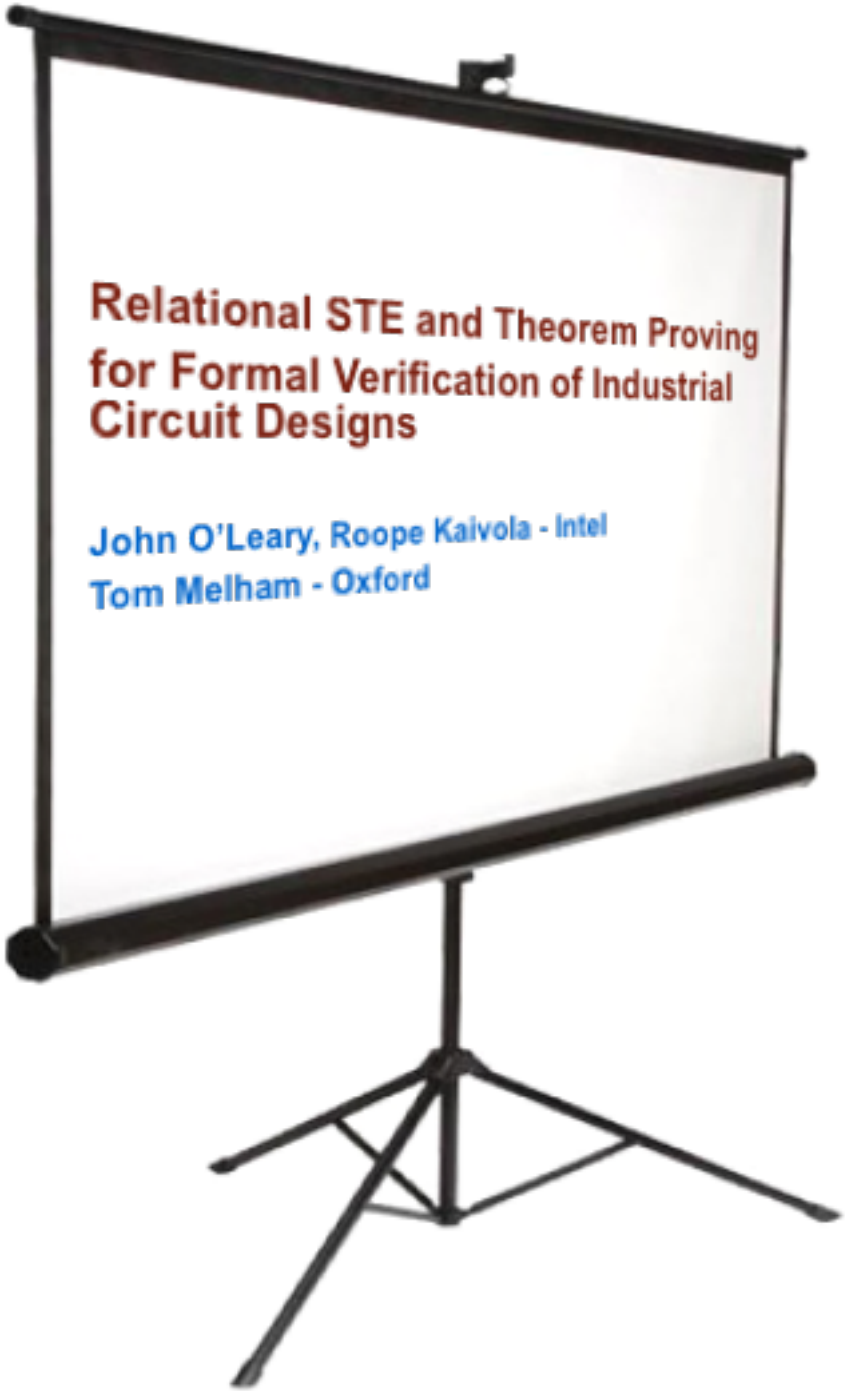
M. D. Aagaard is with the Department of Electrical and Computer Engineering, University of Waterloo, Wateroo, ON N2L 3G1, Canada (e-mail: maagaard@uwaterloo.ca).

