Using a reflective functional language for hardware verification and theorem proving

APPSEM 2005, Frauenchiemsee

John O'Leary
Principal Engineer
Strategic CAD Labs, Intel
john.w.oleary@intel.com

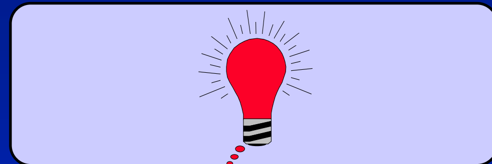
intel.

Acknowledgments

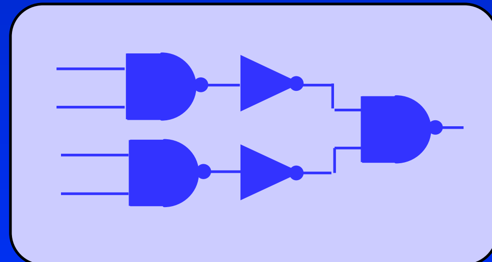
- Joint work with
  - Jeremy Casas
  - Jim Grundy
  - Robert Jones
  - Sava Krstic
  - Carl Seger
  - Tom Melham (Oxford)

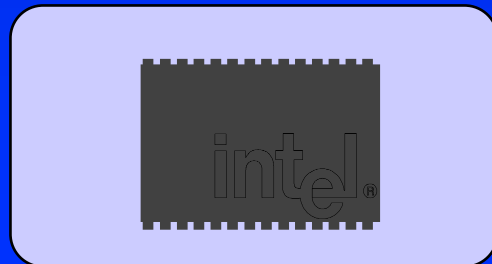
intel.

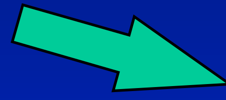
Hardware Verification

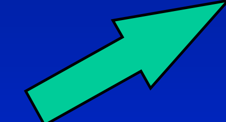


```
behavior of nand4 is
begin process(a,b,c,d)
 begin
   z<=NOT(a AND b AND c AND d)
 end process
end architecture behavior
```

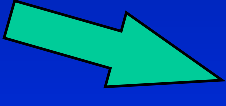


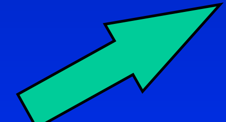




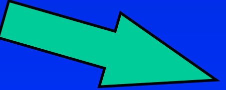


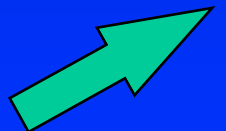
=? *Property Verification*





=? *Equivalence Verification*



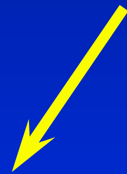


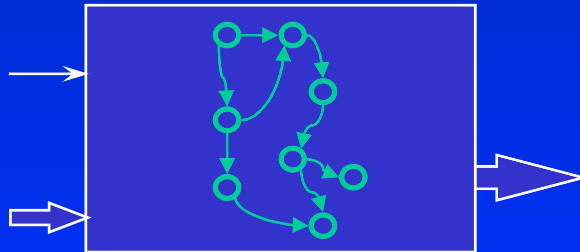
=? *Testing*

intel

Formal Approaches

Property Verification



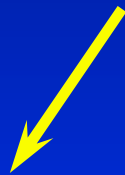


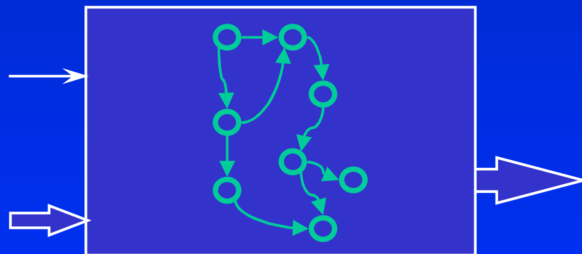
Algorithmic

intel.

Formal Approaches

Property Verification



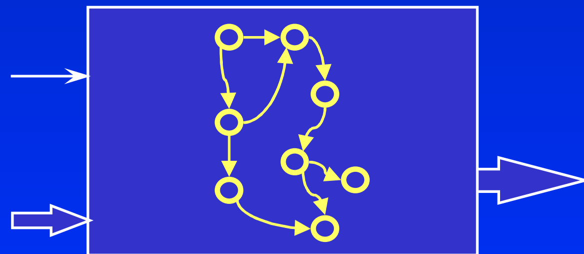


Algorithmic

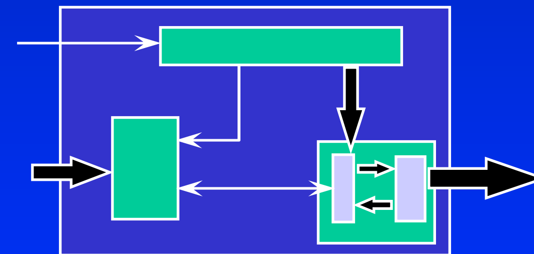
- **Combinational comparison**
- **State machine comparison**
- **Language containment**
- **Symbolic model checking**
- **Symbolic trajectory evaluation**

