

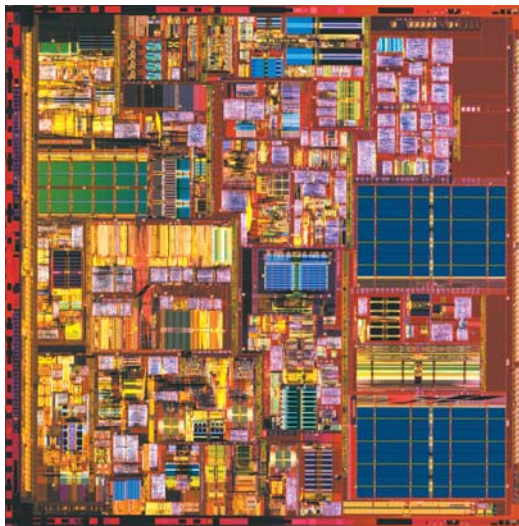
Practical Formal Verification at an Industrial Scale

Tom Melham
University of Oxford
Computing Laboratory



Joint Work With
Strategic CAD Labs
Intel Corporation

Pentium 4 Microprocessor

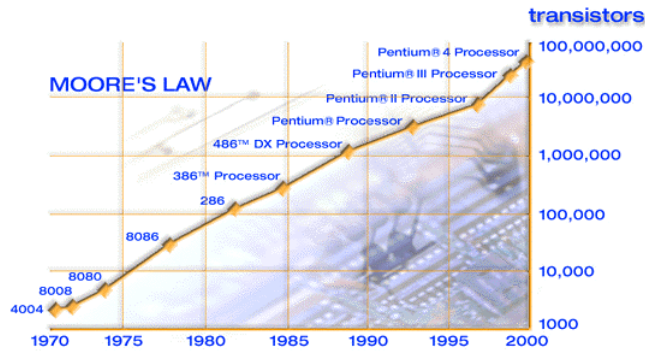


- 55M transistors
- 146mm²
- 0.13μm process
- 3.2GHz

Among the most
complex artefacts
ever produced by
humans...

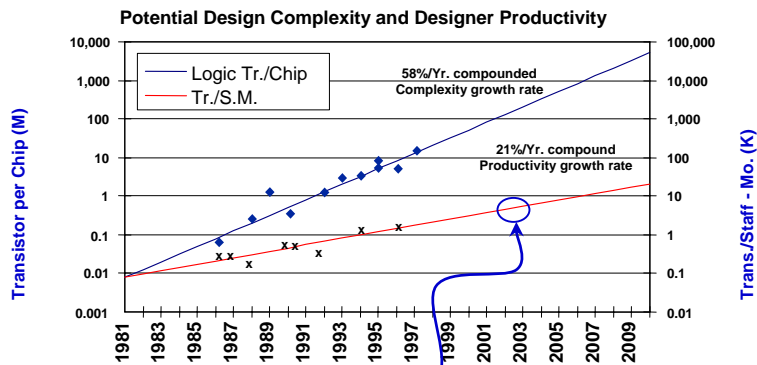
Moore's Law

The number of transistors doubles around every 18 months



3

Design Productivity Crisis



800 people × 3 years
costing \$360,000,000

How many people will it take in 2010?

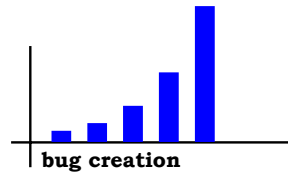
4

The *Verification Crisis*

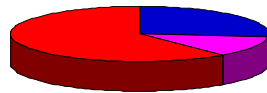
- Computer simulation



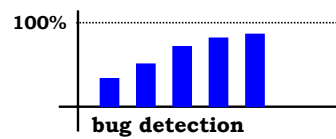
- The verification crisis



- Typical effort



- Design
- Electrical
- Verification



5

Formal Verification

- Use mechanized logic to *prove* correctness
 - no simulation test cases needed
 - complete coverage, effectively exhaustive simulation
- But it's not easy
 - scalability, ease of use, getting specifications
- Different philosophies
 - verification (insurance)
 - bug finding (productivity)
 - verification through design