C. Barrett is with the Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY 10012 USA (e-mail: barrett@cs.nyu.edu).

D. Syme is with Microsoft Research, Cambridge CB3 0FB, U.K. (e-mail: dsyme@microsoft.com).

Digital Object Identifier 10.1109/TCAD.2005.850814

Based around a bespoke FP language with *Edinburgh ML Syntax*.



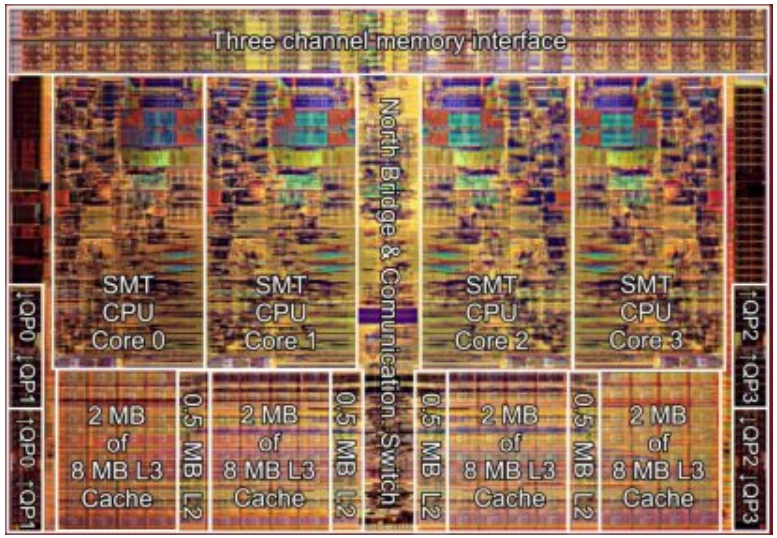
FMCAD 2013

Portland, OR

CPU Datapath Verification at Intel

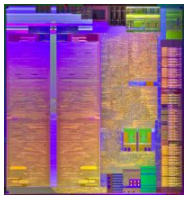
- **Thousands of operations**
  - **Integer, FP, SSE, AVX, …**
  - ***Miscellaneous***
  - **Various operating modes, flags, faults**

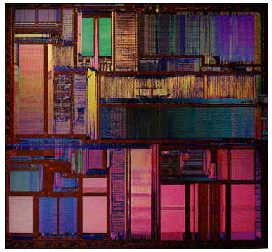
- **Live RTL, changing frequently until a few weeks before tapeout**

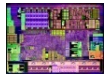


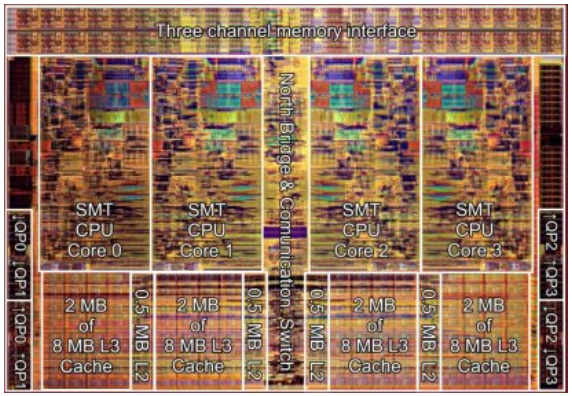
Scaling Up

- Tens of designs
- Different optimization points
- Different teams
- Different countries
- Not only CPUs
- Not all have FV experts on staff