Formal Approaches

Property Verification



Algorithmic



Deductive

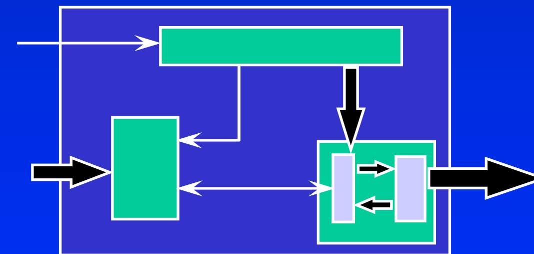
intel

Formal Approaches

Property Verification

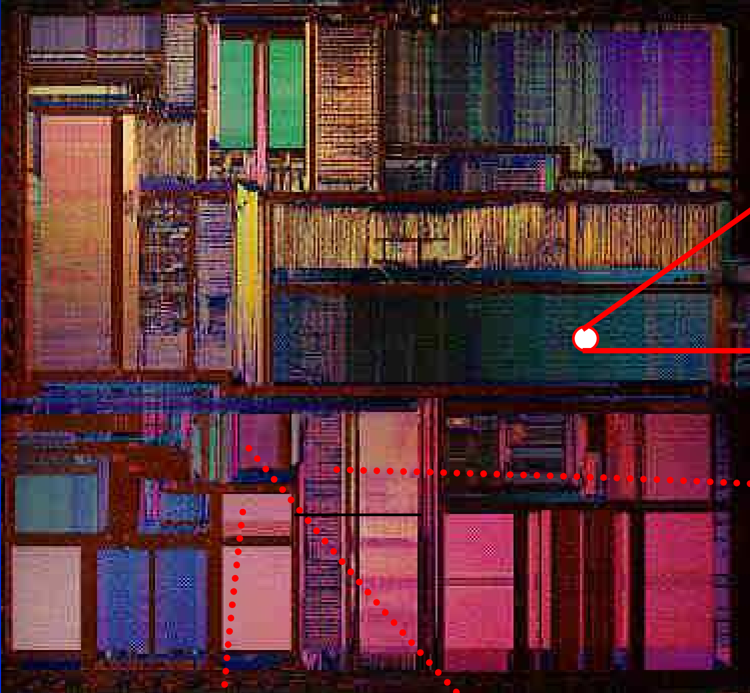
Mechanized Theorem Proving

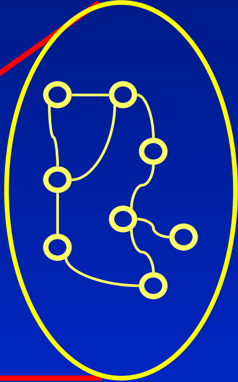
- **HOL**
- **ACL2**
- **Coq**
- **PVS**



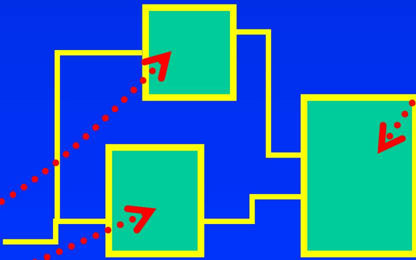
Deductive

Combined Techniques





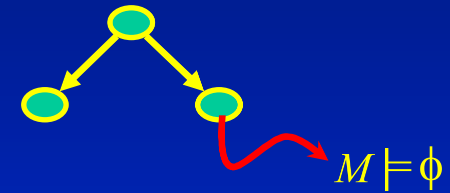
$\models$ **P**



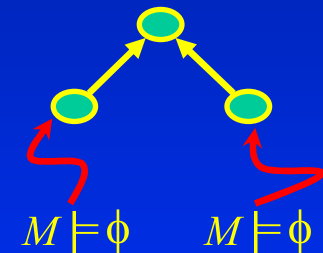
$\vdash$ **P**

Combined Techniques

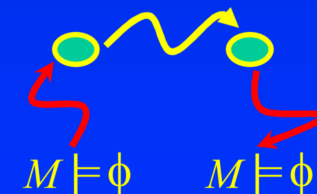
- Aims and use:
  - Model checker as decision procedure



  - Extend the reach of model checker



  - Improving model checking efficiency
  - Reasoning about specifications



intel.

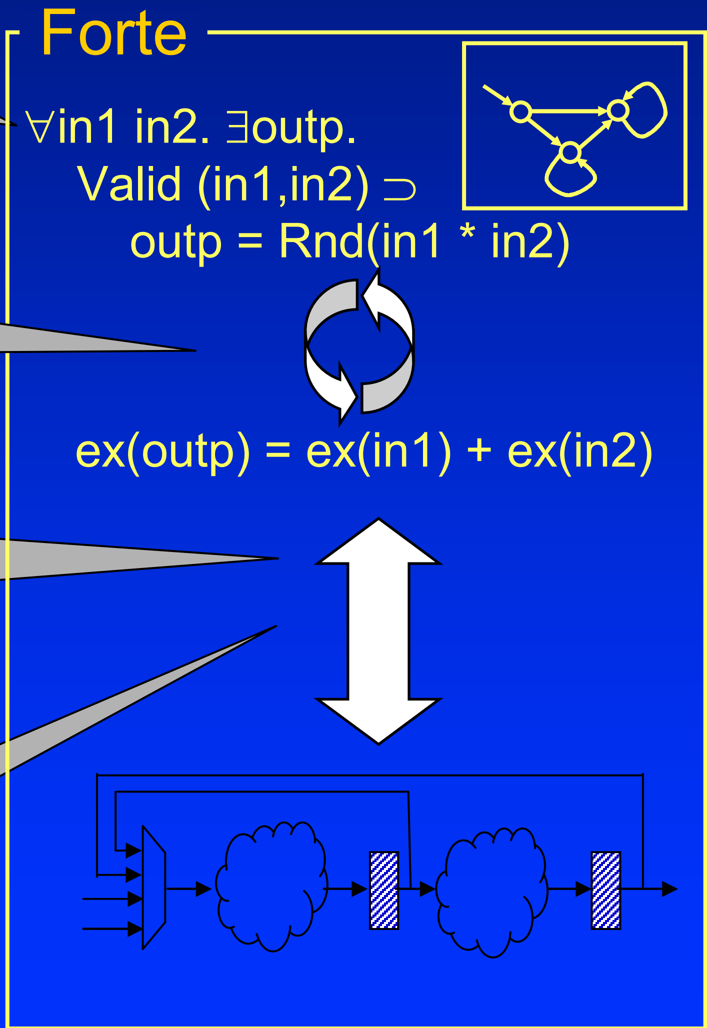
Forte System Architecture

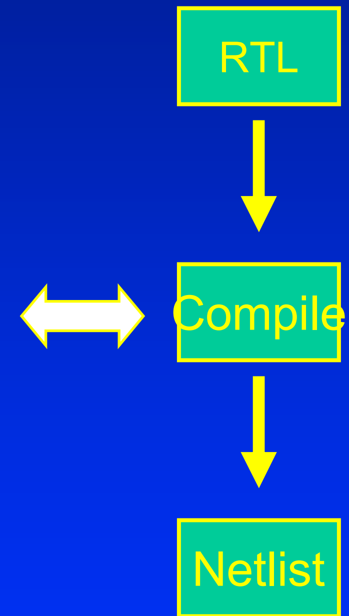
reFLect: Property specification, scripting, tool implementation