6

Verification in Practice

- Industrial experience shows
 - theory by itself is not usable (in practice)
 - theory with a system is not usable (in practice)
 - something else is needed - an FV [methodology](#)
- A systematic, pragmatic approach to organizing large-scale verification efforts:
 - clearly stated plan for the sequence and purpose of the many interdependent activities involved
 - guiding structure for the verification code artifacts to be produced

7

Methodological Principles

- Realistic
 - complete specifications are usually not available
 - access to design engineers is always limited
- Structured
 - helps new users learn
 - increases productivity of experienced users
- Transparent and Sound
 - should be clear what has and has not been proved
 - must be sound, no false positives
- Incremental
 - must be able to measure progress
 - effort should develop 'debugging value' early
 - past work should help with recovery when proofs break

8

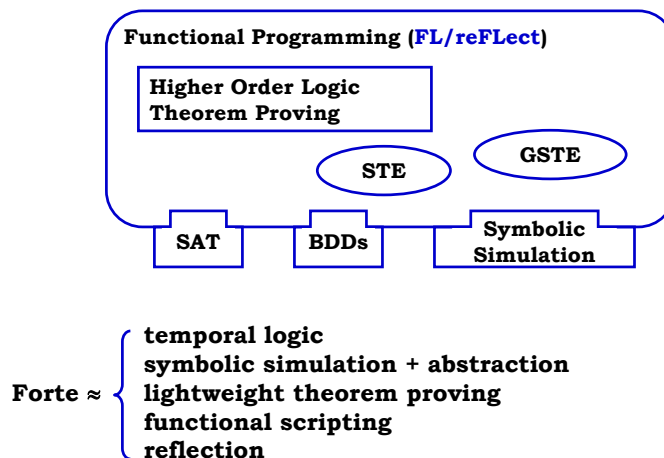
Methodological Principles

- Provides good feedback
 - bulk of verification effort is debugging
 - optimize for proof failure, not success
 - provide focused feedback and tight debugging loop
- Top-down *and* bottom-up
 - top-down for problem reduction/abstraction
 - bottom-up for understanding design and tool capacity limits
- Supports regression
 - verification artifacts should be maintainable
 - and easily adaptable to track design or specification changes
- Allows effort reuse
 - verification is human-intensive
 - amortize cost over changes or multiple designs
 - proof effort should be relatively circuit independent

9

Forte Formal Verification System

Intel's interactive verification environment



10

Some Forte Verifications

- Verification of gate-level floating point implementations against IEEE specification: **FADD**, **FSUB**, **FMUL**, **FDIV**, **FSQRT**, ...
- IA32 instruction-length decoder.

- Pentium 4 FV effort found several 'high quality' bugs
 - FP multiply data space bug
 - interaction between FP ops in different threads, corrupting data
 - FP overflow/underflow flags incorrect in certain processor modes
 - overall FV effort (not just FPU, not just Forte) found ~20 high quality bugs hard to detect by dynamic testing [Bentley, DAC'01].

- Intel has used these techniques on two generations of lead IA32 processors and their proliferations.