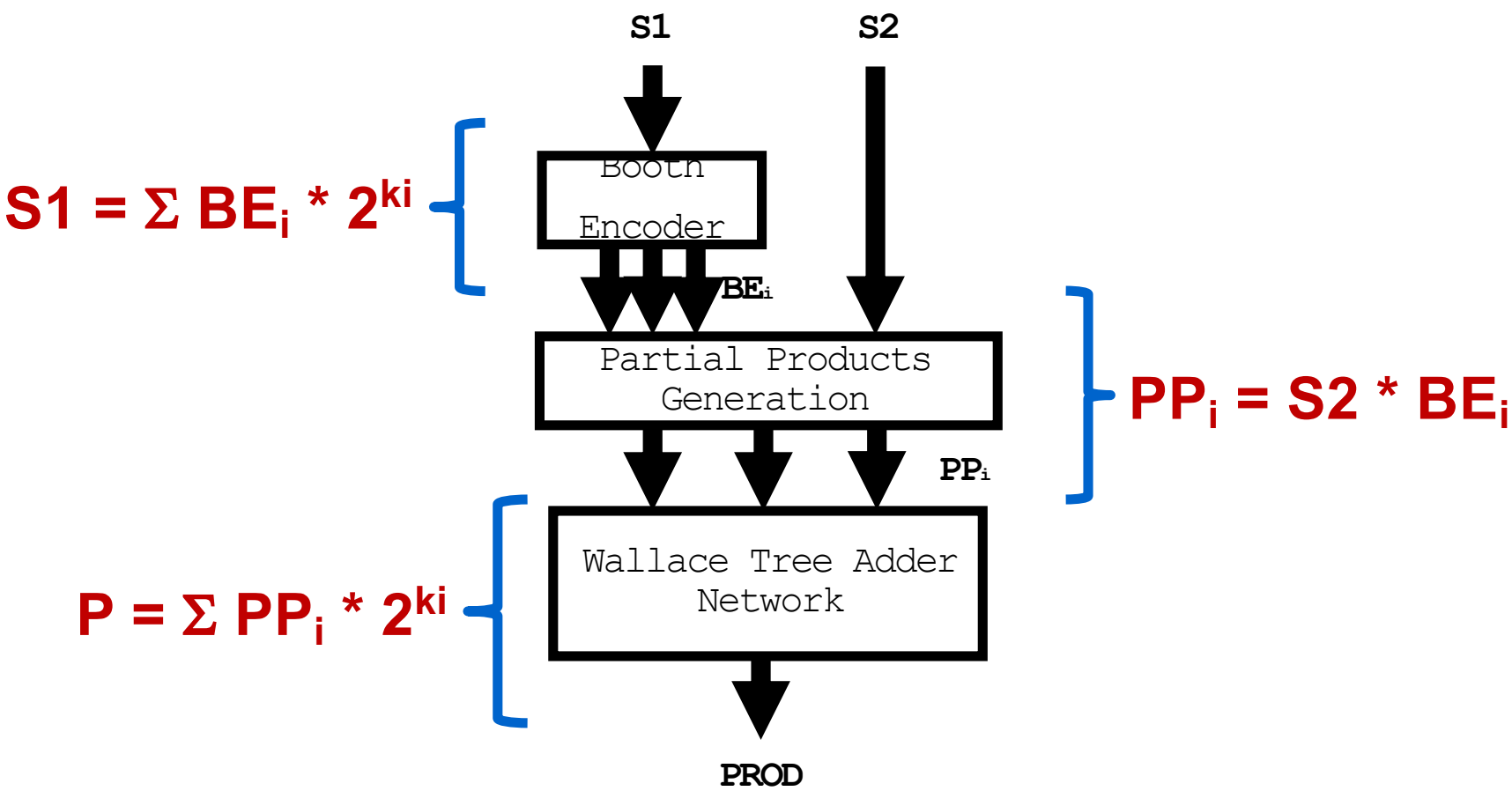




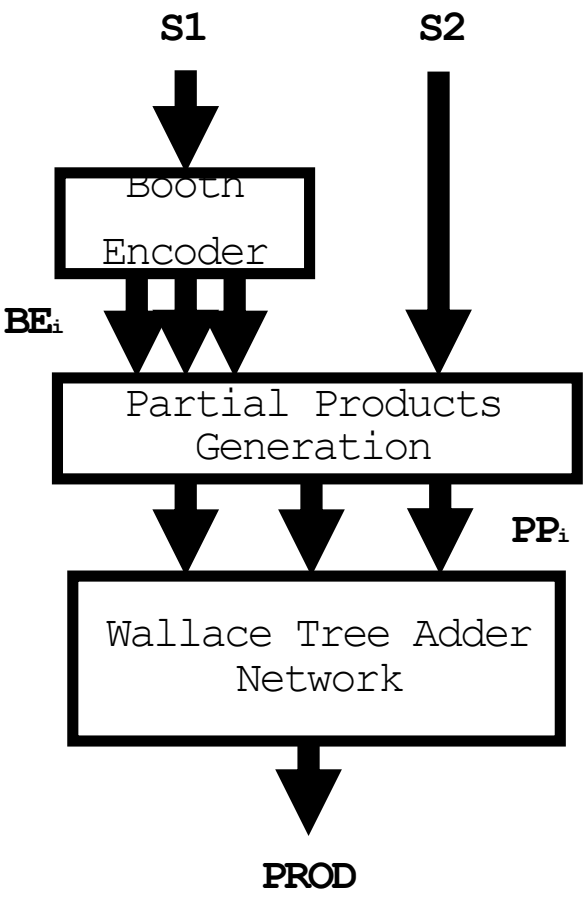




Integer Multiplier



The Multiplier Zoo