Goaled: Reasoning about reFLect programs

(G)STE: model checking for temporal properties

SMT: verification of higher-level models





Outline

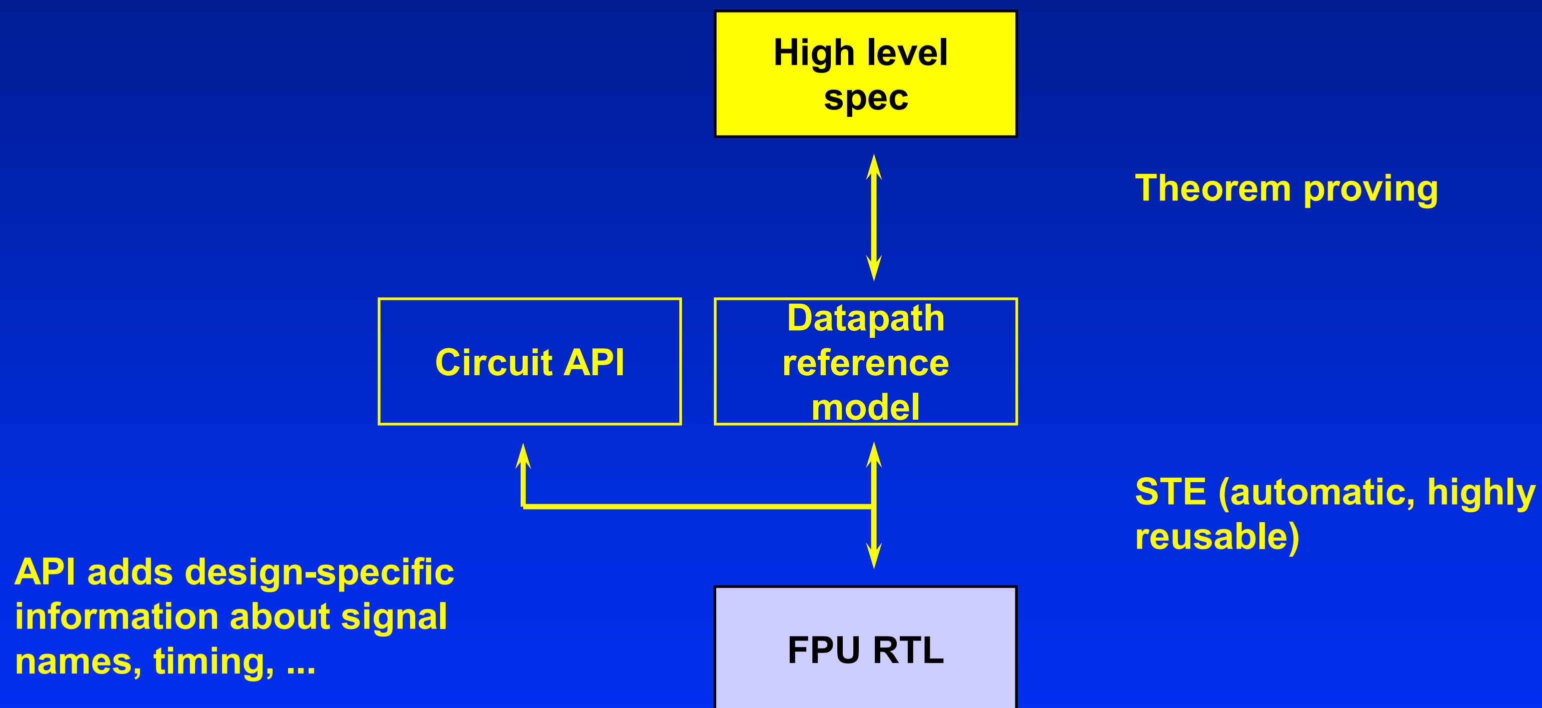
- Hardware verification in the Forte system
  - Floating-point verification
  - Sketch of a typical verification exercise
  - Hardware verification as (functional) programming
- reFLect: a reflective functional language
  - Overview
- Goaled: a theorem prover for reFLect programs
  - Logic of reFLect programs
  - Implementation
  - Integrating evaluation (via reflection) with theorem proving
- Conclusions

intel.

Example: verifying floating-point RTL

- Specify at level of IEEE standard
  - Use infinite precision, unbounded range arithmetic operations
- Verify:
  - All floating-point micro-operations implemented in hardware (but not software or microcode)
  - All processor modes, rounding modes, and precisions
  - Correct generation of flags, faults
- Verify the fully-optimized gate-level RTL description used to generate Si
  - That's what has to be correct when the product ships
  - Formal link to transistor-level schematic
- Replace informal arguments with a machine-verified proof of correctness
  - Automate low-level steps using model checking
  - Carry out model checking, theorem proving in a unified framework

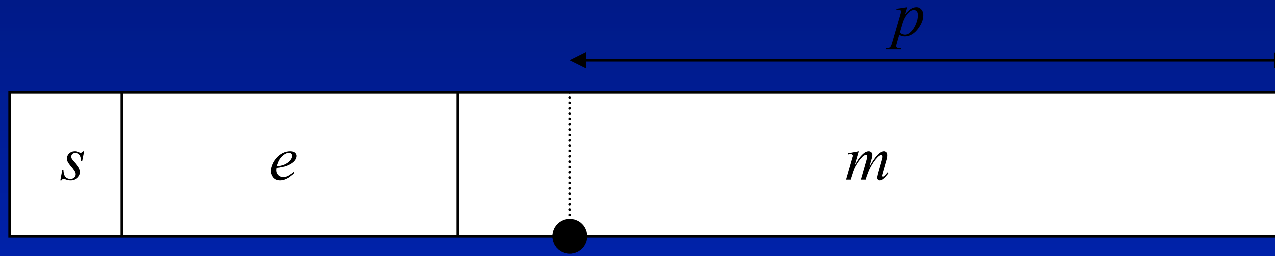
Strategy



Why use a functional language (I)

- The features we need are not well supported by hardware spec languages like PSL, SVA, etc
  - Support for datatypes more abstract than bits and bit vectors
  - Interpreter to facilitate rapid debugging of specifications and proof scripts
  - Powerful structuring techniques to improve modularity and reusability of specs
- Deductive reasoning is inevitable
  - Small number of constructs and well-understood semantics make deductive reasoning tractable