11

Role of Functional Programming

- Specification
 - data modeling (e.g. floating point operations)
 - stipulating I/O behavior
 - stipulating timing behavior

- Scripting
 - encapsulating circuit details
 - generating test cases and running simulations
 - invoking and controlling model checkers
 - scripting theorem proving strategies
 - programming exploration of counterexamples

- Tool Building
 - prototyping model checking strategies
 - making new deductive theorem proving procedures

12

Evaluation

- Scalar data

```
: defix '+';
- :: bool list -> bool list -> bool list
: let a = [F,T,T,T]; // 0111
a :: bool list
: let b = [F,F,F,T]; // 0001
b :: bool list
: a '+' b;
[T,F,F,F] :: bool list
:
```

13

Symbolic Evaluation

- Symbolic data

```
: let A = map variable ["a3","a2","a1","a0"];
A :: bool list
: let B = map variable ["b3","b2","b1","b0"];
B :: bool list
: A '+' B;
[b2&b3&a2&a3 + b1&b3&a1&a2&a3 + b0&b3&a0&a1&a2&a3 +
!b0&!b1&!b2&!b3&a3 OR ... ,
b1&b2&a1&a2 + b0&b2&a0&a1&a2 + !b0&!b1&!b2&a2 +
!b1&!b2&!a0&a2 OR ... ,b0&b1&a0&a1 +
!b0&!b1&a1 + !b1&!a0&a1 + !b0&b1&!a1 +
b0&!b1&a0&!a1 OR ... ,!b0&a0 + b0&!a0] :: bool list
:
```


14

Reflection - reFLect

- Reflection

- programs are data, in the same language
- programs can construct and analyze programs
- programs can run the constructed programs

- Like LISP/ACL2 quotation, but with types:

$\langle 1 + 2 \rangle : \text{term}$ - an abstract syntax tree: 

$\langle 1 + \wedge(2) \rangle = \langle 1 + 2 \rangle$ - term splicing

`let swap $\langle \wedge x + \wedge y \rangle = \langle \wedge y + \wedge x \rangle$` - pattern matching

15

Why Reflection?

- For our Forte applications, we want to
 - reason about arbitrary FL programs
 - reason about FL specifications of hardware
 - analyze & transform hardware models written in FL
 - intimately combine theorem proving and program execution
 - intimately combine model checking and theorem proving
- All require programs to access **syntax** of FL programs.
- Exploit reflection in theorem proving
 - term language of logic = reFLect = theorem prover metalanguage
 - evaluation in reFLect \Rightarrow provable equation in theorem prover

16

Motivating Example - FP Adder

- FPU from Intel Pentium Pro processor
 - large block of RTL from architects and designers
 - highly optimized for speed
 - to be verified 'as is', with no modification of design
- Functionality to verify
 - IEEE-compliant FP addition and subtraction
 - multiple precision: single, double, extended
 - all rounding modes: towards 0, $+\infty$, $-\infty$, to nearest

17

Proof Strategy Overview

High-level Specification
IEEE 754



Reference Model



FADD RTL

- Create a **reference model**
 - must be formal
 - must be executable
- Show that the model
 - satisfies the IEEE spec
 - theorem proving
 - is **equivalent** to the RTL
 - STE model checking