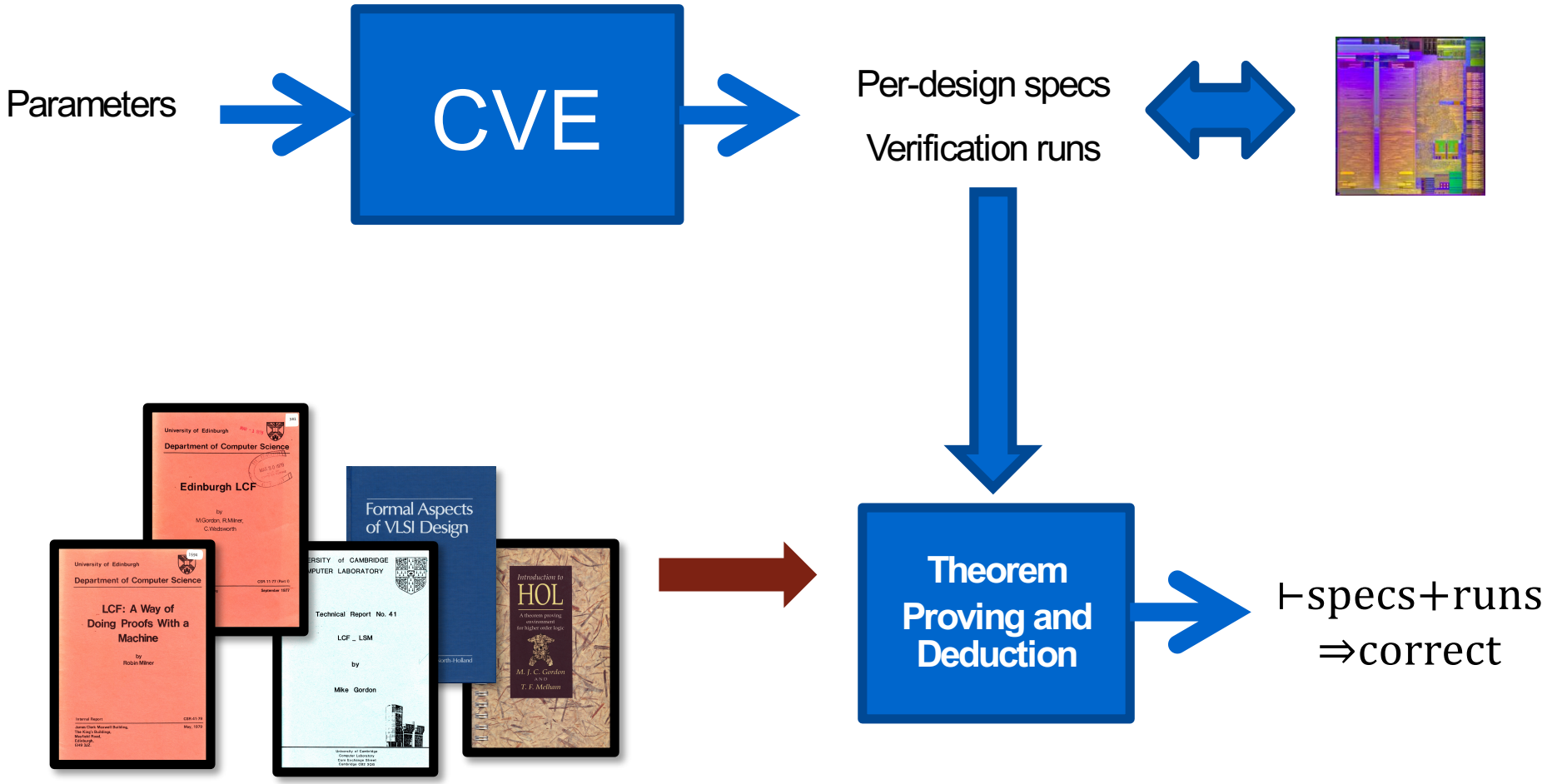


- **10-20 multipliers**
- **Hand designed**
- **Hand optimized**
- **All different**