Floating point representation



- p is number of fractional bits, exponent is biased
- Each floating point bitstring maps to a rational number:

$$R = \frac{s * m * 2^{e}}{2^{p} * 2^{bias}}$$

High level specification

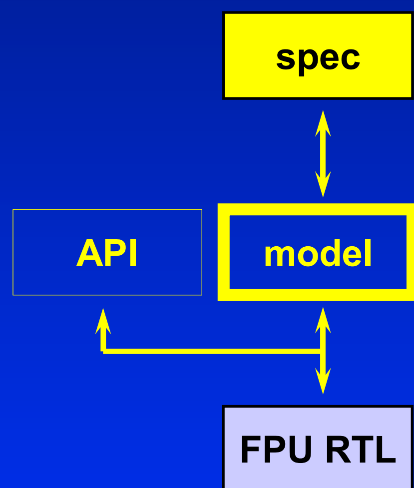
- IEEE Std 754 says:
  - "... each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range, and then [rounded] this intermediate result to fit in the destination's format." (section 5)
  - "When rounding toward negative infinity, the result shall be the format's value closest to (and no greater than) the infinitely precise result." (section 4.2)
- Infinite-precision result V::rat and rounded result R::rat must be related by

$$\underbrace{R \leq V}_{\text{No greater than}} \wedge \underbrace{V < R + ulp'}_{\text{Closest to}}$$

No greater than **Closest to**

intel.

Datapath reference model



- Captures numerical function of the datapath
- Purely algorithmic
- Written in reFLect
- Challenges:
  - Many operating modes, precisions
  - Intended function can be unclear

Datapath reference model

- Adapted a textbook algorithm as a first approximation
  - supported double precision, rounding toward 0, true addition
- Extended the model, driven by interactive execution targeted at interesting cases
  - ex: 1.0E24 - 1.0E1 should yield
    - 1.0E24 when rounding toward +∞ or nearest
    - 1.111…E23 when rounding toward -∞ or zero
- Interpreted programming language greatly simplifies debugging
  - Read/eval/print loop allows rapid evaluation of model on test cases
  - Specialized debugging routines can be consed up quickly
  - Compare the overhead of writing test benches and running simulations in a conventional hardware language

Example – FP subtraction

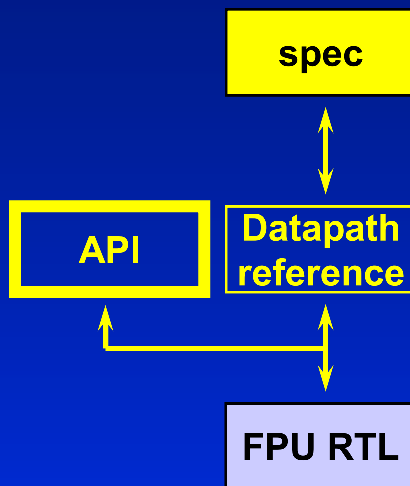
```
valid src1 AND valid src2 ==>
   (rounding_mode = TO_NEG_INF) ==>
     $r <= $src1 - $src2 < $r + ulp'  ...
```

High level spec

Datapath reference

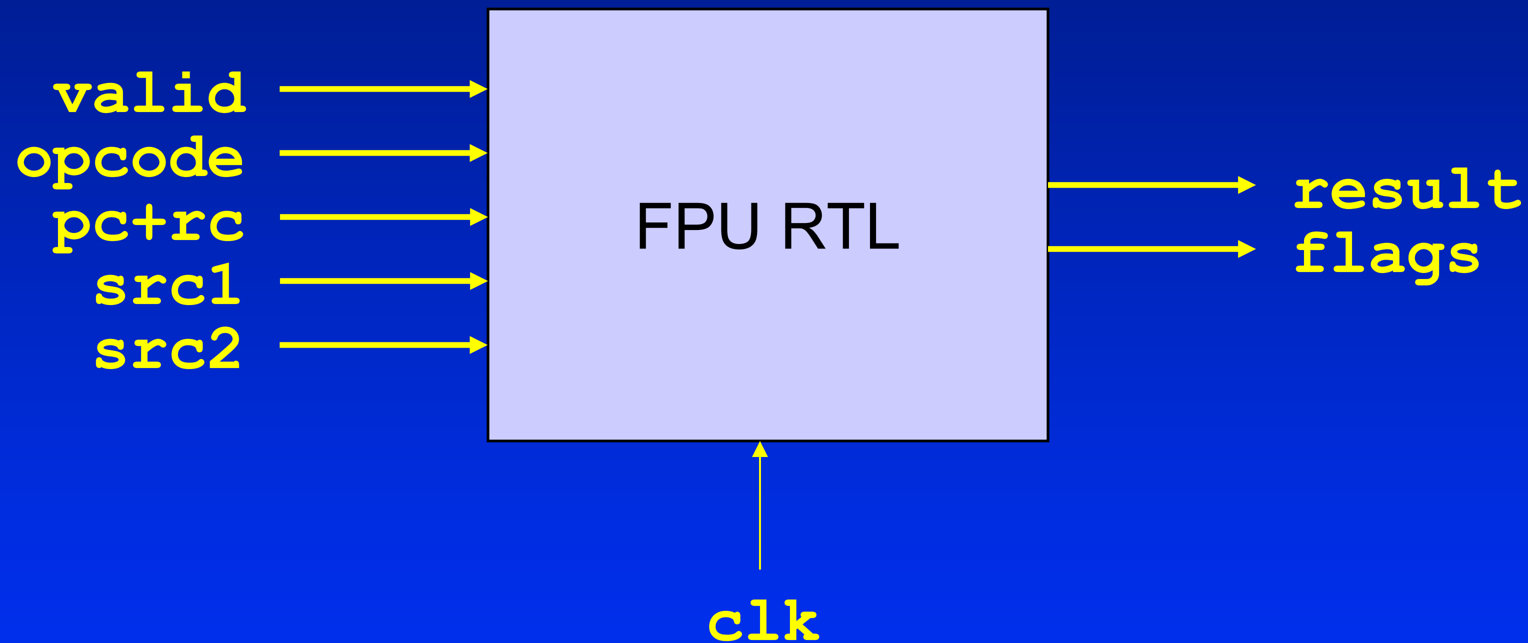
```
let SUB (vld,opcode,pc,rc,src1,src2) =
  ...
  // Find the amount of shift needed
  let diff = ex2 '-' ex1  in
  let rsh = MINv diff (nat_to_bv 17 68) in
  // Do the shift
  let sgf1' = srshift 68 rsh (sgf1@[F]) in
  let sgf2' = sgf2@[F] in
  // Perform the sum (or subtract)
  let add = (sign fp1 != sign fp2) in
  let sum = IF add THENv (sgf2' '+' sgf1')
                   ELSEv (sgf2' '-' sgf1')
  // Now perform rounding
```