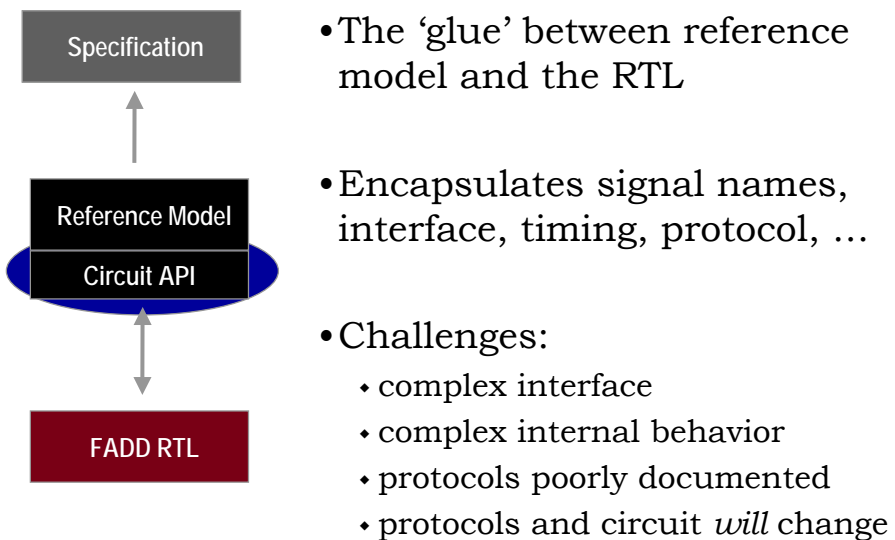
18

Example – Rounding Specification

```
let RND pc rc s sgf =
  // Extract lsb, guard, round sticky bits.
  let L = Lsb pc sgf in
  let G = Guard pc sgf in
  let RS = RoundS pc sgf in
  // Conditionally add one to LSB
  let rbit =
    (rc '=' TO_ZERO) => F
  | (rc '=' TO_POS_INF) => ((NOT s) AND (G OR RS))
  | (rc '=' TO_NEG_INF) => (s AND (G OR RS))
  | (rc '=' TO_NEAREST) => (RS => G | (L AND G))
  | F in
  // Result truncates mantissa to precision specified
  // by pc, adds rbit and pads result with zeros.
  Result rbit pc sgf;
```

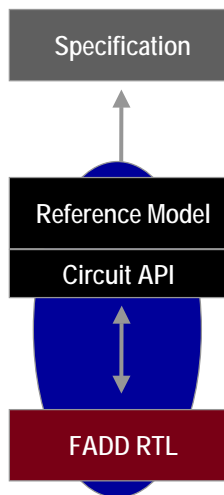
19

Circuit API

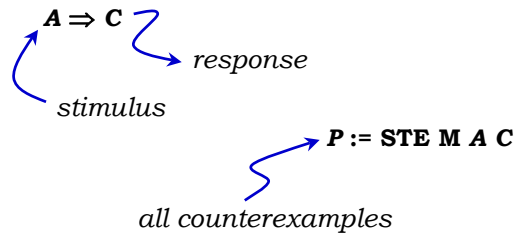


20

Verifying RTL



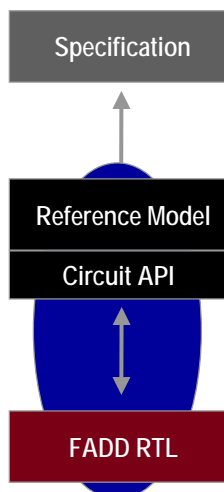
- Symbolic trajectory evaluation



- Based on symbolic simulation
 - smooth transition between simulation and verification
- Abstraction through X values

21

Technical Challenges



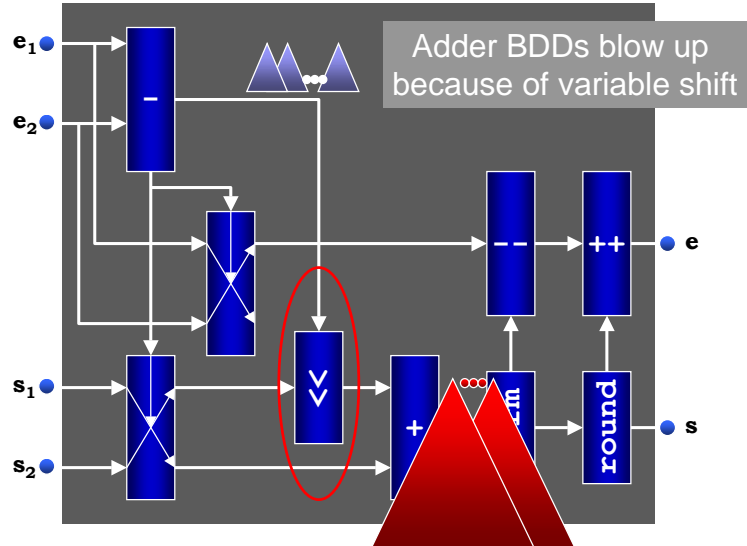
- Key technical insights

- Split input data space to avoid data-dependent shifts
 - Use *parametric representation*
- Overlay STE verifications with different BDD orderings
 - one for LZA circuit, one for result
- Certain other technical devices...