FV Challenges

- Varying specs and verification strategies
  - Implementation changes from design to design
  - Multiplier always requires decomposition
- Ten designers but not ten multiplier FV experts
- Same story for integer, MMX, FP, SSE, GPU flavors of multiplication, addition, division, …
  - Some operations require even more intricate decomposition

The Solution



The Solution *Done Right*

- **An executable logic for writing specs and verification scripts:**

  reFLect (Jim Grundy et al.)

- **A symbolic simulator that with relational specifications in logic:**

  rSTE (Roope Kaivola et al.)

- **A tightly integrated theorem-prover for the deduction:**

  Goaled (John O'Leary et al.)

The reFLect Language

- **Core syntax:**

$$n,o,p \ ::= \ \mathrm{k} \ | \ v \ | \ n\ o \ | \ \underbrace{\lambda p.\ n \ \square \ o}_{\textit{pattern matching}} \ | \ \underbrace{\langle\, n \,\rangle \ | \ \hat{}n{:}\sigma}_{\textit{reflection}}$$

- **Plus extensions driven by necessity**

  BDDs built in as a primitive type

  Quotient types

  Overloading

  Named function parameters

  Records

  Possibly unsafe features: references, I/O, recursion

Higher Order Logic of reFLect

- **HOL, following Church:**

  λ-calculus
  +
  logical constants
  +
  rules

- **The Goaled logic:**

  reFLect
  +
  logical constants
  +
  rules

Basic idea in both systems:

$n \rightarrow p$ means $\vdash n = p$

Define ∀, ∃, etc by axioms

Add rules for function equality

Proof by evaluation

The Goaled Theorem Prover

- **LCF-style implementation, following HOL and HOL Light**

  Thm is a protected data type, constructible only through a small set of trusted function calls (a.k.a. inference rules)

- **Features driven by necessity**

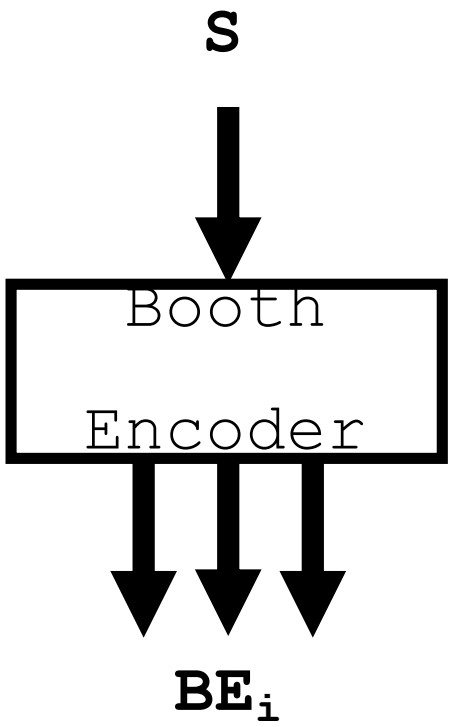
  Theories: of reFLect data types, natural numbers, integers, rationals, lists, pairs, reFLect ADTs, ...