Circuit API



- The glue between our datapath reference models and the RTL
- Captures timing, interface protocol, …
- Programmed in reFLect
- Challenges:
  - Complex, design-dependent interface
  - Complex internal behavior
  - Protocols are poorly documented, and will change
- Developed during the initial phase of verification and evolving with RTL changes

Design-specific interface behavior



- Define a “transaction” abstraction along with mappings to concrete signals

Transaction

- A tuple containing the important facts about an incoming/outgoing operation

```
inp_trans:
  (
    valid,                    // valid bit
    opcode,                   // what operation?
    pc,                       // precision control
    rc,                       // rounding control
    src1,                     // operand 1
    src2                      // operand 2
  )

outp_trans:
  (
    result,
    flags
  )
```

intel.

Circuit API

```
API :: inp_trans -> (inp_trans->outp_trans) -> (A # C)

let API trans f =
  val (vld,opcode,pc,rc,src1,src2)= trans in
    (
    // circuit inputs
      ("op_vld" is vld in_phase (clk, t)) and
      ("op_code" isv opcode in_phase (clk, t)) and
      ("pre_ctl" isv pc in_phase (clk, t+1)) and
      ("rnd_ctl" isv rc in_phase (clk, t+1)) and
      ("in_data1" isv src1 in_phase (clk, t+1)) and
      ("in_data2" isv src2 in_phase (clk, t+1))
    ,
    // circuit outputs
      ("result" isv fst (f trans) in_phase (clk, t+4))
      and
      ("flags" isv snd (f trans) in_phase (clk, t+5))
    ) ;
```

intel.

Circuit API

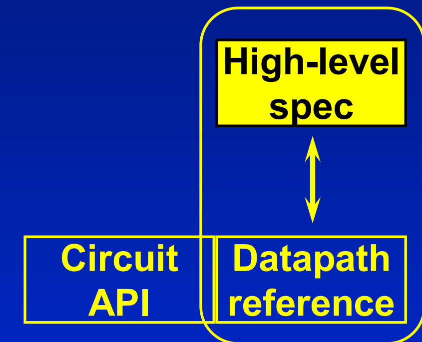
```
API :: inp_trans -> (inp_trans->outp_trans) -> A # C

let API trans f =
  val (vld,opcode,pc,rc,src1,src2)= trans in
    (
     …
      ("result" isv fst (f trans) in_phase (clk, t+4))
      and
      ("flags" isv snd (f trans) in_phase (clk, t+5))
    ) ;
```

- API is a higher order function taking the datapath reference model as an argument
- API is reusable without change to verify other operations implemented on the same hardware
- By tweaking the API, the datapath reference models can be used unchanged on new designs

intel.

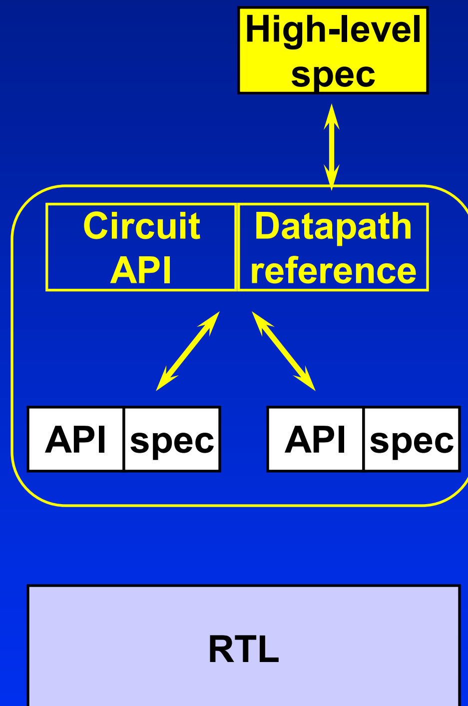
Verification



- Verifies IEEE-compliance of the datapath reference
- Algorithm verification
  - Done via user-guided theorem proving
  - Theory of bitstring arithmetic bridges the gap between circuit bit vectors (lists of booleans) and integers
  - Further arithmetic reasoning proves final result

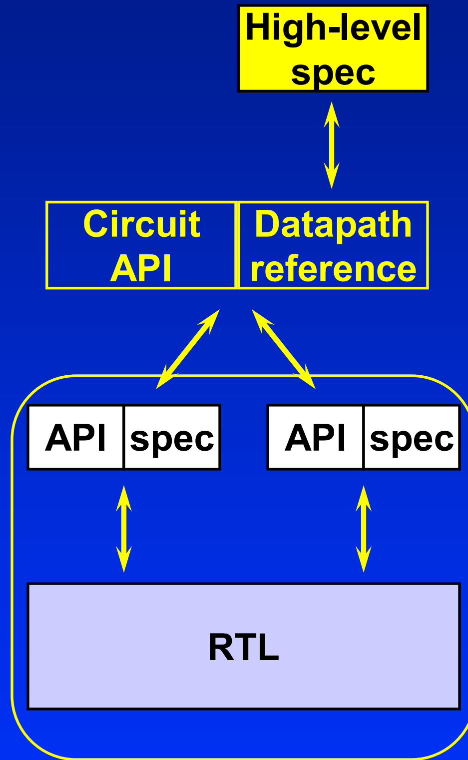
intel

Verification



- Decomposition strategy dictated by FSUB complexity issues
  - data-dependent shifts in bit-level spec and RTL
  - BDD blowup in RTL's LZA circuitry
- Key insights:
  - split input data space to avoid data-dependent shifts
  - employ two STE verifications with different BDD orderings (one for LZA circuit, one for result)
- Decomposition is verified in theorem prover
  - Logic of STE specifications with pre/post conditions
  - Proves exhaustiveness of data space decomposition

Verification



- Automatic verification using symbolic trajectory evaluation (STE)
- 342 cases, each covering one area of the input data space
- Executed in parallel on workstation network

intel.