- Methodology & tool do not replace scientific insight, technical skill, and innovation!

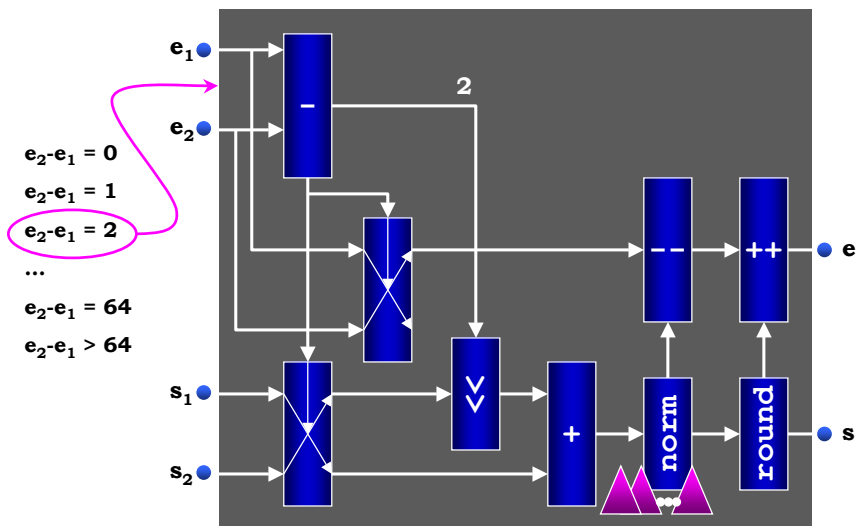
22

Data-Dependent Shifts



23

Solution - Input Case Splitting



24

Combining Cases - Theorem Proving

Goaled
Theorem
Prover

$\langle \Lambda \rangle \vdash n = p$

$\Lambda \quad n \rightarrow p$

reFlect
Interpreter

STE inference rules

$\vdash \exists h. \text{STE ckt } h \text{ A C}$

$\vdash \exists h. \text{STE ckt } h \text{ A B}$

$\vdash \exists h. \text{STE ckt } h \text{ B C}$

logic

$\vdash \text{STE ckt opt1 A B}$

$\vdash \text{STE ckt opt2 B C}$

$\text{STE ckt opt1 A B} \rightarrow \text{True}$

$\text{STE ckt opt2 B C} \rightarrow \text{True}$

25

Extracting Logical Content

- Parametric

- represent predicate

$$P \subseteq B \times B \times \dots \times B$$

by function with image P .

- Example

$a \ b \ c \ d$
 $\{(1, 0, 0, 1),$
 $(1, 0, 0, 0),$
 $(0, 1, 0, 1)\}$

$p, \neg p, 0, \neg p \vee q$

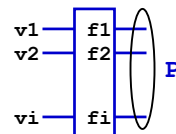
- Parametric verification

- want to check

$$P[\mathbf{xs}] \supset \text{STE } A[\mathbf{xs}] \text{ C}[\mathbf{xs}]$$

- parametric representation:

$$\mathbf{fs}[\mathbf{vs}] = \text{param}(\mathbf{xs}, P[\mathbf{xs}])$$

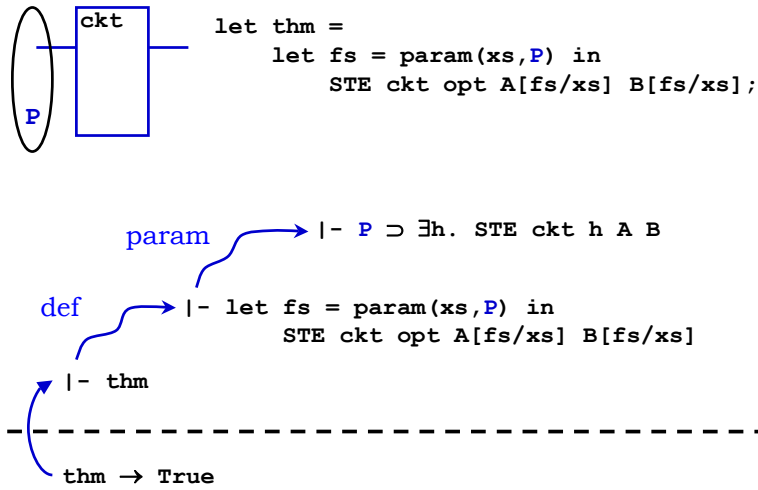


- more efficient verification:

$$\text{STE } A[\mathbf{fs}[\mathbf{vs}]] \text{ C}[\mathbf{fs}[\mathbf{vs}]]$$

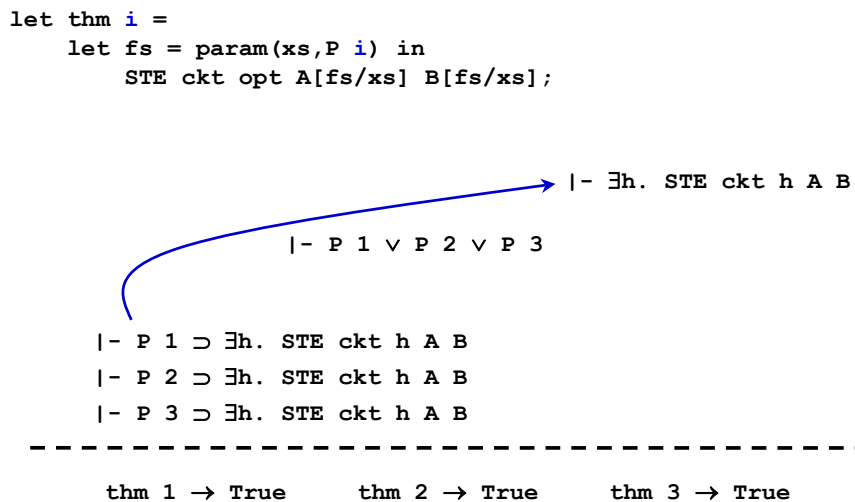
26

Input Constraints



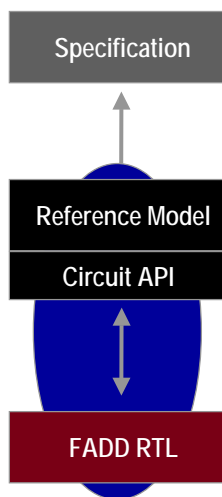
27

Input Case Splitting



28

The Completed RTL Verification



- Final proof
 - 342 cases, verified in parallel on workstation network
 - scripted in FL
- Reuse on subsequent designs
 - reference model
 - case splitting strategy
 - circuit API
- Now a routine technique, usable by non-experts

29

Some Forte Conclusions

- Industrial-scale formal hardware verification
 - an interactive, *programming*, activity
 - deep insight into tool capabilities *and* the design
 - essential to separate essence from circuit details
 - human comprehension of and reasoning about essence
 - contain and automate away the accidental details
- Explicit methodology principles, realistic
- Effective tools
 - open framework, heavily customised by scripting
 - specifically supports the methodology/approach

30

'Formal Methods' - on Hard Problems

- Not automatic, not manual
- Human
 - See and express the essence of the design analysis in domain-friendly terms
- Machine
 - Execute/enact the analysis, 'automating away' formal & implementation details
- Verification tool support
 - express the needed abstractions
 - fluently explore verification strategies
 - tie abstractions+strategies to the implementation
 - open, programmable, tool
 - white-box integration of components in a scripting environment

31

Thank You