  Proof automation: rewriting, first order solving, linear arithmetic

  Bitstring arithmetic

  Support for the reflect language extensions

Limitations of STE



$$S = \sum_{i=0}^{N-1} BE_i * 2^{ki}$$

- Trajectory assertion:

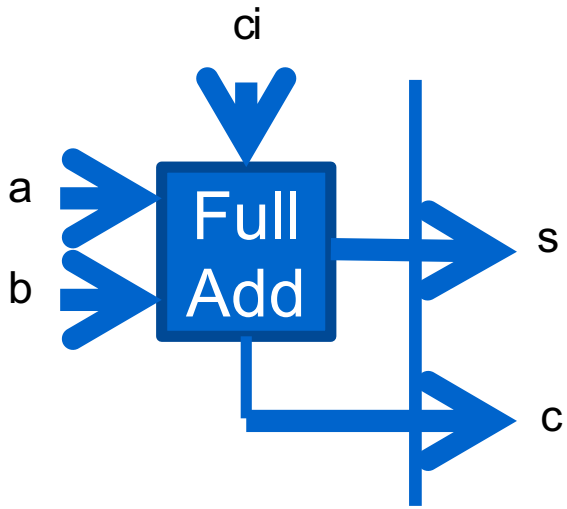
  ckt |= [[S is v ==>> (BE_i is f_i(v))]]

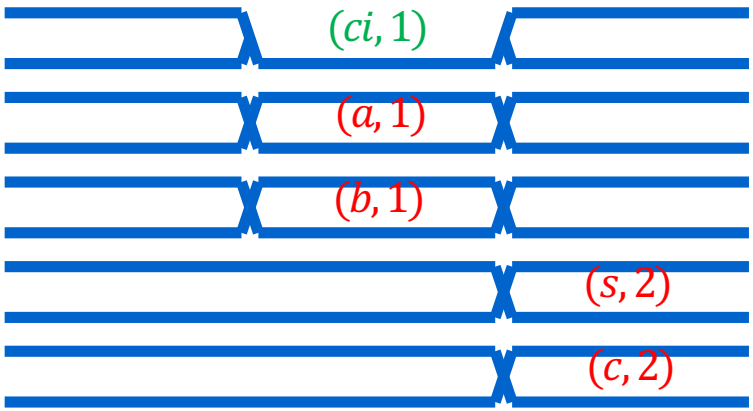
- **But**

  You need a special purpose reasoning system for this special purpose logic

  Relational specifications cannot be expressed directly

Relational STE Intuition





rSTE ckt

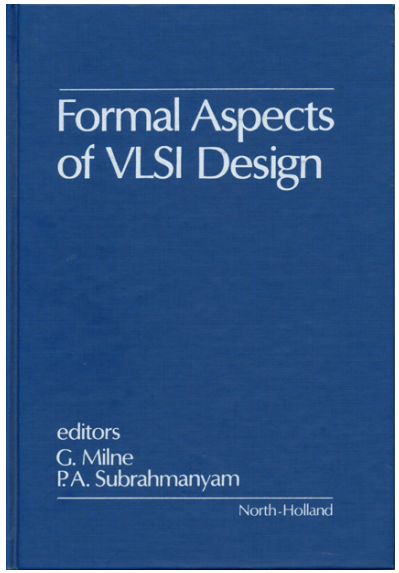
["$!\,(ci, 1)$"] ["$(a, 1) + (b, 1) = (s, 2) + 2\times(c, 2)$"]

From Relational STE to Logic

- **Theorem relating STE simulations to pure logic:**

$$\forall ckt\ cin\ cout.$$
$$rSTE\ ckt\ cin\ cout \Rightarrow$$
$$\forall e.\ [\![ckt]\!]\ e \Rightarrow$$
$$holds\ cin\ e \Rightarrow holds\ cout\ c$$

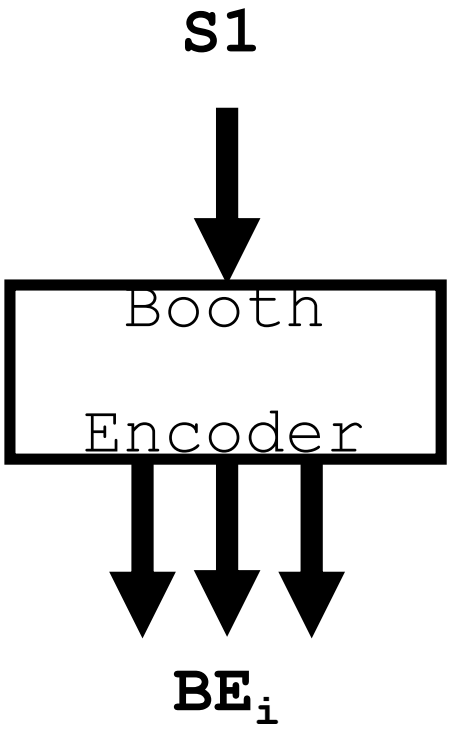
From ad-hoc specification language to pure higher order logic...