Verification in practice

- Regression
  - RTL is “live”, and changes frequently until the very final stages of the project
  - Model checking automation at lower levels allows regression to be automated and provides robustness in the face of ECOs
- Debugging
  - Symbolic trajectory evaluation (like symbolic simulation) is intuitive for designers and architects
  - Formal verification counterexamples are easily simulated
  - Forte’s waveform viewer, schematic browser, etc aid in debugging RTL/spec disagreements in terms that designers and architects can relate to
- Verification in the large
  - Proof design: how do we solve verification problems systematically?
  - Proof engineering: how do we write maintainable and modifiable proofs?

Why use a functional language (II)

- Functional languages have the features we need for specification and reasoning
  - Support for datatypes more abstract than bits and bit vectors
  - Interpreter facilitates rapid debugging
  - Higher order functions that improve modularity and reusability
  - Support for deductive reasoning
- Verification turns out to be a programming activity
  - Specifications and proofs must be systematically developed and evolved
  - Substantial “scriptology” involved in running the model checker, regressing the proofs, debugging
  - Use a language that is a help rather than a hindrance

intel.

Some Forte Verifications

- Verification of gate-level floating point implementations (FADD, FSUB, FMUL, FDIV, FSQRT, FCOM, etc) vs IEEE-level spec
  - [O'Leary et al, ITJ'99; Aagaard et al, DAC'00; Kaivola+Aagaard, TPHOLs'00; Kaivola+Kohatsu, CHARME'01; Narasimhan+Kaivola, DATE'02]
- Verification of iA32 instruction-length decoder vs Pentium PRM
- We have used these techniques on two successive generations of lead iA32 processors and their proliferations
- Pentium® 4 processor verification effort found several *very high quality* pre-Si bugs
  - FP Multiply dataspace bug
  - Inadvertent interaction between FP operations in different threads resulting in data corruption
  - FP flags (overflow/underflow) incorrect in certain processor modes
  - Overall FV effort (not just FPU, not just Forte) found ~20 high quality bugs that would have been hard to detect by dynamic testing

Outline

- Hardware verification in the Forte system
  - Floating-point verification
  - Sketch of a typical verification exercise
  - Hardware verification as (functional) programming
- reFLect: a reflective functional language
  - Overview
- Goaled: a theorem prover for reFLect programs
  - Logic of reFLect programs
  - Implementation
  - Integrating evaluation (via reflection) with theorem proving
- Conclusions

intel

Role of FL in Forte

- Specification
  - reference modeling (e.g. floating point operations)
  - stipulating I/O behavior
  - stipulating timing behavior
- Scripting
  - generating test cases and running simulations
  - invoking and controlling model checkers
  - scripting theorem proving strategies
  - programming exploration of counterexamples
- Tool Building
  - Implementing model checkers, formal equivalence checkers
  - making new deductive theorem proving procedures

A New FL for Forte - reFLect

- New functional language for Forte:

  reFLect = FL + reflection features

- Reflection
  - programs are data, *in the same language*
  - programs can construct and analyze programs
  - programs can run the programs they construct

Think LISP/ACL2 quotation, but with types.

intel.

Why Reflection?

- For our Forte applications, we want to
  - reason about FL specifications of hardware
  - intimately combine theorem proving and program execution
  - intimately combine model checking and theorem proving
  - reason about arbitrary FL programs
  - embed conventional specification languages into FL
  - analyze & transform hardware models written in FL

  All require programmatic access to *syntax* of programs.

- reFLect implementation is tuned to support
  - Goaled (our higher order logic theorem prover)
  - IDV (Integrated Design and Verification environment)

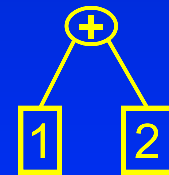
The reFLect Language

- Approximate Syntax

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

- Examples

$\langle 1 + 2 \rangle$ - an abstract syntax tree:



$\langle 1 + \hat{}\langle 2 \rangle \rangle = \langle 1 + 2 \rangle$ - term splicing

$(\lambda \langle \hat{}x + \hat{}y \rangle.\ \langle \hat{}y + \hat{}x \rangle)\ \langle 1 + 2 \rangle = \langle 2 + 1 \rangle$

- pattern matching β-reduction

intel.

Technicalities

- Syntactic subtleties

$$(\lambda\langle \hat{}x + \hat{}y\rangle.\ z)\ [z \to \langle \hat{}x\rangle] \ = (\lambda\langle \hat{}x' + \hat{}y\rangle.\ \langle \hat{}x\rangle)$$

- Type System

$$\frac{\vdash \Lambda : \mathit{term} \qquad \vdash C[\mathrm{v}:\sigma] : \tau}{\vdash \langle\ \mathrm{C}[\hat{}\Lambda:\sigma]\ \rangle\ :\ \mathit{term}}$$

- Reduction

$$\frac{\vdash \Lambda : \phi(\sigma)}{\vdash \langle\ C[\hat{}\langle\Lambda\rangle :\sigma]\ \rangle \to \langle\ C_{\phi}[\Lambda]\ \rangle}$$

$$\frac{p \text{ ready } a \qquad (p,\theta) \text{ matches } a}{\vdash (\lambda p.\ n \,\square\, o)\ a \to n\theta}$$

See JFP paper for details.

Examples

- Abstract syntax functions

let mk_apply f x **=** ⟨**^**f **^**x⟩

let dest_apply ⟨**^**f **^**x⟩ **=** (f, x)

```
let mk_apply f x = {|`f `x|}
let dest_apply {|`f `x|} = (f, x)
```

Maths	ASCII
$\lambda p.\ n$	`\p.n`
$\langle\ n\ \rangle$	`{| n |}`
$\hat{}n$	`` `n ``

- Swap operands of plus:

```
letrec S {| `x + `y |} = {| `(S y) + `(S x) |}
  /\   S {| `f `x |}   = {| `(S f) `(S x) |}
  /\   S {| \`p. `b |} = {| \`p. `(S b) |}
  /\   S x = x
```

Reflection in reFLect

- Evaluation

  $\vdash$ eval : $term \rightarrow term$

- Value

  $\vdash$ value$_\sigma$: $term \rightarrow \sigma$

- Lifting

  $\vdash$ lift : $\sigma \rightarrow term$

- Examples

  eval $\langle (\lambda x.x)\ 1 \rangle \rightarrow \langle 1 \rangle$

  value$_{int} \langle 1+2 \rangle \rightarrow 1+2 \rightarrow 3$

  value$_{\alpha\rightarrow\alpha} \langle \lambda x.x \rangle \rightarrow$ 'identity function'

  lift $1 \rightarrow \langle 1 \rangle$

  lift $(1+2) \rightarrow \langle 3 \rangle$

  lift $(\lambda x.x) \rightarrow \langle \boxed{\lambda x.x} \rangle$

reFLect – {lift,value,eval} has subject reduction, strong normalization, and Church-Rosser properties. Theoretical properties of full reFLect are under investigation.