Why Higher-Order Logic is a good Formalism for Specifying and Verifying Hardware

Can a Simulator Verify a Circuit?

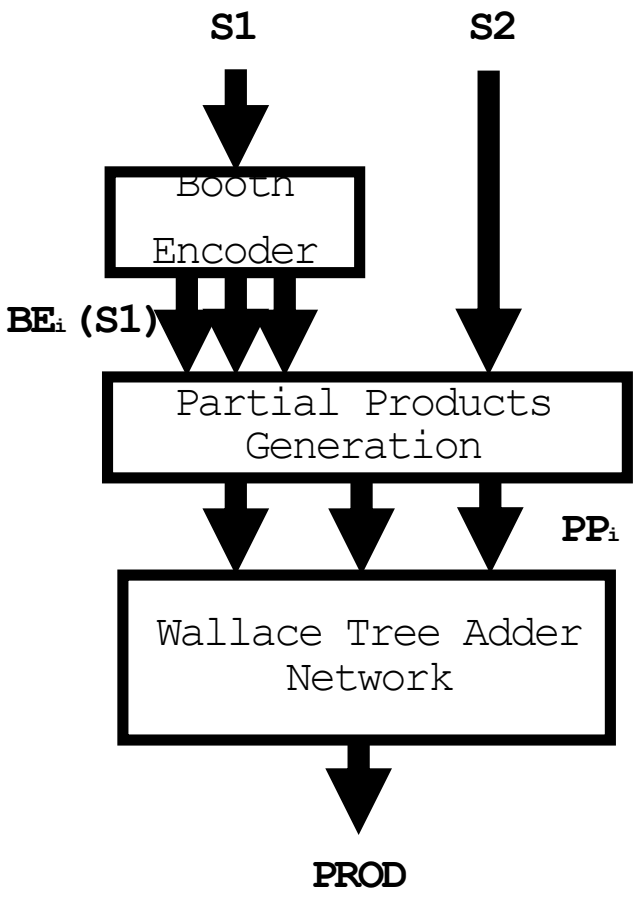
Relational STE in Action



- $\forall e. [\![ckt]\!] e \Rightarrow$
  $holds\ [\,]\ e \Rightarrow holds\ [boothc]\ e$
- $\forall e. [\![ckt]\!] e \Rightarrow holds\ (boothc)\ e$
- $\forall e. [\![ckt]\!] e \Rightarrow eqn1(s2i\ e\ s1)$
- $\forall e. [\![ckt]\!] e \Rightarrow$

$$\left(s2i\ e\ s1 = \sum_{i=0}^{N-1} BE_i(s2i\ e\ s1) \times 2^{ki} \right)$$

The Theorem Proving Part



$$\forall e. [\![ckt]\!]e \Rightarrow$$
$$(\bigwedge s2i\ e\ pp_i = BE_i(s2i\ e\ s1) \times s2i\ e\ s2)$$

$$\forall e. [\![ckt]\!]e \Rightarrow$$
$$\left(s2i\ e\ prod = \sum_{i=0}^{N-1}(s2i\ e\ pp_i) \times 2^{ki}\right)$$

$$\forall e. [\![ckt]\!]e \Rightarrow$$
$$\left(s2i\ e\ s1 = \sum_{i=0}^{N-1} BE_i(s2i\ e\ s1) \times 2^{ki}\right)$$

$$\forall e. [\![ckt]\!]e \Rightarrow$$
$$(s2i\ e\ prod = s2i\ e\ s1 \times s2i\ e\ s2)$$

Common Thread in this Work

Programmed deductive algorithms for a class of theorems

vs

Interactive goal-directed proof with tactics

Heuristic proof methods or algorithmic decision procedures

Common Threads in this Work

The Influence of Ideas that Mike Gordon Upheld

LCF methodology

Importance of a formal definition framework

Simple types

Hardware verification

Hardware specification in a ‘standard’ logic

Relational hardware specification

Applied and practical

Rigour, truth, modesty, and generosity