Implementation

- In heavy, daily use at Intel since January 2002
  - Intel has used reFLect's predecessor since 1995
- Several major tools based on reFLect
  - Next generation formal equivalence verification system
  - IDV, a transformation-based hardware design environment
  - Goaled (our next topic)
- Forte (reFLect + model checkers) is available for non-commercial use
  - http://www.intel.com/software/products/opensource/tools1/verification/download.htm

intel

Outline

- Hardware verification in the Forte system
  - Floating-point verification
  - Sketch of a typical verification exercise
  - Hardware verification as (functional) programming
- reFLect: a reflective functional language
  - Overview
- Goaled: a theorem prover for reFLect programs
  - Logic of reFLect programs
  - Implementation
  - Integrating evaluation (via reflection) with theorem proving
- Conclusions

intel

Higher Order Logic of reFLect

- HOL, following Church:

  Logic = { λ-calculus + logical constants + rules }

- The reFLect logic:

  Logic = { reFLect + logical constants + rules }

- Basic idea in both systems:

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

  Define $\forall$, $\exists$, etc by axioms

  Add rules for function equality

  Reflection rules

  Axioms for built-in types and constants

intel.

Implementation

- Theorem = quoted term

  |- Λ means $\langle\Lambda\rangle$ is a theorem

```
lettype thm = |- term;
```

- Example, a reduction rule

```
> BETA_CONV {|(\x.f x y) a|};
|- (\x.f x y) a = f a y
```

- Example, code for $\wedge$-introduction

```
let CONJ (A1 |- P) (A2 |- Q) =
  A1 ∪ A2 |- {|`P AND `Q|}
```

intel.

Implementation

- LCF-style, following in the footsteps of 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)
- Theories of reFLect data types
  - Natural numbers, integers, rationals
  - Lists, pairs, reFLect ADTs
  - Bitstring arithmetic
- Proof automation
  - Unconditional and conditional (contextual) rewriting
  - First order solver based on model elimination
  - Universal linear arithmetic over N, Z, Q
  - Experimental link with CVC Lite (as oracle)

Proof by Evaluation

- Logical status of evaluation:
  - In the logic, we say $\vdash n = p$ if we know $n \to p$
  - Given n, the reFLect evaluator computes p such that $n \to p$
- We can use eval : *term* $\to$ *term* to reduce ground terms

```
let EVAL_CONV t =
  let u = eval t in
    mk_theorem [] {| ^t = ^u |};

> EVAL_CONV {| (\x.x+1) 2 |};
|- (\x.x+1) 2 = 3
```

- Fast proof by evaluation — using reFLect interpreter
  - Building block for decision procedures (e.g. quantifier elimination)
  - Generating Goaled theorems from model checking runs

intel.

Reasoning about specifications

Goaled Theorem Prover

```
|- STE ckt [] A C
```

STE inference rules

```
|- ∃h. STE ckt h A B
|- ∃h. STE ckt h B C
```

logic

```
|- STE ckt opt1 A B
|- STE ckt opt2 B C
```

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

EVAL_CONV

$\Lambda \quad n \rightarrow p$

```
STE ckt opt1 A B → True

STE ckt opt1 B C → True
```

reFLect Interpreter

intel

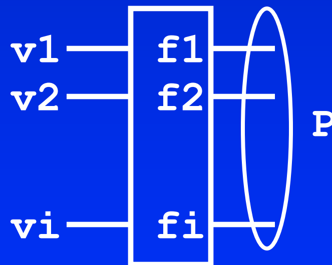
Making model checking efficient

- Parametric method
  - want to check

```
P[xs] ⊃ STE A[xs] C[xs]
```

  - parametric representation:

```
param(xs,P[xs]) = fs[vs]
     s.t. P[fs[vs]] = True
```



  - more efficient verification:

```
STE A[fs[vs]] C[fs[vs]]
```

- Verification uses
  - environment constraints
  - input case-splitting

- As a Goaled theorem

```
|- ∀P A C xs.
    (SAT xs P) ⊃
      (P ⊃ STE A C)
           =
      STE A[param(xs,P)/xs]
           C[param(xs,P)/xs]
```

  - explicitly links the reFLect logic and the embedded STE logic.

Rewriting with evaluation

- Conversions map terms to theorems
  - `BETA_CONV {|(\x.f x y) a|} = |- (\x.f x y) a = f a y`
  - `EVAL_CONV {|(\x.x+1) 2 |}  = |- (\x.x+1) 2 = 3`

- Rewriting is a clever application of conversions

```
let REWRITE_CONV thms =
    TOP_DEPTH_CONV (REWRITES_CONV thms)
```

Term traversal strategy (TOP_DEPTH_CONV)

Transforms a list of theorems to a conversion (REWRITES_CONV)

- Integrating evaluation with rewriting

```
let REWRITE_CONV thms =
    TOP_DEPTH_CONV (REWRITES_CONV thms
                        ORELSE_CONV EVAL_CONV)
```

intel.

Rewriting with evaluation

- Efficient reduction of constant subterms and side conditions

```
bvless_thm:
|- ∀ as bs.
      (length as = length bs)
        ==>
          (bvless as bs  =
           bv2int as < bv2int bs)
```

```
{| C[bvless bv1 bv2] |}
```

Applying **bvless_thm**

```
|- length bv1 = length bv2 ==>
      C[bvless bv1 bv2] = C[bv2int bv1 < bv2int bv2]
```

EVAL_CONV

```
|- C[bvless bv1 bv2] = C[bv2int bv1 < bv2int bv2]
```

intel

Decision procedures with evaluation

- Solving linear arithmetic goals by variable elimination

```
Forall x.
  x > 1001r ==> 2r*x > 1001r
```

Negate, canonize and attempt to refute

```
Exists x.
  (-1r)*x + 1001r <= 0r AND
  2r*x + (-1001r) <= 0r
```

Eliminate quantified variable ‘x’

```
1001r2 <= 0r
```

EVAL_CONV

```
F
```

intel.

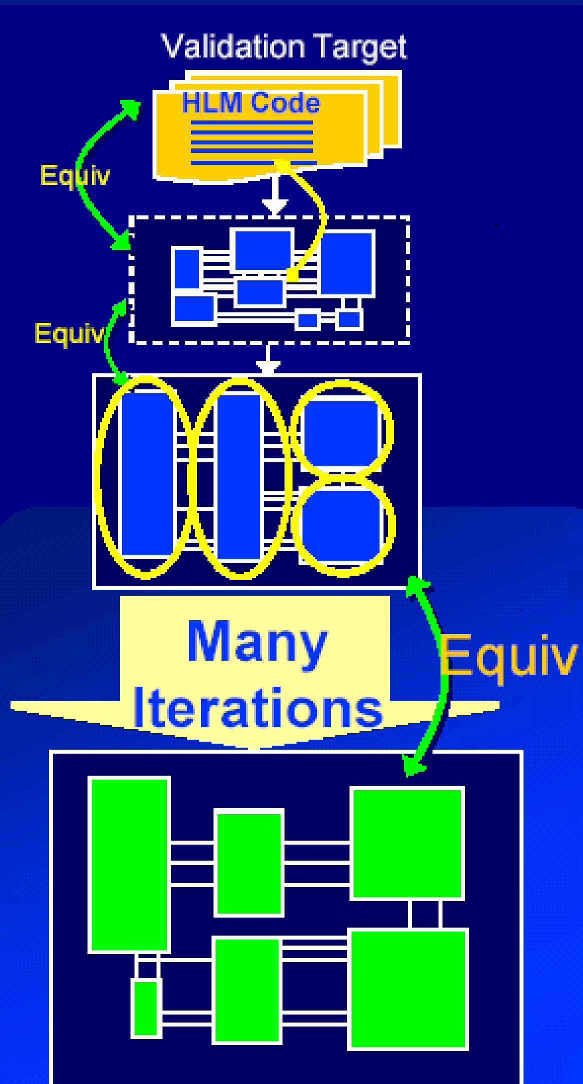
Outline

- Hardware verification in the Forte system
  - Floating-point verification
  - Sketch of a typical verification exercise
  - Hardware verification as (functional) programming
- reFLect: a reflective functional language
  - Overview
- Goaled: a theorem prover for reFLect programs
  - Logic of reFLect programs
  - Implementation
  - Integrating evaluation (via reflection) with theorem proving
- Conclusions

Summary

- Practical, scalable hardware verification…
  - Is a programming activity and requires a programming language
  - Needs a mixture of algorithmic and deductive techniques – the language had better be a clean one
- reFLect: a typed functional language with reflection
  - Specification, scripting and tool-building language that's amenable to formal reasoning
- Goaled: a theorem prover for reasoning about reFLect programs
- This is semantics, applied

Use of reFLect in hardware design



[Spirakis DAC'03, DATE'04]

- Raise the level of abstraction by creating a high level model (HLM)
- Successively refine the design with formal verification ensuring the correctness of each refinement
- Tightly integrate logical and physical design
- Store and re-use transformations developed during the design process
- Final implementation maintains equivalence to HLM while meeting design goals
- A prototype Integrated Design and Verification system based on these ideas has been built in reFLect

Other current and future work

- Theory
  - Semantics of “full” reflect (with lift and value)
- Integration of SMT solvers with reFLect/Goaled
- Symbolic and partial evaluation of reFLect programs
- Applications of reflection
  - Analysis of designs modeled in reFLect
    - Area, timing and power estimation
    - Finding abstraction opportunities for formal verification
  - Design transformations
    - Synthesis
    - Optimization
  - Proving correctness of decision procedures
- Verification applications
  - Processors
  - Protocols
  - Embedded software

Related & Other Work

- Lifted-FL, earlier version of reflection in FL
  - Aagaard, Jones, Seger: TPHOLs'99 (LNCS 1690)
- Language/Semantics/Theory of reFLect
  - Grundy, Melham, O'Leary: Oxford TR PRG-RR-03-16.
  - Krstic & Matthews: OGI TR CSE-04-014.
  - Ongoing work on denotational semantics at Intel and OGI.
- Metaprogramming frameworks
  - MetaML – Taha, Sheard: SIGPLAN Notices, 32(12), 2002.
  - Template Haskell – Sheard, Peyton Jones: Haskell'02
- Embeddings, formal design by refinement
  - Numerous papers in the literature.
- HDLs & reflection in LISP and ACL2
  - Brock, Hunt, Young: TPCD'92 and elsewhere.
- Others too numerous to mention (including some at APPSEM'05)

Selected References

GSTE: J. Yang and C.-J.H. Seger, "Introduction to Generalized Symbolic Trajectory Evaluation", *IEEE Transactions on VLSI Systems*, vol. 11, no.3 (June 2003), pp. 345-353.

J. Yang and C.-J.H. Seger, "Generalized Symbolic Trajectory Evaluation – Abstraction in Action", *Third Conference on Formal Methods in CAD*, Portland, OR, November 2002, pp. 70-87.

reFLect: J. Grundy, T.F. Melham, and J.W. O'Leary, "A Reflective Functional Language for Hardware Design and Theorem Proving", *Journal of Functional Programming,* in press.

Goaled: J.W. O'Leary, J. Grundy and T.F. Melham, "A Reflective Functional Language for Hardware Design and Theorem Proving", *Fifth Workshop on Designing Correct Circuits,* Barcelona, Spain, March 2004.

Forte: C.-J.H. Seger, R.B. Jones, J.W. O'Leary, T. Melham, M.D. Aagaard, C. Barrett, and D. Syme, "An Industrially Effective Environment for Formal Hardware Verification", *IEEE Transactions on Computer-Aided Design*, vol. 24, no.9 (September 2005), pp. 1381-1406.

Methodology: R. B. Jones, J.W. O'Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, "Practical Formal Verification in Microprocessor Design", *IEEE Design & Test of Computers*, vol. 18, no. 4 (July/August 2001), pp. 16–25.

Arithmetic: J.W. O'Leary, X. Zhao, R. Gerth, and C.-J.H. Seger, "Formally Verifying IEEE Compliance of Floating-Point Hardware", *Intel Technology Journal* (First Quarter, 1999).

Thank you

intel.